EBook Gratis

APRENDIZAJE C++

Free unaffiliated eBook created from
Stack Overflow contributors.

#c++

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cplusplus

It is an unofficial and free C++ ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C++.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con C ++

Observaciones

El programa 'Hello World' es un ejemplo común que se puede usar simplemente para verificar la presencia del compilador y la biblioteca. Utiliza la biblioteca estándar de C ++, con `std::cout` de `<iostream>` , y solo tiene que compilar un archivo, lo que minimiza la posibilidad de un error de usuario durante la compilación.

El proceso para compilar un programa C ++ difiere inherentemente entre compiladores y sistemas operativos. El tema Compilación y creación contiene los detalles sobre cómo compilar el código C ++ en diferentes plataformas para una variedad de compiladores.

Versiones

Versión	Estándar	Fecha de lanzamiento
C ++ 98	ISO / IEC 14882: 1998	1998-09-01
C ++ 03	ISO / IEC 14882: 2003	2003-10-16
C ++ 11	ISO / IEC 14882: 2011	2011-09-01
C ++ 14	ISO / IEC 14882: 2014	2014-12-15
C ++ 17	TBD	2017-01-01
C ++ 20	TBD	2020-01-01

Examples

Hola Mundo

Este programa imprime `Hello World!` al flujo de salida estándar:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

Véalo en vivo en Coliru .

Análisis

Examinemos cada parte de este código en detalle:

- `#include <iostream>` es una **directiva de preprocesador** que incluye el contenido del archivo de cabecera estándar de C ++ `iostream` .

  `iostream` es un **archivo de encabezado de biblioteca estándar** que contiene definiciones de los flujos de entrada y salida estándar. Estas definiciones se incluyen en el `std` nombres `std` , que se explica a continuación.

  Los **flujos de entrada / salida (E / S) estándar** proporcionan formas para que los programas obtengan entrada y salgan a un sistema externo, generalmente el terminal.

- `int main() { ... }` define una nueva función llamada `main` . Por convención, la función `main` se llama a la ejecución del programa. Sólo debe haber una función `main` en un programa de C ++, y siempre debe devolver un número del tipo `int` .

  Aquí, el `int` es lo que se llama el tipo de retorno de la función. El valor devuelto por la función `main` es un **código de salida.**

  Por convención, un sistema que ejecuta el programa interpreta como exitoso un código de salida del programa `0` o `EXIT_SUCCESS` . Cualquier otro código de retorno está asociado con un error.

  Si no hay ninguna declaración de `return` , la función `main` (y, por lo tanto, el propio programa) devuelve `0` de forma predeterminada. En este ejemplo, no necesitamos escribir explícitamente la `return 0;` .

  Todas las demás funciones, excepto aquellas que devuelven el tipo `void` , deben devolver explícitamente un valor de acuerdo con su tipo de retorno, o de lo contrario no deben devolverlo en absoluto.

- `std::cout << "Hello World!" << std::endl;` grabados "¡Hola mundo!" al flujo de salida estándar:

  - `std` es un espacio de nombres , y `::` es el **operador de resolución de alcance** que permite buscar objetos por nombre dentro de un espacio de nombres.

    Hay muchos espacios de nombres. Aquí, usamos `::` para mostrar que queremos usar `cout` desde el `std` nombres `std` . Para obtener más información, consulte Operador de resolución de alcance - Documentación de Microsoft .

  - `std::cout` es el objeto de **flujo de salida estándar** , definido en `iostream` , y se imprime en la salida estándar (`stdout`).

  - `<<` es, *en este contexto* , el **operador de inserción de flujo** , llamado así porque *inserta* un objeto en el objeto de *flujo* .

La biblioteca estándar define el operador `<<` para realizar la inserción de datos para ciertos tipos de datos en flujos de salida. `stream << content` inserta el `content` en el flujo y devuelve lo mismo, pero el flujo actualizado. Esto permite encadenar inserciones de secuencias: `std::cout << "Foo" << " Bar";` Imprime "FooBar" en la consola.

- `"Hello World!"` es una **cadena de caracteres literal** , o un "texto literal". El operador de inserción de flujo para los literales de cadena de caracteres se define en el archivo `iostream` .

- `std::endl` es un objeto especial de **manipulador de flujo de E / S** , también definido en el archivo `iostream` . Insertar un manipulador en un flujo cambia el estado del flujo.

  El manipulador de flujo `std::endl` hace dos cosas: primero inserta el carácter de fin de línea y luego vacía el búfer del flujo para forzar que el texto aparezca en la consola. Esto asegura que los datos insertados en la transmisión realmente aparezcan en su consola. (Los datos de transmisión generalmente se almacenan en un búfer y luego se "descargan" en lotes, a menos que se fuerce un vaciado de inmediato).

  Un método alternativo que evita la descarga es:

  ```
  std::cout << "Hello World!\n";
  ```

  donde `\n` es la **secuencia de escape de caracteres** para el carácter de nueva línea.

- El punto y coma (`;`) notifica al compilador que una declaración ha finalizado. Todas las declaraciones de C ++ y las definiciones de clase requieren un punto y coma finalizado.

Comentarios

Un **comentario** es una forma de colocar texto arbitrario dentro del código fuente sin que el compilador de C ++ lo interprete con un significado funcional. Los comentarios se utilizan para dar una idea del diseño o método de un programa.

Hay dos tipos de comentarios en C ++:

Comentarios de una sola línea

La secuencia de doble barra diagonal hacia adelante `//` marcará todo el texto hasta que aparezca una nueva línea como comentario:

```
int main()
{
   // This is a single-line comment.
   int a;  // this also is a single-line comment
   int i;  // this is another single-line comment
}
```

C-Style / Block Comentarios

La secuencia `/*` se usa para declarar el inicio del bloque de comentarios y la secuencia `*/` se usa para declarar el final del comentario. Todo el texto entre las secuencias de inicio y finalización se interpreta como un comentario, incluso si el texto es de otro modo una sintaxis de C ++ válida. Estos a veces se denominan comentarios de "estilo C", ya que esta sintaxis de comentario se hereda del lenguaje predecesor de C ++, C:

```
int main()
{
   /*
    *  This is a block comment.
    */
   int a;
}
```

En cualquier comentario de bloque, puedes escribir lo que quieras. Cuando el compilador encuentra el símbolo `*/` , termina el comentario de bloque:

```
int main()
{
   /* A block comment with the symbol /*
      Note that the compiler is not affected by the second /*
      however, once the end-block-comment symbol is reached,
      the comment ends.
   */
   int a;
}
```

El ejemplo anterior es un código válido de C ++ (y C). Sin embargo, tener `/*` adicional dentro de un comentario de bloque puede dar como resultado una advertencia en algunos compiladores.

Los comentarios en bloque también pueden comenzar y terminar *dentro de* una sola línea. Por ejemplo:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Importancia de los comentarios

Al igual que con todos los lenguajes de programación, los comentarios proporcionan varios beneficios:

- Documentación explícita de código para facilitar la lectura / mantenimiento
- Explicación del propósito y funcionalidad del código.
- Detalles sobre la historia o razonamiento detrás del código.
- Colocación de derechos de autor / licencias, notas de proyectos, agradecimientos especiales, créditos de contribuyentes, etc. directamente en el código fuente.

Sin embargo, los comentarios también tienen sus desventajas:

- Deben mantenerse para reflejar cualquier cambio en el código.
- Los comentarios excesivos tienden a hacer que el código sea *menos* legible

La necesidad de comentarios se puede reducir escribiendo un código claro y autodocumentado. Un ejemplo simple es el uso de nombres explicativos para variables, funciones y tipos. Factorizar tareas relacionadas lógicamente en funciones discretas va de la mano con esto.

Marcadores de comentario utilizados para deshabilitar el código

Durante el desarrollo, los comentarios también se pueden usar para deshabilitar rápidamente partes del código sin borrarlo. A menudo, esto es útil para propósitos de prueba o depuración, pero no es un buen estilo para nada que no sean ediciones temporales. Esto a menudo se conoce como "comentar".

Del mismo modo, mantener las versiones antiguas de un fragmento de código en un comentario con fines de referencia es desagradable, ya que desordena los archivos al tiempo que ofrece poco valor en comparación con la exploración del historial del código a través de un sistema de versiones.

Función

Una **función** es una unidad de código que representa una secuencia de sentencias.

Las funciones pueden aceptar **argumentos** o valores y **devolver** un solo valor (o no). Para usar una función, una **llamada de función** se usa en valores de argumento y el uso de la llamada de función se reemplaza con su valor de retorno.

Cada función tiene una **firma de tipo** : los tipos de sus argumentos y el tipo de su tipo de retorno.

Las funciones están inspiradas en los conceptos del procedimiento y la función matemática.

- Nota: las funciones de C ++ son esencialmente procedimientos y no siguen la definición exacta o las reglas de las funciones matemáticas.

Las funciones a menudo están destinadas a realizar una tarea específica. y puede ser llamado desde otras partes de un programa. Una función debe ser declarada y definida antes de ser llamada en otro lugar en un programa.

- Nota: las definiciones de funciones populares pueden estar ocultas en otros archivos incluidos (a menudo por conveniencia y reutilización en muchos archivos). Este es un uso común de los archivos de encabezado.

Declaración de funciones

Una **declaración de función** declara la existencia de una función con su nombre y tipo de firma para el compilador. La sintaxis es la siguiente:

```
int add2(int i); // The function is of the type (int) -> (int)
```

En el ejemplo anterior, la función `int add2(int i)` declara lo siguiente al compilador:

- El **tipo de retorno** es `int` .
- El **nombre** de la función es `add2` .
- El **número de argumentos** a la función es 1:
  - El primer argumento es del tipo `int` .
  - El primer argumento se mencionará en el contenido de la función con el nombre `i` .

El nombre del argumento es opcional; La declaración para la función también podría ser la siguiente:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

Según la **regla de** una **definición** , una función con un cierto tipo de firma solo se puede declarar o definir una vez en una base de código C ++ completa visible para el compilador de C ++. En otras palabras, las funciones con una firma de tipo específica no se pueden redefinir, solo deben definirse una vez. Por lo tanto, lo siguiente no es válido en C ++:

```
int add2(int i);  // The compiler will note that add2 is a function (int) -> int
int add2(int j);  // As add2 already has a definition of (int) -> int, the compiler
                  // will regard this as an error.
```

Si una función no devuelve nada, su tipo de retorno se escribe como `void` . Si no toma parámetros, la lista de parámetros debe estar vacía.

```
void do_something(); // The function takes no parameters, and does not return anything.
                     // Note that it can still affect variables it has access to.
```

Llamada de función

Una función puede ser llamada después de que haya sido declarada. Por ejemplo, el siguiente programa llama a `add2` con el valor de `2` dentro de la función de `main` :

```
#include <iostream>

int add2(int i);    // Declaration of add2

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
```

```
// add2's definition may be LINKED in from another object file.

int main()
{
    std::cout << add2(2) << "\n";  // add2(2) will be evaluated at this point,
                                   // and the result is printed.
    return 0;
}
```

Aquí, `add2(2)` es la sintaxis de una llamada de función.

Definición de la función

Una *definición de función* * es similar a una declaración, excepto que también contiene el código que se ejecuta cuando se llama a la función dentro de su cuerpo.

Un ejemplo de una definición de función para `add2` podría ser:

```
int add2(int i)       // Data that is passed into (int i) will be referred to by the name i
{                     // while in the function's curly brackets or "scope."

    int j = i + 2;    // Definition of a variable j as the value of i+2.
    return j;         // Returning or, in essence, substitution of j for a function call to
                      // add2.
}
```

Sobrecarga de funciones

Puedes crear múltiples funciones con el mismo nombre pero diferentes parámetros.

```
int add2(int i)           // Code contained in this definition will be evaluated
{                         // when add2() is called with one parameter.
    int j = i + 2;
    return j;
}

int add2(int i, int j)    // However, when add2() is called with two parameters, the
{                         // code from the initial declaration will be overloaded,
    int k = i + j + 2 ;   // and the code in this declaration will be evaluated
    return k;             // instead.
}
```

Ambas funciones se llaman con el mismo nombre `add2` , pero la función real que se llama depende directamente de la cantidad y el tipo de los parámetros en la llamada. En la mayoría de los casos, el compilador de C ++ puede calcular qué función llamar. En algunos casos, el tipo debe ser explícitamente establecido.

Parámetros predeterminados

Los valores predeterminados para los parámetros de función solo se pueden especificar en las declaraciones de función.

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;               // If multiply() is called with one parameter, the
}                               // value will be multiplied by the default, 7.
```

En este ejemplo, se puede llamar a `multiply()` con uno o dos parámetros. Si solo se proporciona un parámetro, `b` tendrá un valor predeterminado de 7. Los argumentos predeterminados se deben colocar en los últimos argumentos de la función. Por ejemplo:

```
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former
```

Llamadas de Funciones Especiales - Operadores

Existen llamadas a funciones especiales en C ++ que tienen una sintaxis diferente a `name_of_function(value1, value2, value3)` . El ejemplo más común es el de los operadores.

Ciertas secuencias de caracteres especiales que se reducirán a las llamadas de función del compilador, como `!` , `+` , `-` , `*` , `%` y `<<` y muchos más. Estos caracteres especiales se asocian normalmente con el uso no programado o se usan para la estética (por ejemplo, el carácter `+` se reconoce comúnmente como el símbolo de adición tanto en la programación en C ++ como en matemáticas elementales).

C ++ maneja estas secuencias de caracteres con una sintaxis especial; pero, en esencia, cada aparición de un operador se reduce a una llamada de función. Por ejemplo, la siguiente expresión en C ++:

```
3+3
```

es equivalente a la siguiente llamada de función:

```
operator+(3, 3)
```

Todos los nombres de funciones del operador comienzan con el `operator` .

Mientras que en el predecesor inmediato de C ++, C, a los nombres de funciones de operador no se les pueden asignar diferentes significados al proporcionar definiciones adicionales con diferentes tipos de firmas, en C ++, esto es válido. "Ocultar" las definiciones de funciones adicionales bajo un nombre de función único se conoce como **sobrecarga de operadores** en C ++, y es una convención relativamente común, pero no universal, en C ++.

Visibilidad de prototipos y declaraciones de funciones.

En C ++, el código debe ser declarado o definido antes de su uso. Por ejemplo, lo siguiente produce un error de tiempo de compilación:

```
int main()
{
  foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{
}
```

Hay dos formas de resolver esto: poner la definición o la declaración de `foo()` antes de su uso en `main()` . Aquí hay un ejemplo:

```
void foo(int x) {}  //Declare the foo function and body first

int main()
{
  foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

Sin embargo, también es posible "declarar hacia adelante" la función poniendo solo una declaración "prototipo" antes de su uso y luego definiendo el cuerpo de la función más adelante:

```
void foo(int);  // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main()
{
  foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

El prototipo debe especificar el tipo de retorno (`void`), el nombre de la función (`foo`) y los tipos de variables de la lista de argumentos (`int`), pero los nombres de los argumentos NO son necesarios .

Una forma común de integrar esto en la organización de los archivos de origen es hacer un archivo de encabezado que contenga todas las declaraciones de prototipo:

```
// foo.h
void foo(int); // prototype declaration
```

y luego proporcionar la definición completa en otro lugar:

```
// foo.cpp --> foo.o
```

```
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
```

y luego, una vez compilado, vincule el archivo de objeto correspondiente `foo.o` al archivo de objeto compilado donde se usa en la fase de enlace, `main.o` :

```
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
int main() { foo(2); } // foo is valid to call because its prototype declaration was
beforehand.
// the prototype and body definitions of foo are linked through the object files
```

Se produce un error de "símbolo externo no resuelto" cuando existen la función *prototipo* y la *llamada* , pero el *cuerpo de* la función no está definido. Estos pueden ser más difíciles de resolver ya que el compilador no informará el error hasta la etapa final de vinculación, y no sabe a qué línea saltar en el código para mostrar el error.

El proceso de compilación estándar de C ++.

El código del programa ejecutable de C ++ generalmente es producido por un compilador.

Un **compilador** es un programa que traduce código de un lenguaje de programación a otra forma que es (más) directamente ejecutable para una computadora. Usar un compilador para traducir código se llama **compilación.**

C ++ hereda la forma de su proceso de compilación de su lenguaje "principal", C. A continuación se muestra una lista que muestra los cuatro pasos principales de la compilación en C ++:

1. El preprocesador de C ++ copia el contenido de cualquier archivo de encabezado incluido en el archivo de código fuente, genera un código de macro y reemplaza las constantes simbólicas definidas usando #define con sus valores.
2. El archivo de código fuente expandido producido por el preprocesador C ++ se compila en lenguaje ensamblador apropiado para la plataforma.
3. El código del ensamblador generado por el compilador se ensambla en el código de objeto apropiado para la plataforma.
4. El archivo de código de objeto generado por el ensamblador está vinculado junto con los archivos de código de objeto para cualquier función de biblioteca utilizada para producir un archivo ejecutable.

- Nota: algunos códigos compilados están vinculados entre sí, pero no para crear un programa final. Por lo general, este código "vinculado" también se puede empaquetar en un formato que puede ser usado por otros programas. Este "paquete de código empaquetado y utilizable" es lo que los programadores de C ++ denominan una **biblioteca.**

Muchos compiladores de C ++ también pueden combinar o deshacer ciertas partes del proceso de compilación para facilitar o para un análisis adicional. Muchos programadores de C ++ usarán diferentes herramientas, pero todas las herramientas generalmente seguirán este proceso generalizado cuando estén involucrados en la producción de un programa.

El siguiente enlace extiende esta discusión y proporciona un bonito gráfico para ayudar. [1]: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Preprocesador

El preprocesador es una parte importante del compilador.

Edita el código fuente, recorta algunos bits, cambia otros y agrega otras cosas.

En los archivos de origen, podemos incluir directivas de preprocesador. Estas directivas le indican al preprocesador que realice acciones específicas. Una directiva comienza con un # en una nueva línea. Ejemplo:

```
#define ZERO 0
```

La primera directiva de preprocesador que encontrará es probablemente la

```
#include <something>
```

directiva. Lo que hace es toma todos `something` y lo inserta en el archivo de donde estaba la directiva. El programa hola mundo comienza con la línea.

```
#include <iostream>
```

Esta línea agrega las funciones y objetos que le permiten usar la entrada y salida estándar.

El lenguaje C, que también utiliza el preprocesador, no tiene tantos archivos de encabezado como el lenguaje C ++, pero en C ++ puede usar todos los archivos de encabezado C.

La siguiente directiva importante es probablemente la

```
#define something something_else
```

directiva. Esto le dice al preprocesador que a medida que avanza en el archivo, debe reemplazar cada ocurrencia de `something` con `something_else` . También puede hacer cosas similares a las funciones, pero eso probablemente cuenta como C ++ avanzado.

El `something_else` no es necesario, pero si se define `something` como nada, entonces fuera de las directivas de preprocesador, todas las apariciones de `something` se desvanecerá.

En realidad, esto es útil, debido a las `#if` , `#else` y `#ifdef` directivas. El formato para estos sería el siguiente:

```
#if something==true
//code
#else
//more code
#endif
```

```
#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

Estas directivas insertan el código que está en el bit verdadero y borran los bits falsos. Esto se puede usar para tener bits de código que solo se incluyen en ciertos sistemas operativos, sin tener que volver a escribir todo el código.

Lea Empezando con C ++ en línea: https://riptutorial.com/es/cplusplus/topic/206/empezando-con-c-plusplus

Capítulo 2: Administracion de recursos

Introducción

Una de las cosas más difíciles de hacer en C y C ++ es la administración de recursos. Afortunadamente, en C ++, tenemos muchas maneras de diseñar el manejo de recursos en nuestros programas. Este artículo espera explicar algunos de los modismos y métodos utilizados para administrar los recursos asignados.

Examples

Adquisición de recursos es la inicialización

La adquisición de recursos es la inicialización (RAII) es un lenguaje común en la gestión de recursos. En el caso de la memoria dinámica, utiliza punteros inteligentes para llevar a cabo la gestión de recursos. Cuando se usa RAII, un recurso adquirido se otorga de inmediato a un puntero inteligente o administrador de recursos equivalente. Solo se puede acceder al recurso a través de este administrador, por lo que el administrador puede realizar un seguimiento de varias operaciones. Por ejemplo, `std::auto_ptr` libera automáticamente su recurso correspondiente cuando queda fuera del alcance o se elimina de otro modo.

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    {
        auto_ptr ap(new int(5)); // dynamic memory is the resource
        cout << *ap << endl; // prints 5
    } // auto_ptr is destroyed, its resource is automatically freed
}
```

C ++ 11

El principal problema de `std::auto_ptr` es que no se puede copiar sin transferir la propiedad:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // prints 5
    auto_ptr ap2(ap1); // copy ap2 from ap1; ownership now transfers to ap2
    cout << *ap2 << endl; // prints 5
    cout << ap1 == nullptr << endl; // prints 1; ap1 has lost ownership of resource
}
```

Debido a estas extrañas semánticas de copia, `std::auto_ptr` no se puede usar en contenedores,

entre otras cosas. La razón por la que hace esto es para evitar que se borre la memoria dos veces: si hay dos `auto_ptrs` con propiedad del mismo recurso, ambos intentan liberarla cuando se destruyen. La liberación de un recurso ya liberado generalmente puede causar problemas, por lo que es importante prevenirlo. Sin embargo, `std::shared_ptr` tiene un método para evitar esto y no transfiere la propiedad al copiar:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    shared_ptr sp2;
    {
        shared_ptr sp1(new int(5)); // give ownership to sp1
        cout << *sp1 << endl; // prints 5
        sp2 = sp1; // copy sp2 from sp1; both have ownership of resource
        cout << *sp1 << endl; // prints 5
        cout << *sp2 << endl; // prints 5
    } // sp1 goes out of scope and is destroyed; sp2 has sole ownership of resource
    cout << *sp2 << endl;
} // sp2 goes out of scope; nothing has ownership, so resource is freed
```

Mutexes y seguridad de rosca

Pueden surgir problemas cuando varios subprocesos intentan acceder a un recurso. Para un ejemplo simple, supongamos que tenemos un hilo que agrega uno a una variable. Para ello, primero lee la variable, le agrega una y luego la almacena de nuevo. Supongamos que inicializamos esta variable a 1, luego creamos dos instancias de este hilo. Una vez que ambos subprocesos terminan, la intuición sugiere que esta variable debería tener un valor de 3. Sin embargo, la siguiente tabla ilustra lo que podría salir mal:

	Hilo 1	Hilo 2
Tiempo Paso 1	Lee 1 de la variable	
Tiempo Paso 2		Lee 1 de la variable
Tiempo Paso 3	Agrega 1 más 1 para obtener 2	
Tiempo Paso 4		Agrega 1 más 1 para obtener 2
Tiempo paso 5	Almacenar 2 en variable	
Tiempo Paso 6		Almacenar 2 en variable

Como puede ver, al final de la operación, 2 está en la variable, en lugar de 3. La razón es que el Subproceso 2 leyó la variable antes de que el Subproceso 1 terminara de actualizarla. ¿La solución? Mutexes.

Un mutex (acrónimo de **mut** UAL **ex** conclusión) es un objeto de la gestión de recursos diseñado para resolver este tipo de problema. Cuando un hilo quiere acceder a un recurso, "adquiere" la

exclusión mutua del recurso. Una vez que se termina de acceder al recurso, el hilo "libera" el mutex. Mientras se adquiere el mutex, todas las llamadas para adquirir el mutex no volverán hasta que se libere el mutex. Para entender mejor esto, piense en un mutex como una línea de espera en el supermercado: los hilos se alinean al tratar de adquirir el mutex y luego esperar a que los hilos por delante terminen, luego usar el recurso, luego salir del paso. línea liberando el mutex. Habría pandemonium completo si todos trataran de acceder al recurso a la vez.

C ++ 11

`std::mutex` es la implementación de C ++ 11 de un mutex.

```
#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // function to be run in thread
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // prints 1

    thread t1(add_1, var, m); // create thread with arguments
    thread t2(add_1, var, m); // create another thread
    t1.join(); t2.join(); // wait for both threads to finish

    cout << var << endl; // prints 3
}
```

Lea Administracion de recursos en línea:
https://riptutorial.com/es/cplusplus/topic/8336/administracion-de-recursos

Capítulo 3: Alcances

Examples

Alcance de bloque simple

El alcance de una variable en un bloque `{ ... }`, comienza después de la declaración y termina al final del bloque. Si hay un bloque anidado, el bloque interno puede ocultar el alcance de una variable que se declara en el bloque externo.

```
{
    int x = 100;
    //   ^
    //   Scope of `x` begins here
    //
}   // <- Scope of `x` ends here
```

Si un bloque anidado comienza dentro de un bloque externo, una nueva variable declarada con el mismo nombre que está antes en la clase externa, oculta el primero.

```
{
    int x = 100;

    {
        int x = 200;

        std::cout << x;  // <- Output is 200
    }

    std::cout << x;  // <- Output is 100
}
```

Variables globales

Para declarar una instancia única de una variable a la que se puede acceder en diferentes archivos de origen, es posible hacerlo en el ámbito global con la palabra clave `extern`. Esta palabra clave le dice al compilador que en alguna parte del código hay una definición para esta variable, por lo que puede usarse en todas partes y toda la escritura / lectura se realizará en un lugar de la memoria.

```
// File my_globals.h:

#ifndef __MY_GLOBALS_H__
#define __MY_GLOBALS_H__

extern int circle_radius; // Promise to the compiler that circle_radius
                          // will be defined somewhere

#endif
```

```
// File foo1.cpp:

#include "my_globals.h"

int circle_radius = 123; // Defining the extern variable
```

```
// File main.cpp:

#include "my_globals.h"
#include <iostream>

int main()
{
    std::cout << "The radius is: " << circle_radius << "\n";'
    return 0;
}
```

Salida:

```
The radius is: 123
```

Lea Alcances en línea: https://riptutorial.com/es/cplusplus/topic/3453/alcances

Capítulo 4: Algoritmos de la biblioteca estándar

Examples

std :: for_each

```
template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
```

Efectos:

Aplica `f` al resultado de la desreferenciación de todos los iteradores en el rango `[first, last)` partir del `first` y continúa hasta el `last - 1` .

Parámetros:

`first, last` - el rango para aplicar `f` a.

`f` - objeto llamable que se aplica al resultado de la anulación de la referencia de cada iterador en el rango `[first, last)` .

Valor de retorno:

`f` (hasta C ++ 11) y `std::move(f)` (desde C ++ 11).

Complejidad:

Se aplica `f` exactamente el `last - first` vez.

Ejemplo:

c ++ 11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

Aplica la función dada para cada elemento del vector `v` imprimiendo este elemento a la `stdout` .

std :: next_permutation

```
template< class Iterator >
bool next_permutation( Iterator first, Iterator last );
template< class Iterator, class Compare >
bool next_permutation( Iterator first, Iterator last, Compare cmpFun );
```

Efectos:

Tamice la secuencia de datos del rango [primero, último) en la siguiente permutación lexicográficamente más alta. Si se proporciona `cmpFun` , la regla de permutación se personaliza.

Parámetros:
`first` - el comienzo del rango a permutar, inclusive
`last` - el final del rango a ser permutado, exclusivo

Valor de retorno:
Devuelve true si existe tal permutación.
De lo contrario, el rango se cambia a la permutación lexicográficamente más pequeña y devuelve false.

Complejidad:
O (n), n es la distancia del `first` al `last` .

Ejemplo :

```
std::vector< int > v { 1, 2, 3 };
do
{
   for( int i = 0; i < v.size(); i += 1 )
   {
       std::cout << v[i];
   }
   std::cout << std::endl;
}while( std::next_permutation( v.begin(), v.end() ) );
```

Imprima todos los casos de permutación de 1,2,3 en orden lexicográficamente creciente.
salida:

```
123
132
213
231
312
321
```

std :: acumular

Definido en el encabezado `<numeric>`

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init); // (1)

template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation f); // (2)
```

Efectos:

std :: Se acumula realiza la operación de plegado usando la función `f` en el rango `[first, last)` comenzando con `init` como valor acumulador.

Efectivamente es equivalente a:

```
T acc = init;
for (auto it = first; first != last; ++it)
    acc = f(acc, *it);
return acc;
```

En la versión (1) el `operator+` se usa en lugar de `f` , por lo que acumular sobre el contenedor es equivalente a la suma de los elementos del contenedor.

Parámetros:

`first, last` - el rango para aplicar `f` a.
`init` - valor inicial del acumulador.
`f` - función de plegado binario.

Valor de retorno:

Valor acumulado de `f` aplicaciones.

Complejidad:

O (n × k) , donde *n* es la distancia de la `first` a la `last` , *O (k)* es la complejidad de la función `f` .

Ejemplo:

Ejemplo de suma simple:

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

Salida:

```
10
```

Convertir dígitos a número:

c ++ 11

```
class Converter {
public:
    int operator()(int a, int d) const { return a * 10 + d; }
};
```

y después

```
const int ds[3] = {1, 2, 3};
int n = std::accumulate(ds, ds + 3, 0, Converter());
std::cout << n << std::endl;
```

c ++ 11

```
const std::vector<int> ds = {1, 2, 3};
int n = std::accumulate(ds.begin(), ds.end(),
                        0,
                        [](int a, int d) { return a * 10 + d; });
std::cout << n << std::endl;
```

Salida:

```
123
```

std :: encontrar

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

Efectos

Encuentra la primera aparición de val dentro del rango [primero, último)

Parámetros

`first` => iterador que apunta al comienzo del rango `last` => iterador que apunta al final del rango `val` => el valor a encontrar dentro del rango

Regreso

Un iterador que apunta al primer elemento dentro del rango que es igual (==) a val, el iterador apunta a durar si no se encuentra val.

Ejemplo

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> intVec {4, 6, 8, 9, 10, 30, 55,100, 45, 2, 4, 7, 9, 43, 48};

  //define iterators
  vector<int>::iterator  itr_9;
  vector<int>::iterator  itr_43;
  vector<int>::iterator  itr_50;

  //calling find
  itr_9 = find(intVec.begin(), intVec.end(), 9); //occurs twice
  itr_43 = find(intVec.begin(), intVec.end(), 43); //occurs once

  //a value not in the vector
  itr_50 = find(intVec.begin(), intVec.end(), 50); //does not occur
```

```
  cout << "first occurence of: " << *itr_9 << endl;
  cout << "only occurence of: " << *itr_43 << Lendl;


  /*
    let's prove that itr_9 is pointing to the first occurence
    of 9 by looking at the element after 9, which should be 10
    not 43
  */
  cout << "element after first 9: " << *(itr_9 + 1) << ends;

  /*
    to avoid dereferencing intVec.end(), lets look at the
    element right before the end
  */
  cout << "last element: " << *(itr_50 - 1) << endl;

  return 0;
}
```

Salida

```
first occurence of: 9
only occurence of: 43
element after first 9: 10
last element: 48
```

std :: cuenta

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

Efectos

Cuenta el número de elementos que son iguales a val.

Parámetros

`first` => iterador que apunta al comienzo del rango
`last` => iterador que apunta al final del rango
`val` => La ocurrencia de este valor en el rango será contada

Regreso

El número de elementos en el rango que son iguales (==) a val.

Ejemplo

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;
```

```
int main(int argc, const char * argv[]) {

  //create vector
  vector<int> intVec{4,6,8,9,10,30,55,100,45,2,4,7,9,43,48};

  //count occurences of 9, 55, and 101
  size_t count_9 = count(intVec.begin(), intVec.end(), 9); //occurs twice
  size_t count_55 = count(intVec.begin(), intVec.end(), 55); //occurs once
  size_t count_101 = count(intVec.begin(), intVec.end(), 101); //occurs once

  //print result
  cout << "There are " << count_9  << " 9s"<< endl;
  cout << "There is " << count_55  << " 55"<< endl;
  cout << "There is " << count_101  << " 101"<< ends;

  //find the first element == 4 in the vector
  vector<int>::iterator itr_4 = find(intVec.begin(), intVec.end(), 4);

  //count its occurences in the vector starting from the first one
  size_t count_4 = count(itr_4, intVec.end(), *itr_4); // should be 2

  cout << "There are " << count_4  << " " << *itr_4 << endl;

  return 0;
}
```

Salida

```
There are 2 9s
There is 1 55
There is 0 101
There are 2 4
```

std :: count_if

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate red);
```

Efectos

Cuenta el número de elementos en un rango para el que una función de predicado especificada es verdadera

Parámetros

`first` => iterador que apunta al principio del rango `last` => iterador que apunta al final del rango `red` => función de predicado (devuelve verdadero o falso)

Regreso

El número de elementos dentro del rango especificado para el cual la función de predicado devolvió verdadero.

Ejemplo

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/*
    Define a few functions to use as predicates
*/

//return true if number is odd
bool isOdd(int i){
  return i%2 == 1;
}

//functor that returns true if number is greater than the value of the constructor parameter
provided
class Greater {
  int _than;
public:
  Greater(int th): _than(th){}
  bool operator()(int i){
    return i > _than;
  }
};

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> myvec = {1,5,8,0,7,6,4,5,2,1,5,0,6,9,7};

  //using a lambda function to count even numbers
  size_t evenCount = count_if(myvec.begin(), myvec.end(), [](int i){return i % 2 == 0;}); //
>= C++11

  //using function pointer to count odd number in the first half of the vector
  size_t oddCount = count_if(myvec.begin(), myvec.end()- myvec.size()/2, isOdd);

  //using a functor to count numbers greater than 5
  size_t greaterCount = count_if(myvec.begin(), myvec.end(), Greater(5));

  cout << "vector size: " << myvec.size() << endl;
  cout << "even numbers: " << evenCount << " found" << endl;
  cout << "odd numbers: " << oddCount << " found" << endl;
  cout << "numbers > 5: " << greaterCount << " found"<< endl;

  return 0;
}
```

Salida

```
vector size: 15
even numbers: 7 found
odd numbers: 4 found
numbers > 5: 6 found
```

std :: find_if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

Efectos

Encuentra el primer elemento en un rango para el cual la función de predicado `pred` devuelve true.

Parámetros

`first` => iterador que apunta al principio del rango `last` => iterador que apunta al final del rango `pred` => función de predicado (devuelve verdadero o falso)

Regreso

Un iterador que apunta al primer elemento dentro del rango para el que predice la función predate devuelve true para. El iterador apunta a durar si no se encuentra val

Ejemplo

```
#include <iostream>
#include <vector>
#include <algorithm>


using namespace std;

/*
    define some functions to use as predicates
*/

//Returns true if x is multiple of 10
bool multOf10(int x) {
  return x % 10 == 0;
}

//returns true if item greater than passed in parameter
class Greater {
  int _than;

public:
  Greater(int th):_than(th){

  }
  bool operator()(int data) const
  {
    return data > _than;
  }
};


int main()
{

  vector<int> myvec {2, 5, 6, 10, 56, 7, 48, 89, 850, 7, 456};
```

```
  //with a lambda function
  vector<int>::iterator gt10 = find_if(myvec.begin(), myvec.end(), [](int x){return x>10;});
// >= C++11

  //with a function pointer
  vector<int>::iterator pow10 = find_if(myvec.begin(), myvec.end(), multOf10);

  //with functor
  vector<int>::iterator gt5 = find_if(myvec.begin(), myvec.end(), Greater(5));

  //not Found
  vector<int>::iterator nf = find_if(myvec.begin(), myvec.end(), Greater(1000)); // nf points
to myvec.end()


  //check if pointer points to myvec.end()
  if(nf != myvec.end()) {
    cout << "nf points to: " << *nf << endl;
  }
  else {
    cout << "item not found" << endl;
  }



  cout << "First item >   10: " << *gt10  << endl;
  cout << "First Item n * 10: " << *pow10 << endl;
  cout << "First Item >    5: " << *gt5   << endl;

  return 0;
}
```

Salida

```
item not found
First item >   10: 56
First Item n * 10: 10
First Item >    5: 6
```

std :: min_element

```
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last,Compare comp);
```

Efectos

Encuentra el elemento mínimo en un rango.

Parámetros

`first` - iterador que apunta al comienzo del rango
`last` : iterador que apunta al final del rango `comp` : un puntero de función u objeto de función que

toma dos argumentos y devuelve verdadero o falso, lo que indica si el argumento es menor que el argumento 2. Esta función no debe modificar las entradas

Regreso

Iterador al elemento mínimo en el rango.

Complejidad

Lineal en uno menos que el número de elementos comparados.

Ejemplo

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>  //to use make_pair

using namespace std;

//function compare two pairs
bool pairLessThanFunction(const pair<string, int> &p1, const pair<string, int> &p2)
{
  return p1.second < p2.second;
}

int main(int argc, const char * argv[]) {

  vector<int> intVec {30,200,167,56,75,94,10,73,52,6,39,43};

  vector<pair<string, int>> pairVector = {make_pair("y", 25), make_pair("b", 2),
make_pair("z", 26), make_pair("e", 5) };


  // default using < operator
  auto minInt = min_element(intVec.begin(), intVec.end());

  //Using pairLessThanFunction
  auto minPairFunction = min_element(pairVector.begin(), pairVector.end(),
pairLessThanFunction);


  //print minimum of intVector
  cout << "min int from default: " << *minInt << endl;

  //print minimum of pairVector
  cout << "min pair from PairLessThanFunction: " << (*minPairFunction).second << endl;

  return 0;
}
```

Salida

```
min int from default: 6
min pair from PairLessThanFunction: 2
```

Usando std :: nth_element para encontrar la mediana (u otros cuantiles)

El algoritmo `std::nth_element` toma tres iteradores: un iterador al principio, a la posición *n* y al final. Una vez que la función devuelve, el *n-ésimo* elemento (por orden) será el *n-ésimo* elemento más pequeño. (La función tiene sobrecargas más elaboradas, p. Ej., Algunos tienen funciones de comparación; consulte el enlace anterior para ver todas las variaciones).

Nota Esta función es muy eficiente, tiene una complejidad lineal.

Por el bien de este ejemplo, definamos la mediana de una secuencia de longitud *n* como el elemento que estaría en la posición ⌈n / 2⌉. Por ejemplo, la mediana de una secuencia de longitud 5 es el tercer elemento más pequeño, y también lo es la mediana de una secuencia de longitud 6.

Para usar esta función para encontrar la mediana, podemos usar lo siguiente. Digamos que empezamos con

```
std::vector<int> v{5, 1, 2, 3, 4};

std::vector<int>::iterator b = v.begin();
std::vector<int>::iterator e = v.end();

std::vector<int>::iterator med = b;
std::advance(med, v.size() / 2);

// This makes the 2nd position hold the median.
std::nth_element(b, med, e);

// The median is now at v[2].
```

Para encontrar el *p* th cuantil , cambiaríamos algunas de las líneas de arriba:

```
const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);
```

y buscar el cuantil en `pos` .

Lea Algoritmos de la biblioteca estándar en línea:
https://riptutorial.com/es/cplusplus/topic/3177/algoritmos-de-la-biblioteca-estandar

Capítulo 5: Alineación

Introducción

Todos los tipos en C ++ tienen una alineación. Esta es una restricción en la dirección de memoria dentro de la cual se pueden crear objetos de ese tipo. Una dirección de memoria es válida para la creación de un objeto si la división de esa dirección por la alineación del objeto es un número entero.

Las alineaciones de tipo son siempre una potencia de dos (incluido 1).

Observaciones

La norma garantiza lo siguiente:

- El requisito de alineación de un tipo es un divisor de su tamaño. Por ejemplo, una clase con un tamaño de 16 bytes podría tener una alineación de 1, 2, 4, 8 o 16, pero no 32. (Si los miembros de una clase solo tienen un tamaño de 14 bytes, pero la clase debe tener un requisito de alineación de 8, el compilador insertará 2 bytes de relleno para hacer que el tamaño de la clase sea igual a 16.)

- Las versiones firmadas y sin firmar de un tipo entero tienen el mismo requisito de alineación.
- Un puntero para `void` tiene el mismo requisito de alineación que un puntero para `char` .
- Las versiones calificadas para cv y no calificadas para cv de un tipo tienen el mismo requisito de alineación.

Tenga en cuenta que si bien existe alineación en C ++ 03, no fue hasta C ++ 11 que se hizo posible consultar la alineación (usando `alignof`) y la alineación de control (utilizando `alignas`).

Examples

Consultar la alineación de un tipo.

c ++ 11

El requisito de alineación de un tipo se puede consultar utilizando la palabra clave `alignof` como un operador `alignof` . El resultado es una expresión constante de tipo `std::size_t` , *es decir,* se puede evaluar en tiempo de compilación.

```
#include <iostream>
int main() {
    std::cout << "The alignment requirement of int is: " << alignof(int) << '\n';
}
```

Salida posible

El requisito de alineación de int es: 4

Si se aplica a una matriz, produce el requisito de alineación del tipo de elemento. Si se aplica a un tipo de referencia, produce el requisito de alineación del tipo referenciado. (Las referencias en sí mismas no tienen alineación, ya que no son objetos).

Controlando la alineación

C ++ 11

La palabra clave `alignas` se puede usar para forzar a una variable, miembro de datos de clase, declaración o definición de una clase, o declaración o definición de una enumeración, para tener una alineación particular, si se admite. Se presenta en dos formas:

- `alignas(x)` , donde `x` es una expresión constante, le da a la entidad la alineación `x` , si es compatible.
- `alignas(T)` , donde `T` es un tipo, le da a la entidad una alineación igual al requisito de alineación de `T` , es decir, `alignof(T)` , si es compatible.

Si se aplican múltiples especificadores `alignas` a la misma entidad, se aplica el más estricto.

En este ejemplo, se garantiza que el búfer `buf` se alineará adecuadamente para contener un objeto `int` , aunque su tipo de elemento sea un `unsigned char` , que puede tener un requisito de alineación más débil.

```
alignas(int) unsigned char buf[sizeof(int)];
new (buf) int(42);
```

`alignas` no se puede usar para dar a un tipo una alineación más pequeña que la que tendría el tipo sin esta declaración:

```
alignas(1) int i; //Il-formed, unless `int` on this platform is aligned to 1 byte.
alignas(char) int j; //Il-formed, unless `int` has the same or smaller alignment than `char`.
```

`alignas` , cuando se le da una expresión constante entera, se le debe dar una alineación válida. Las alineaciones válidas son siempre potencias de dos y deben ser mayores que cero. Los compiladores deben admitir todas las alineaciones válidas hasta la alineación del tipo `std::max_align_t` . *Pueden* apoyar las alineaciones más grandes que esto, pero el soporte para la asignación de memoria para tales objetos es limitada. El límite superior de las alineaciones depende de la implementación.

C ++ 17 cuenta con soporte directo en `operator new` para asignar memoria para tipos sobre alineados.

Lea Alineación en línea: https://riptutorial.com/es/cplusplus/topic/9249/alineacion

Capítulo 6: Archivo I / O

Introducción

El archivo de C ++ I / O se realiza a través de *secuencias* . Las abstracciones clave son:

`std::istream` para leer texto.

`std::ostream` para escribir texto.

`std::streambuf` para leer o escribir personajes.

La entrada formateada utiliza el `operator>>` .

La salida formateada utiliza el `operator<<` .

Las secuencias utilizan `std::locale` , por ejemplo, para detalles del formato y para la traducción entre las codificaciones externas y la codificación interna.

Más sobre streams: <iostream> Library

Examples

Abriendo un archivo

La apertura de un archivo se realiza de la misma manera para las 3 secuencias de archivos (`ifstream` , `ofstream` y `fstream`).

Puedes abrir el archivo directamente en el constructor:

```
std::ifstream ifs("foo.txt");  // ifstream: Opens file "foo.txt" for reading only.

std::ofstream ofs("foo.txt");  // ofstream: Opens file "foo.txt" for writing only.

std::fstream iofs("foo.txt");  // fstream:  Opens file "foo.txt" for reading and writing.
```

Alternativamente, puede usar la función de miembro `open()` stream `open()` :

```
std::ifstream ifs;
ifs.open("bar.txt");           // ifstream: Opens file "bar.txt" for reading only.

std::ofstream ofs;
ofs.open("bar.txt");           // ofstream: Opens file "bar.txt" for writing only.

std::fstream iofs;
iofs.open("bar.txt");          // fstream:  Opens file "bar.txt" for reading and writing.
```

Usted debe comprobar **siempre** si un archivo ha sido abierto con éxito (incluso cuando se escribe). Las fallas pueden incluir: el archivo no existe, el archivo no tiene los derechos de acceso

correctos, el archivo ya está en uso, se produjeron errores en el disco, la unidad se desconectó ... La verificación se puede hacer de la siguiente manera:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("fooo.txt");  // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

Cuando la ruta del archivo contenga barras diagonales inversas (por ejemplo, en el sistema Windows), debe eliminarlas correctamente:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:\\folder\\foo.txt"); // using escaped backslashes
```

C ++ 11

o usar el literal en bruto:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs(R"(c:\folder\foo.txt)"); // using raw literal
```

o use barras diagonales en su lugar:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

C ++ 11

Si desea abrir un archivo con caracteres no ASCII en la ruta en Windows actualmente, puede usar el argumento de ruta de caracteres anchos **no estándar** :

```
// Open the file 'пример\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\foo.txt)"); // using wide characters with raw literal
```

Leyendo de un archivo

Hay varias formas de leer datos de un archivo.

Si sabe cómo se formatean los datos, puede usar el operador de extracción de flujo (`>>`). Supongamos que tiene un archivo llamado *foo.txt* que contiene los siguientes datos:

```
John Doe 25 4 6 1987
Jane Doe 15 5 24 1976
```

Luego puede usar el siguiente código para leer los datos del archivo:

```
// Define variables.
std::ifstream is("foo.txt");
std::string firstname, lastname;
int age, bmonth, bday, byear;

// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't
// correspond to the type of the input variable (for example, the string "foo" can't be
// extracted into an 'int' variable).
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)
    // Process the data that has been read.
```

El operador de extracción de flujo `>>` extrae cada carácter y se detiene si encuentra un carácter que no se puede almacenar o si es un carácter especial:

- Para los tipos de cadena, el operador se detiene en un espacio en blanco ( ) o en una nueva línea (`\n`).
- Para los números, el operador se detiene en un carácter no numérico.

Esto significa que la siguiente versión del archivo *foo.txt* también se leerá con éxito en el código anterior:

```
John
Doe 25
4 6 1987


Jane
Doe
15 5
24
1976
```

El operador de extracción de flujo `>>` siempre devuelve el flujo dado a él. Por lo tanto, se pueden encadenar varios operadores para leer datos consecutivamente. Sin embargo, una corriente también se puede utilizar como una expresión booleana (como se muestra en el `while` bucle en el código anterior). Esto se debe a que las clases de flujo tienen un operador de conversión para el tipo `bool` . Este operador `bool()` devolverá `true` siempre que la secuencia no tenga errores. Si un flujo entra en un estado de error (por ejemplo, porque no se pueden extraer más datos), el operador `bool()` devolverá el valor `false` . Por lo tanto, el `while` de bucle en el código anterior se saldrá después del archivo de entrada se ha leído hasta el final.

Si desea leer un archivo completo como una cadena, puede usar el siguiente código:

```
// Opens 'foo.txt'.
std::ifstream is("foo.txt");
std::string whole_file;

// Sets position to the end of the file.
is.seekg(0, std::ios::end);

// Reserves memory for the file.
whole_file.reserve(is.tellg());
```

```
// Sets position to the start of the file.
is.seekg(0, std::ios::beg);

// Sets contents of 'whole_file' to all characters in the file.
whole_file.assign(std::istreambuf_iterator<char>(is),
  std::istreambuf_iterator<char>());
```

Este código reserva espacio para la `string` con el fin de reducir las asignaciones de memoria innecesarias.

Si desea leer un archivo línea por línea, puede usar la función `getline()` :

```
std::ifstream is("foo.txt");

// The function getline returns false if there are no more lines.
for (std::string str; std::getline(is, str);) {
    // Process the line that has been read.
}
```

Si desea leer un número fijo de caracteres, puede usar la función miembro de la secuencia `read()` :

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

Después de ejecutar un comando de lectura, siempre debe verificar si el indicador de estado de error `failbit` se ha establecido, ya que indica si la operación falló o no. Esto se puede hacer llamando a la función miembro de la secuencia de archivos `fail()` :

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

Escribiendo en un archivo

Hay varias formas de escribir en un archivo. La forma más sencilla es utilizar una secuencia de archivo de salida (`ofstream`) junto con el operador de inserción de secuencia (`<<`):

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

En lugar de `<<` , también puede usar la función miembro `write()` la secuencia del archivo de salida:

```
std::ofstream os("foo.txt");
```

```
if(os.is_open()){
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

Después de escribir en un flujo, siempre debe verificar si se ha establecido el indicador de estado de error `badbit` , ya que indica si la operación falló o no. Esto se puede hacer llamando a la función miembro del flujo de salida del archivo `bad()` :

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Modos de apertura

Al crear una secuencia de archivos, puede especificar un modo de apertura. Un modo de apertura es básicamente una configuración para controlar cómo la secuencia abre el archivo.

(Todos los modos se pueden encontrar en el `std::ios` nombres `std::ios`).

Se puede proporcionar un modo de apertura como segundo parámetro para el constructor de un flujo de archivos o para su función miembro `open()` :

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

Se debe tener en cuenta que debe configurar `ios::in` o `ios::out` si desea establecer otros indicadores, ya que no están establecidos de forma implícita por los miembros de iostream aunque tengan un valor predeterminado correcto.

Si no especifica un modo de apertura, se utilizan los siguientes modos predeterminados:

- `ifstream` - `in`
- `ofstream` - `out`
- `fstream` - `in` y `out`

Los modos de apertura de archivos que puede especificar por diseño son:

Modo	Sentido	por	Descripción
`app`	adjuntar	Salida	Anexa datos al final del archivo.
`binary`	binario	De entrada y salida	La entrada y salida se realiza en binario.
`in`	entrada	Entrada	Abre el archivo para su lectura.
`out`	salida	Salida	Abre el archivo para escribir.

Modo	Sentido	por	Descripción
`trunc`	truncar	De entrada y salida	Elimina el contenido del archivo al abrirlo.
`ate`	al final	Entrada	Va al final del archivo al abrir.

Nota: la configuración del modo `binary` permite que los datos se lean / escriban exactamente como están; no configurarlo permite la traducción del carácter `'\n'` nueva línea `'\n'` a / desde una secuencia de final de línea específica de la plataforma.

Por ejemplo, en Windows, la secuencia de final de línea es CRLF (`"\r\n"`).
Escribe: `"\n"` => `"\r\n"`
Lee: `"\r\n"` => `"\n"`

Cerrando un archivo

Cerrar un archivo explícitamente rara vez es necesario en C ++, ya que una secuencia de archivos cerrará automáticamente su archivo asociado en su destructor. Sin embargo, debe intentar limitar la vida útil de un objeto de flujo de archivos, de modo que no mantenga abierto el manejador de archivos más de lo necesario. Por ejemplo, esto se puede hacer poniendo todas las operaciones de archivo en un ámbito propio (`{}`):

```
std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
}  // The ofstream will go out of scope here.
   // Its destructor will take care of closing the file properly.
```

Llamar a `close()` explícitamente solo es necesario si desea reutilizar el mismo objeto `fstream` más tarde, pero no desea mantener el archivo abierto entre:

```
// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
std::string const prepared_data = prepare_data();

// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();

// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");
```

```
// Write the data to the file "foo.txt".
output << more_prepared_data;

// Close the file "foo.txt" once again.
output.close();
```

Flushing un arroyo

Las secuencias de archivos se almacenan en búfer de forma predeterminada, al igual que muchos otros tipos de secuencias. Esto significa que las escrituras en la secuencia pueden no causar que el archivo subyacente cambie inmediatamente. A fin de forzar que todas las escrituras en búfer se realicen inmediatamente, puede *vaciar* la secuencia. Puede hacerlo directamente invocando el método `flush()` o mediante el manipulador de flujo `std::flush` :

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

Hay un manipulador de flujo `std::endl` que combina la escritura de una nueva línea con la descarga de flujo:

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

El almacenamiento en búfer puede mejorar el rendimiento de la escritura en una secuencia. Por lo tanto, las aplicaciones que escriben mucho deben evitar el lavado innecesario. Por el contrario, si la E / S se realiza con poca frecuencia, las aplicaciones deben considerar vaciar con frecuencia para evitar que los datos se atasquen en el objeto de flujo.

Leyendo un archivo ASCII en un std :: string

```
std::ifstream f("file.txt");

if (f)
{
  std::stringstream buffer;
  buffer << f.rdbuf();
  f.close();

  // The content of "file.txt" is available in the string `buffer.str()`
}
```

El método `rdbuf()` devuelve un puntero a un `streambuf` que puede `streambuf` en el `buffer` través de la función miembro `stringstream::operator<<` .

Otra posibilidad (popularizada en Effective STL por Scott Meyers) es:

```
std::ifstream f("file.txt");

if (f)
{
  std::string str((std::istreambuf_iterator<char>(f)),
                  std::istreambuf_iterator<char>());

  // Operations on `str`...
}
```

Esto es bueno porque requiere poco código (y permite leer un archivo directamente en cualquier contenedor STL, no solo cadenas) pero puede ser lento para archivos grandes.

NOTA : los paréntesis adicionales alrededor del primer argumento del constructor de cadenas son esenciales para evitar el problema de *análisis más desconcertante* .

Por último, si bien no menos importante:

```
std::ifstream f("file.txt");

if (f)
{
  f.seekg(0, std::ios::end);
  const auto size = f.tellg();

  std::string str(size, ' ');
  f.seekg(0);
  f.read(&str[0], size);
  f.close();

  // Operations on `str`...
}
```

que es probablemente la opción más rápida (entre las tres propuestas).

Leyendo un archivo en un contenedor

En el siguiente ejemplo, usamos `std::string` y `operator>>` para leer los elementos del archivo.

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v;

    std::string s;
    while(file >> s) // keep reading until we run out
    {
        v.push_back(s);
    }
```

En el ejemplo anterior, simplemente estamos iterando a través del archivo leyendo un "elemento" a la vez utilizando el `operator>>` . Este mismo efecto se puede lograr utilizando el `std::istream_iterator` que es un iterador de entrada que lee un "elemento" a la vez desde el flujo. Además, la mayoría de los contenedores se pueden construir utilizando dos iteradores, por lo que

podemos simplificar el código anterior para:

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v(std::istream_iterator<std::string>{file},
                                std::istream_iterator<std::string>{});
```

Podemos extender esto para leer cualquier tipo de objeto que nos guste simplemente especificando el objeto que queremos leer como el parámetro de plantilla para `std::istream_iterator` . Por lo tanto, podemos simplemente extender lo anterior para leer líneas (en lugar de palabras) como esta:

```
// Unfortunately there is  no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};


    std::ifstream file("file3.txt");

    // Read the lines of a file into a container.
    std::vector<std::string>  v(std::istream_iterator<Line>{file},
                                std::istream_iterator<Line>{});
```

Leyendo un `struct` desde un archivo de texto formateado.

C ++ 11

```
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
        return is;
    }
```

```
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info;) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << "  name: " << info.name << '\n';
        std::cout << "   age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}
```

file4.txt

```
Wogger Wabbit
2
6.2
Bilbo Baggins
111
81.3
Mary Poppins
29
154.8
```

Salida:

```
name: Wogger Wabbit
 age: 2 years
height: 6.2lbs

name: Bilbo Baggins
 age: 111 years
height: 81.3lbs

name: Mary Poppins
 age: 29 years
height: 154.8lbs
```

Copiando un archivo

```
std::ifstream  src("source_filename", std::ios::binary);
std::ofstream  dst("dest_filename",   std::ios::binary);
dst << src.rdbuf();
```

C ++ 17

Con C ++ 17, la forma estándar de copiar un archivo es incluir el encabezado `<filesystem>` y usar `copy_file` :

```
std::fileystem::copy_file("source_filename", "dest_filename");
```

La biblioteca del sistema de archivos se desarrolló originalmente como `boost.filesystem` y finalmente se fusionó con ISO C ++ a partir de C ++ 17.

¿Revisar el final del archivo dentro de una condición de bucle, mala práctica?

`eof` devuelve `true` solo **después de** leer el final del archivo. NO indica que la próxima lectura será el final de la transmisión.

```
while (!f.eof())
{
  // Everything is OK

  f >> buffer;

  // What if *only* now the eof / fail bit is set?

  /* Use `buffer` */
}
```

Podrías escribir correctamente:

```
while (!f.eof())
{
  f >> buffer >> std::ws;

  if (f.fail())
    break;

  /* Use `buffer` */
}
```

pero

```
while (f >> buffer)
{
  /* Use `buffer` */
}
```

Es más sencillo y menos propenso a errores.

Otras referencias:

- `std::ws` : descarta los espacios en blanco iniciales de un flujo de entrada
- `std::basic_ios::fail` : devuelve `true` si se ha producido un error en la secuencia asociada

Escribir archivos con configuraciones locales no estándar

Si necesita escribir un archivo con una configuración regional diferente a la predeterminada, puede usar `std::locale` y `std::basic_ios::imbue()` para hacer eso para una secuencia de archivos específica:

Guía de uso:

- Siempre debe aplicar un local a una secuencia antes de abrir el archivo.
- Una vez que se haya imbuido el flujo, no debe cambiar la configuración regional.

Razones para las restricciones: Imbuir una secuencia de archivos con una configuración regional tiene un comportamiento indefinido si la configuración regional actual no es independiente del estado o no apunta al principio del archivo.

Las transmisiones UTF-8 (y otras) no son independientes del estado. Además, una secuencia de archivos con una configuración regional UTF-8 puede intentar leer el marcador de la lista de materiales del archivo cuando se abre; así que solo abrir el archivo puede leer los caracteres del archivo y no estará al principio.

```
#include <iostream>
#include <fstream>
#include <locale>

int main()
{
  std::cout << "User-preferred locale setting is "
            << std::locale("").name().c_str() << std::endl;

  // Write a floating-point value using the user's preferred locale.
  std::ofstream ofs1;
  ofs1.imbue(std::locale(""));
  ofs1.open("file1.txt");
  ofs1 << 78123.456 << std::endl;

  // Use a specific locale (names are system-dependent)
  std::ofstream ofs2;
  ofs2.imbue(std::locale("en_US.UTF-8"));
  ofs2.open("file2.txt");
  ofs2 << 78123.456 << std::endl;

  // Switch to the classic "C" locale
  std::ofstream ofs3;
  ofs3.imbue(std::locale::classic());
  ofs3.open("file3.txt");
  ofs3 << 78123.456 << std::endl;
}
```

El cambio explícito a la configuración regional clásica "C" es útil si su programa utiliza una configuración regional predeterminada diferente y desea garantizar un estándar fijo para leer y escribir archivos. Con una configuración regional preferida de "C", el ejemplo escribe

```
78,123.456
78,123.456
78123.456
```

Si, por ejemplo, el entorno local preferido es alemán y, por lo tanto, utiliza un formato de número diferente, el ejemplo escribe

```
78 123,456
78,123.456
78123.456
```

(note la coma decimal en la primera línea).

Lea Archivo I / O en línea: https://riptutorial.com/es/cplusplus/topic/496/archivo-i---o

Capítulo 7: Archivos de encabezado

Observaciones

En C ++, como en C, el compilador de C ++ y el proceso de compilación hacen uso del preprocesador de C. Según lo especificado en el manual del preprocesador GNU C, un archivo de encabezado se define de la siguiente manera:

> Un archivo de encabezado es un archivo que contiene declaraciones C y definiciones de macro (consulte Macros) que se compartirán entre varios archivos de origen. Solicita el uso de un archivo de encabezado en su programa incluyéndolo, con la directiva de preprocesamiento de C '#include'.
>
> Los archivos de encabezado tienen dos propósitos.
>
> - Los archivos de encabezado del sistema declaran las interfaces a partes del sistema operativo. Los incluye en su programa para proporcionar las definiciones y declaraciones que necesita para invocar llamadas de sistema y bibliotecas.
> - Sus propios archivos de encabezado contienen declaraciones de interfaces entre los archivos de origen de su programa. Cada vez que tenga un grupo de declaraciones relacionadas y definiciones de macro, todas o la mayoría de las cuales son necesarias en varios archivos de origen diferentes, es una buena idea crear un archivo de encabezado para ellos.

Sin embargo, para el preprocesador de C en sí, un archivo de encabezado no es diferente de un archivo de origen.

El esquema de organización del archivo de encabezado / fuente es simplemente una convención estándar y fuertemente establecida por varios proyectos de software con el fin de proporcionar separación entre la interfaz y la implementación.

Aunque el propio estándar de C ++ no lo impone formalmente, seguir la convención del archivo fuente / encabezado es altamente recomendable y, en la práctica, ya es casi ubicuo.

Tenga en cuenta que los archivos de encabezado pueden ser reemplazados como una convención de estructura de archivos de proyecto por la próxima característica de los módulos, que aún debe considerarse para su inclusión en un futuro estándar de C ++ en el momento de la escritura (por ejemplo, C ++ 20).

Examples

Ejemplo básico

El siguiente ejemplo contendrá un bloque de código que se debe dividir en varios archivos de origen, como se indica en los comentarios de `// filename` .

Archivos fuente

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
  return global_value; // return 42;
}
```

Los archivos de encabezado luego se incluyen en otros archivos de origen que desean utilizar la funcionalidad definida por la interfaz del encabezado, pero no requieren conocimiento de su implementación (por lo tanto, reduciendo el acoplamiento de código). El siguiente programa hace uso del encabezado `my_function.h` como se definió anteriormente:

```
// main.cpp

#include <iostream>       // A C++ Standard Library header.
#include "my_function.h"  // A personal header

int main(int argc, char** argv) {
  std::cout << my_function() << std::endl;
  return 0;
```

```
}
```

El proceso de compilación

Dado que los archivos de encabezado a menudo forman parte de un flujo de trabajo del proceso de compilación, un proceso de compilación típico que hace uso de la convención de archivo de encabezado / fuente generalmente hará lo siguiente.

Suponiendo que el archivo de encabezado y el archivo de código fuente ya están en el mismo directorio, un programador ejecutaría los siguientes comandos:

```
g++ -c my_function.cpp       # Compiles the source file my_function.cpp
                             # --> object file my_function.o

g++ main.cpp my_function.o   # Links the object file containing the
                             # implementation of int my_function()
                             # to the compiled, object version of main.cpp
                             # and then produces the final executable a.out
```

Alternativamente, si uno desea compilar `main.cpp` en un archivo de objeto primero, y luego vincular solo los archivos de objeto como el paso final:

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

Plantillas en archivos de encabezado

Las plantillas requieren la generación de código en tiempo de compilación: una función de plantilla, por ejemplo, se convertirá efectivamente en múltiples funciones distintas una vez que una función de plantilla esté parametrizada por el uso en el código fuente.

Esto significa que la función de plantilla, la función de miembro y las definiciones de clase no pueden delegarse a un archivo de código fuente separado, ya que cualquier código que utilizará cualquier construcción con plantilla requiere conocimiento de su definición para generar generalmente cualquier código derivado.

Por lo tanto, el código de plantilla, si se coloca en encabezados, también debe contener su definición. Un ejemplo de esto es a continuación:

```
// templated_function.h

template <typename T>
T* null_T_pointer() {
  T* type_point = NULL; // or, alternatively, nullptr instead of NULL
                        // for C++11 or later
  return type_point;
}
```

Lea Archivos de encabezado en línea: https://riptutorial.com/es/cplusplus/topic/7211/archivos-de-encabezado

Capítulo 8: Aritmética de punto flotante

Examples

Los números de punto flotante son raros

El primer error que casi todos los programadores cometen es suponer que este código funcionará según lo previsto:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

El programador novato asume que esto resumirá cada número en el rango `0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99` , para obtener el resultado `199` la respuesta matemáticamente correcta.

Dos cosas suceden que hacen esto falso:

1. El programa como está escrito nunca concluye. `a` nunca se vuelve igual a `2` , y el bucle nunca termina.
2. Si reescribimos la lógica del bucle para verificar `a < 2` lugar, el bucle termina, pero el total termina siendo algo diferente de `199` . En las máquinas compatibles con IEEE754, a menudo suman aproximadamente `201` lugar.

La razón por la que esto sucede es que **los números de punto flotante representan aproximaciones de sus valores asignados** .

El ejemplo clásico es el siguiente cálculo:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Si bien lo que vemos el programador son tres números escritos en base10, lo que ven el compilador (y el hardware subyacente) son números binarios. Debido a que `0.1` , `0.2` y `0.3` requieren una división perfecta por `10` cual es bastante fácil en un sistema base-10, pero imposible en un sistema base-2, estos números deben almacenarse en formatos imprecisos, de manera similar a como se muestra el número `1/3` tiene que ser almacenado en la forma imprecisa `0.333333333333333...` en base-10.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a =     0011111111011100110011001100110011001100110011001100110011001100110011010; //imperfect
```

```
representation of 0.1
double b =     0011111111001001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.2
double c =     0011111111010011001100110011001100110011001100110011001100110011; //imperfect
representation of 0.3
double a + b = 0011111111010011001100110011001100110011001100110011001100110100; //Note that
this is not quite equal to the "canonical" 0.3!
```

Lea Aritmética de punto flotante en línea: https://riptutorial.com/es/cplusplus/topic/5115/aritmetica-de-punto-flotante

Capítulo 9: Arrays

Introducción

Las matrices son elementos del mismo tipo que se colocan en ubicaciones de memoria contiguas. Los elementos pueden ser referenciados individualmente por un identificador único con un índice agregado.

Esto le permite declarar múltiples valores de variables de un tipo específico y acceder a ellos individualmente sin necesidad de declarar una variable para cada valor.

Examples

Tamaño de matriz: tipo seguro en tiempo de compilación.

```
#include <stddef.h>     // size_t, ptrdiff_t

//---------------------------------- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
    -> Size
{ return n; }


//---------------------------------- Usage:

#include <iostream>
using namespace std;
auto main()
    -> int
{
    int const   a[]     = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
    Size const  n       = n_items( a );
    int         b[n]    = {};       // An array of the same size as a.

    (void) b;
    cout << "Size = " << n << "\n";
}
```

El lenguaje C para tamaño de matriz, `sizeof(a)/sizeof(a[0])` , aceptará un puntero como argumento y, por lo general, dará un resultado incorrecto.

Para C ++ 11

usando C ++ 11 puedes hacer:

```
std::extent<decltype(MyArray)>::value;
```

Ejemplo:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

Hasta C ++ 17 (a partir de este escrito) C ++ no tenía un lenguaje central incorporado ni una utilidad de biblioteca estándar para obtener el tamaño de una matriz, pero esto puede implementarse pasando la matriz *por referencia* a una plantilla de función, como mostrado anteriormente. Punto fino pero importante: el parámetro de tamaño de la plantilla es un `size_t` , algo inconsistente con el tipo de resultado de la función `Size` con signo, para acomodar el compilador g ++ que a veces insiste en `size_t` para la coincidencia de la plantilla.

Con C ++ 17 y versiones posteriores, se puede usar `std::size` , que está especializado para arreglos.

Matriz en bruto de tamaño dinámico

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>            // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }

auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\n";
    cout << "n? ";
    int const   n   = int_from( cin );
    int*        a   = new int[n];       // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';

    delete[] a;
}
```

Un programa que declara una matriz `T a[n];` donde `n` se determina un tiempo de ejecución, se puede compilar con ciertos compiladores que admiten *matrices de longitud variable* (*VLA)* C99 como una extensión de lenguaje. Pero los VLA no son compatibles con C ++ estándar. Este ejemplo muestra cómo asignar manualmente una matriz de tamaño dinámico a través de una `new[]` expresión `new[]` ,

```
int*        a   = new int[n];       // ← Allocation of array of n items.
```

... luego utilícelo, y finalmente deséchelo mediante una `delete[]` -expresión:

```
delete[] a;
```

La matriz asignada aquí tiene valores indeterminados, pero se puede inicializar con cero simplemente agregando un paréntesis vacío `()` , así: `new int[n]()` . Más generalmente, para un tipo de elemento arbitrario, esto realiza una *inicialización de valores* .

Como parte de una función en una jerarquía de llamadas, este código no sería seguro de excepción, ya que una excepción antes de la expresión `delete[]` (y después de la `new[]`) causaría una pérdida de memoria. Una forma de abordar ese problema es automatizar la limpieza, por ejemplo, mediante un puntero inteligente `std::unique_ptr` . Pero una forma generalmente mejor de abordarlo es simplemente usar `std::vector` : para eso está `std::vector` .

Expandiendo la matriz de tamaño dinámico usando std :: vector.

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>            // std::sort
#include <iostream>
#include <vector>               // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;      // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); }  // Expands as necessary.
    }

    sort( a.begin(), a.end() );
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` es una plantilla de clase de biblioteca estándar que proporciona la noción de una matriz de tamaño variable. Se encarga de toda la administración de la memoria, y el búfer es contiguo, por lo que se puede pasar un puntero al búfer (por ejemplo, `&v[0]` o `v.data()`) a las funciones de API que requieren una matriz sin formato. Incluso se puede expandir un `vector` en tiempo de ejecución, por ejemplo, a través de la función miembro `push_back` que agrega un elemento.

La complejidad de la secuencia de *n* operaciones `push_back` , incluida la copia o el movimiento involucrado en las expansiones vectoriales, se amortiza O (*n*). "Amortizado": en promedio.

Internamente, esto generalmente se logra cuando el vector *duplica* su tamaño de búfer, su capacidad, cuando se necesita un búfer más grande. Por ejemplo, para un búfer que comienza como tamaño 1 y se duplica repetidamente según sea necesario para *n* = 17 llamadas `push_back` , esto implica 1 + 2 + 4 + 8 + 16 = 31 operaciones de copia, que es menos de 2 × *n* = 34. Y más generalmente, la suma de esta secuencia no puede exceder de 2 × *n* .

En comparación con el ejemplo de matriz sin formato de tamaño dinámico, este código basado en `vector` no requiere que el usuario suministre (y sepa) el número de elementos por adelantado. En su lugar, el vector se expande según sea necesario, para cada nuevo valor de elemento especificado por el usuario.

Una matriz de matriz sin formato de tamaño fijo (es decir, una matriz sin formato 2D).

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   n_rows  = 3;
    int const   n_cols  = 7;
    int const   m[n_rows][n_cols] =             // A raw array matrix.
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];       // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Salida:

```
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
```

C ++ no admite sintaxis especial para indexar una matriz multidimensional. En su lugar, una matriz de este tipo se ve como una matriz de matrices (posiblemente de matrices, etc.), y se utiliza la notación de índice único ordinario `[ i ]` para cada nivel. En el ejemplo anterior, `m[y]` refiere a la fila `y` de `m` , donde `y` es un índice basado en cero. Entonces esta fila puede ser indexado a su vez, por ejemplo, `m[y][x]` , que se refiere a la `x` ésimo elemento - o columna - de fila `y` .

Es decir, el último índice varía más rápidamente, y en la declaración, el rango de este índice, que es el número de columnas por fila, es el último y el tamaño "más interno" especificado.

Como C ++ no proporciona soporte integrado para matrices de tamaño dinámico, aparte de la asignación dinámica, una matriz de tamaño dinámico a menudo se implementa como una clase. Luego, la notación de indexación de matriz sin procesar `m[y][x]` tiene algún costo, ya sea al exponer la implementación (de modo que, por ejemplo, una vista de una matriz transpuesta se vuelve prácticamente imposible) o al agregar un poco de sobrecarga y pequeños inconvenientes cuando se hace al regresar un objeto proxy del `operator[]` . Y así, la notación de indexación para tal abstracción puede y será generalmente diferente, tanto en el aspecto como en el orden de los índices, por ejemplo, `m(x,y)` o `m.at(x,y)` o `m.item(x,y)` .

Una matriz de tamaño dinámico utilizando std :: vector para almacenamiento.

Desafortunadamente, a partir de C ++ 14 no hay una clase de matriz de tamaño dinámico en la biblioteca estándar de C ++. Clases de matriz que apoyan el tamaño dinámico son sin embargo disponibles a partir de una serie de bibliotecas 3 [a] parte, incluyendo la biblioteca Boost Matrix (una sub-biblioteca dentro de la biblioteca Boost).

Si no desea una dependencia en Boost o alguna otra biblioteca, entonces la matriz de tamaño dinámico de un hombre pobre en C ++ es como

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

… Donde `vector` es `std::vector` . La matriz se crea aquí copiando un vector de fila *n* veces donde *n* es el número de filas, aquí 3. Tiene la ventaja de proporcionar la misma notación de indexación `m[y][x]` que para una matriz de matriz sin formato de tamaño fijo, pero es un poco ineficiente porque implica una asignación dinámica para cada fila, y es un poco inseguro porque es posible cambiar inadvertidamente el tamaño de una fila.

Un enfoque más seguro y eficiente es usar un solo vector como *almacenamiento* para la matriz y asignar el código del cliente (*x* , *y*) a un índice correspondiente en ese vector:

```
// A dynamic size matrix using std::vector for storage.

//---------------------------------------- Machinery:
#include <algorithm>        // std::copy
#include <assert.h>         // assert
#include <initializer_list> // std::initializer_list
#include <vector>           // std::vector
#include <stddef.h>         // ptrdiff_t

namespace my {
    using Size = ptrdiff_t;
    using std::initializer_list;
    using std::vector;

    template< class Item >
    class Matrix
    {
    private:
```

```
        vector<Item>    items_;
        Size            n_cols_;

        auto index_for( Size const x, Size const y ) const
            -> Size
        { return y*n_cols_ + x; }

    public:
        auto n_rows() const -> Size { return items_.size()/n_cols_; }
        auto n_cols() const -> Size { return n_cols_; }

        auto item( Size const x, Size const y )
            -> Item&
        { return items_[index_for(x, y)]; }

        auto item( Size const x, Size const y ) const
            -> Item const&
        { return items_[index_for(x, y)]; }

        Matrix(): n_cols_( 0 ) {}

        Matrix( Size const n_cols, Size const n_rows )
            : items_( n_cols*n_rows )
            , n_cols_( n_cols )
        {}

        Matrix( initializer_list< initializer_list<Item> > const& values )
            : items_()
            , n_cols_( values.size() == 0? 0 : values.begin()->size() )
        {
            for( auto const& row : values )
            {
                assert( Size( row.size() ) == n_cols_ );
                items_.insert( items_.end(), row.begin(), row.end() );
            }
        }
    };
}  // namespace my

//---------------------------------------- Usage:
using my::Matrix;

auto some_matrix()
    -> Matrix<int>
{
    return
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };
}

#include <iostream>
#include <iomanip>
using namespace std;
auto main() -> int
{
    Matrix<int> const m = some_matrix();
    assert( m.n_cols() == 7 );
    assert( m.n_rows() == 3 );
```

```
    for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
    {
        for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
        {
            cout << setw( 4 ) << m.item( x, y );        // ← Note: not `m[y][x]`!
        }
        cout << '\n';
    }
}
```

Salida:

```
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
```

El código anterior no es de grado industrial: está diseñado para mostrar los principios básicos y satisfacer las necesidades de los estudiantes que aprenden C ++.

Por ejemplo, uno puede definir sobrecargas de `operator()` para simplificar la notación de indexación.

Inicialización de matriz

Una matriz es solo un bloque de ubicaciones de memoria secuencial para un tipo específico de variable. Las matrices se asignan de la misma manera que las variables normales, pero con corchetes anexados a su nombre `[]` que contienen el número de elementos que caben en la memoria de la matriz.

El siguiente ejemplo de una matriz utiliza el tipo `int` , el nombre variable `arrayOfInts` y el número de elementos `[5]` que la matriz tiene espacio:

```
int arrayOfInts[5];
```

Una matriz se puede declarar e inicializar al mismo tiempo así

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

Al inicializar una matriz al enumerar todos sus miembros, no es necesario incluir el número de elementos dentro de los corchetes. Será calculado automáticamente por el compilador. En el siguiente ejemplo, es 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

También es posible inicializar solo los primeros elementos mientras se asigna más espacio. En este caso, es obligatorio definir la longitud entre paréntesis. Lo siguiente asignará una matriz de longitud 5 con inicialización parcial, el compilador inicializa todos los elementos restantes con el valor estándar del tipo de elemento, en este caso cero.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Las matrices de otros tipos de datos básicos pueden inicializarse de la misma manera.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize

char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize

double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};

string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

También es importante tener en cuenta que al acceder a los elementos de la matriz, el índice de elementos de la matriz (o posición) comienza desde 0.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

Lea Arrays en línea: https://riptutorial.com/es/cplusplus/topic/3017/arrays

Capítulo 10: Atributos

Sintaxis

- [[detalles]]: atributo simple sin argumentos
- [[detalles (argumentos)]]: Atributo con argumentos
- __attribute (detalles): No estándar GCC / Clang / IBM específico
- __declspec (detalles): no estándar MSVC específico

Examples

[[sin retorno]]

C ++ 11

C ++ 11 introdujo el atributo `[[noreturn]]` . Se puede usar para que una función indique que la función no regresa a la persona que llama, ya sea ejecutando una declaración de *retorno* , o llegando al final si es cuerpo (es importante tener en cuenta que esto no se aplica a las funciones `void` , ya que devuelva a la persona que llama, simplemente no devuelven ningún valor). Dicha función puede terminar llamando a `std::terminate` o `std::exit` , o lanzando una excepción. También vale la pena señalar que una función de este tipo puede regresar ejecutando `longjmp` .

Por ejemplo, la función a continuación siempre lanzará una excepción o llamará `std::terminate` , por lo que es un buen candidato para `[[noreturn]]` :

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

Este tipo de funcionalidad permite al compilador finalizar una función sin una declaración de retorno si sabe que el código nunca se ejecutará. Aquí, debido a que la llamada a `ownAssertFailureHandler` (definida anteriormente) en el código a continuación nunca regresará, el compilador no necesita agregar código debajo de esa llamada:

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
            sequence.push_back(i);
        return sequence;
    }
```

```
    ownAssertFailureHandler("Negative number passed to createSequence()"s);
    // return std::vector<int>{}; //< Not needed because of [[noreturn]]
}
```

Es un comportamiento indefinido si la función realmente regresará, por lo que no se permite lo siguiente:

```
[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
        ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]
}
```

Tenga en cuenta que `[[noreturn]]` se utiliza principalmente en las funciones void. Sin embargo, esto no es un requisito, permitiendo que las funciones se usen en la programación genérica:

```
template<class InconsistencyHandler>
double fortyTwoDivideBy(int i) {
    if (i == 0)
         i = InconsistencyHandler::correct(i);
    return 42. / i;
}

struct InconsistencyThrower {
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("Unknown
inconsistency"s); }
}

struct InconsistencyChangeToOne {
    static int correct(int i) { return 1; }
}

double fortyTwo = fortyTwoDivideBy<InconsistencyChangeToOne>(0);
double unreachable = fortyTwoDivideBy<InconsistencyThrower>(0);
```

Las siguientes funciones de biblioteca estándar tienen este atributo:

- std :: abortar
- std :: exit
- std :: quick_exit
- std :: inesperado
- std :: terminar
- std :: rethrow_exception
- std :: throw_with_nested
- std :: nested_exception :: rethrow_nested

[[caer a través]]

C ++ 17

Cada vez que se termina un `case` en un `switch` , se ejecutará el código del siguiente caso. Este último se puede prevenir usando la declaración 'break'. Dado que este llamado comportamiento

fallido puede introducir errores cuando no está previsto, varios compiladores y analizadores estáticos emiten una advertencia al respecto.

A partir de C ++ 17, se introdujo un atributo estándar para indicar que la advertencia no es necesaria cuando el código debe cumplirse. Los compiladores pueden dar advertencias de forma segura cuando un caso finaliza sin `break` o `[[fallthrough]]` y tiene al menos una declaración.

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "Using modern C++" << std::endl;
        [[fallthrough]]; // > No warning
    case 1998:
    case 2003:
        standard = input;
}
```

Consulte la propuesta para obtener ejemplos más detallados sobre cómo se puede usar `[[fallthrough]]` .

[[obsoleto]] y [[obsoleto ("motivo")]]

C ++ 14

C ++ 14 introdujo una forma estándar de desaprobar funciones a través de atributos. `[[deprecated]]` se puede usar para indicar que una función está en desuso. `[[deprecated("reason")]]` permite agregar una razón específica que puede ser mostrada por el compilador.

```
void function(std::unique_ptr<A> &&a);

// Provides specific message which helps other programmers fixing there code
[[deprecated("Use the variant with unique_ptr instead, this function will be removed in the
next release")]]
void function(std::auto_ptr<A> a);

// No message, will result in generic warning if called.
[[deprecated]]
void function(A *a);
```

Este atributo puede ser aplicado a:

- la declaracion de una clase
- un nombre de typedef
- una variable
- un miembro de datos no estáticos
- Una función
- una enumeración
- una plantilla de especialización

(ref. c ++ 14 borrador estándar : 7.6.5 Atributo obsoleto)

[[nodiscard]]

C ++ 17

El atributo `[[nodiscard]]` se puede usar para indicar que el valor de retorno de una función no debe ignorarse cuando se realiza una llamada de función. Si se ignora el valor de retorno, el compilador debe dar una advertencia sobre esto. El atributo se puede agregar a:

- Definición de una función
- Un tipo

Agregar el atributo a un tipo tiene el mismo comportamiento que agregar el atributo a cada función que devuelve este tipo.

```
template<typename Function>
[[nodiscard]] Finally<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0);                    // Just to make comments clear!
    ++i;                               // i == 1
    auto exit1 = onExit([&i]{ --i; }); // Reduce by 1 on exiting f()
    ++i;                               // i == 2
    onExit([&i]{ --i; });              // BUG: Reducing by 1 directly
                                       //      Compiler warning expected
    std::cout << i << std::end;        // Expected: 2, Real: 1
}
```

Consulte la propuesta para obtener ejemplos más detallados sobre cómo se puede usar `[[nodiscard]]` .

Nota: Los detalles de la implementación de `Finally` / `onExit` se omiten en el ejemplo, vea Finally / ScopeExit .

[[maybe_unused]]

El atributo `[[maybe_unused]]` se crea para indicar en el código que cierta lógica podría no ser utilizada. Esto se vincula a menudo con las condiciones del preprocesador, donde se puede usar o no. Como los compiladores pueden dar advertencias sobre variables no utilizadas, esta es una forma de suprimirlas indicando la intención.

Un ejemplo típico de las variables que se necesitan en las construcciones de depuración cuando no se necesitan en producción son los valores de retorno que indican el éxito. En las compilaciones de depuración, la condición debe ser confirmada, aunque en producción, estas afirmaciones se han eliminado.

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo",
stringifiedLicenseInfo);
assert(mapInsertResult.second); // We only get called during startup, so we can't be in the
map
```

Un ejemplo más complejo son diferentes tipos de funciones de ayuda que se encuentran en un

espacio de nombres sin nombre. Si estas funciones no se utilizan durante la compilación, un compilador puede dar una advertencia sobre ellas. Idealmente, le gustaría protegerlos con las mismas etiquetas de preprocesador que la persona que llama, aunque como esto podría volverse complejo, el atributo `[[maybe_unused]]` es una alternativa más `[[maybe_unused]]` mantener.

```
namespace {
    [[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
    // TODO: Reuse this on BSD, MAC ...
    [[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
#if OS == "WINDOWS"
      return createWindowsConfigFilePath(relativePath);
#elif OS == "LINUX"
      return createLinuxConfigFilePath(relativePath);
#else
#error "OS is not yet supported"
#endif
}
```

Consulte la propuesta para obtener ejemplos más detallados sobre cómo se puede usar `[[maybe_unused]]`.

Lea Atributos en línea: https://riptutorial.com/es/cplusplus/topic/5251/atributos

Capítulo 11: auto

Observaciones

La palabra clave `auto` es un nombre de tipo que representa un tipo deducido automáticamente.

Ya era una palabra clave reservada en C ++ 98, heredada de C. En versiones anteriores de C ++, se podía usar para indicar explícitamente que una variable tiene una duración de almacenamiento automática:

```
int main()
{
  auto int i = 5; // removing auto has no effect
}
```

Ese viejo significado ahora se ha eliminado.

Examples

Muestra auto básica

La palabra clave `auto` proporciona la deducción automática del tipo de una variable.

Es especialmente conveniente cuando se trata de nombres tipográficos largos:

```
std::map< std::string, std::shared_ptr< Widget > > table;
// C++98
std::map< std::string, std::shared_ptr< Widget > >::iterator i = table.find( "42" );
// C++11/14/17
auto j = table.find( "42" );
```

con rango basado en bucles :

```
vector<int> v = {0, 1, 2, 3, 4, 5};
for(auto n: v)
    std::cout << n << ' ';
```

con lambdas :

```
auto f = [](){ std::cout << "lambda\n"; };
f();
```

Para evitar la repetición del tipo:

```
auto w = std::make_shared< Widget >();
```

Para evitar copias sorprendentes e innecesarias:

```
auto myMap = std::map<int,float>();
myMap.emplace(1,3.14);

std::pair<int,float> const& firstPair2 = *myMap.begin();  // copy!
auto const& firstPair = *myMap.begin();  // no copy!
```

El motivo de la copia es que el tipo devuelto es en realidad `std::pair<const int,float>` !

Plantillas de auto y expresión

`auto` también puede causar problemas cuando las plantillas de expresión entran en juego:

```
auto mult(int c) {
    return c * std::valarray<int>{1};
}

auto v = mult(3);
std::cout << v[0]; // some value that could be, but almost certainly is not, 3.
```

La razón es que el `operator*` en `valarray` le proporciona un objeto proxy que se refiere al `valarray` como un medio de evaluación perezosa. Al usar `auto` , estás creando una referencia que cuelga. En lugar de `mult` ha devuelto un `std::valarray<int>` , entonces el código definitivamente se imprimiría 3.

auto, const, y referencias

La palabra clave `auto` por sí misma representa un tipo de valor, similar a `int` o `char` . Se puede modificar con la palabra clave `const` y el símbolo `&` para representar un tipo const o un tipo de referencia, respectivamente. Estos modificadores se pueden combinar.

En este ejemplo, `s` es un tipo de valor (su tipo se deducirá como `std::string`), por lo que cada iteración del bucle `for` *copia* una cadena del vector en `s` .

```
std::vector<std::string> strings = { "stuff", "things", "misc" };
for(auto s : strings) {
    std::cout << s << std::endl;
}
```

Si el cuerpo del bucle modifica `s` (por ejemplo, al llamar a `s.append(" and stuff")`), solo se modificará esta copia, no el miembro original de las `strings` .

Por otro lado, si `s` se declara con `auto&` será un tipo de referencia (se inferirá que es `std::string&`), por lo que en cada iteración del bucle se le asignará una *referencia* a una cadena en el vector:

```
for(auto& s : strings) {
    std::cout << s << std::endl;
}
```

En el cuerpo de este bucle, las modificaciones a `s` afectarán directamente el elemento de las `strings` que hace referencia.

Finalmente, si `s` se declara `const auto&` , será un tipo de referencia const, lo que significa que en cada iteración del bucle se le asignará una *referencia const* a una cadena en el vector:

```
for(const auto& s : strings) {
    std::cout << s << std::endl;
}
```

Dentro del cuerpo de este bucle, `s` no se puede modificar (es decir, no hay métodos no const pueden ser llamados en él).

Cuando se utiliza el modo `auto` con el rango `for` bucles, generalmente es una buena práctica usar `const auto&` si el cuerpo del bucle no modifica la estructura que se está repitiendo, ya que esto evita copias innecesarias.

Tipo de retorno final

`auto` se utiliza en la sintaxis para el tipo de retorno final:

```
auto main() -> int {}
```

que es equivalente a

```
int main() {}
```

Mayormente útil combinado con `decltype` para usar parámetros en lugar de `std::declval<T>` :

```
template <typename T1, typename T2>
auto Add(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs) { return lhs + rhs; }
```

Lambda genérica (C ++ 14)

C ++ 14

C ++ 14 permite usar `auto` en argumento lambda

```
auto print = [](const auto& arg) { std::cout << arg << std::endl; };

print(42);
print("hello world");
```

Esa lambda es mayormente equivalente a

```
struct lambda {
    template <typename T>
    auto operator ()(const T& arg) const {
        std::cout << arg << std::endl;
    }
};
```

y entonces

```
lambda print;

print(42);
print("hello world");
```

objetos de auto y proxy

A veces, el `auto` puede comportarse de la forma no esperada por un programador. El tipo deduce la expresión, incluso cuando la deducción de tipo no es lo correcto.

Como ejemplo, cuando se utilizan objetos proxy en el código:

```
std::vector<bool> flags{true, true, false};
auto flag = flags[0];
flags.push_back(true);
```

Aquí la `flag` no sería `bool` , pero `std::vector<bool>::reference` , ya que para la especialización `bool` del `vector` de plantilla el `operator []` devuelve un objeto proxy con el operador de conversión `operator bool` definido.

Cuando `flags.push_back(true)` modifica el contenedor, esta pseudo-referencia podría terminar colgando, refiriéndose a un elemento que ya no existe.

También hace posible la siguiente situación:

```
void foo(bool b);

std::vector<bool> getFlags();

auto flag = getFlags()[5];
foo(flag);
```

El `vector` se descarta inmediatamente, por lo que la `flag` es una pseudo-referencia a un elemento que ha sido descartado. La llamada a `foo` invoca un comportamiento indefinido.

En casos como este, puede declarar una variable con `auto` e inicializarla mediante la conversión al tipo que desea deducir:

```
auto flag = static_cast<bool>(getFlags()[5]);
```

pero en ese punto, simplemente reemplazar el `auto` por `bool` tiene más sentido.

Otro caso donde los objetos proxy pueden causar problemas son las plantillas de expresión . En ese caso, las plantillas a veces no están diseñadas para durar más allá de la expresión completa actual por razones de eficiencia, y el uso del objeto proxy en la siguiente causa un comportamiento indefinido.

Lea auto en línea: https://riptutorial.com/es/cplusplus/topic/2421/auto

Capítulo 12: Bucles

Introducción

Una instrucción de bucle ejecuta un grupo de instrucciones repetidamente hasta que se cumple una condición. Hay 3 tipos de bucles primitivos en C ++: para, while y do ... while.

Sintaxis

- *sentencia* while (*condición*);
- hacer *enunciado* while (*expresión*);
- para (*para-init-sentencia* ; *condición* ; *expresión*) *sentencia* ;
- para *(para-gama-declaración: Con fines de gama-inicializador) Declaración;*
- rotura ;
- continuar

Observaciones

`algorithm` llamadas de `algorithm` son generalmente preferibles a los bucles escritos a mano.

Si desea algo que ya hace un algoritmo (o algo muy similar), la llamada al algoritmo es más clara, a menudo más eficiente y menos propensa a errores.

Si necesita un bucle que haga algo bastante simple (pero requeriría una confusa maraña de carpetas y adaptadores si estuviera usando un algoritmo), simplemente escriba el bucle.

Examples

Basado en rango para

C ++ 11

`for` bucles pueden utilizarse para recorrer en iteración los elementos de un rango basado en iteradores, sin usar un índice numérico o acceder directamente a los iteradores:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

Esto iterará sobre cada elemento en `v` , con `val` obteniendo el valor del elemento actual. La siguiente declaración:

```
for (for-range-declaration : for-range-initializer ) statement
```

es equivalente a:

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

C ++ 17

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr;
    auto __end = end-expr; // end is allowed to be a different type than begin in C++17
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Este cambio se introdujo para el soporte planificado de Ranges TS en C ++ 20.

En este caso, nuestro bucle es equivalente a:

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
    }
}
```

Tenga en cuenta que `auto val` declara un tipo de valor, que será una copia de un valor almacenado en el rango (lo estamos inicializando desde el iterador). Si los valores almacenados en el rango son costosos de copiar, es posible que desee utilizar `const auto &val` . Tampoco está obligado a utilizar `auto` ; puede usar un nombre de tipo apropiado, siempre que sea implícitamente convertible del tipo de valor del rango.

Si necesita acceso al iterador, basado en rango no puede ayudarlo (no sin esfuerzo, al menos).

Si desea referenciarlo, puede hacerlo:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(float &val: v)
{
    std::cout << val << " ";
}
```

Podría iterar en la referencia `const` si tiene un contenedor `const` :

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

Se utilizarían referencias de reenvío cuando el iterador de secuencia devuelva un objeto proxy y usted necesite operar en ese objeto de forma no `const` . Nota: lo más probable es que confunda a los lectores de su código.

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

El tipo de "gama" proporcionada al intervalo basado- `for` puede ser uno de los siguientes:

- Matrices de idiomas:

  ```
  float arr[] = {0.4f, 12.5f, 16.234f};

  for(auto val: arr)
  {
      std::cout << val << " ";
  }
  ```

  Tenga en cuenta que la asignación de una matriz dinámica no cuenta:

  ```
  float *arr = new float[3]{0.4f, 12.5f, 16.234f};

  for(auto val: arr) //Compile error.
  {
      std::cout << val << " ";
  }
  ```

- Cualquier tipo que tenga funciones miembro `begin()` y `end()` , que devuelve los iteradores a los elementos del tipo. Los contenedores de la biblioteca estándar califican, pero los tipos definidos por el usuario también se pueden usar:

  ```
  struct Rng
  {
      float arr[3];

      // pointers are iterators
      const float* begin() const {return &arr[0];}
      const float* end() const   {return &arr[3];}
      float* begin() {return &arr[0];}
      float* end()   {return &arr[3];}
  ```

```
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

- Cualquier tipo que tenga funciones de `begin(type)` y `end(type)` que no sean miembros que se pueden encontrar mediante búsqueda dependiente de argumento, según el `type` . Esto es útil para crear un tipo de rango sin tener que modificar el tipo de clase en sí:

```
namespace Mine
{
    struct Rng {float arr[3];};

    // pointers are iterators
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

En bucle

Un bucle `for` ejecuta instrucciones en el `loop body` del `loop body` , mientras que la `condition` del bucle es verdadera. Antes de que la `initialization statement` bucle se ejecute exactamente una vez. Después de cada ciclo, se ejecuta la parte de `iteration execution` .

Un bucle `for` se define de la siguiente manera:

```
for (/*initialization statement*/; /*condition*/; /*iteration execution*/)
{
    // body of the loop
}
```

Explicación de las declaraciones del marcador de posición:

- `initialization statement` : esta sentencia se ejecuta solo una vez, al comienzo del bucle `for` . Puede ingresar una declaración de múltiples variables de un tipo, como `int i = 0, a = 2, b = 3`

. Estas variables solo son válidas en el ámbito del bucle. Las variables definidas antes del bucle con el mismo nombre se ocultan durante la ejecución del bucle.

- `condition` : esta declaración se evalúa antes de cada ejecución del *cuerpo del bucle* y cancela el bucle si se evalúa como `false` .
- `iteration execution` : esta instrucción se ejecuta después del *cuerpo* del bucle, antes de la siguiente evaluación de la *condición* , a menos que el bucle `for` sea abortado en el *cuerpo* (por `break` , `goto` , `return` o una excepción lanzada). Puede ingresar varias declaraciones en la parte de `iteration execution` , como `a++, b+=10, c=b+a` .

El equivalente aproximado de un `for` de bucle, reescrito como un `while` bucle es:

```
/*initialization*/
while (/*condition*/)
{
    // body of the loop; using 'continue' will skip to increment part below
    /*iteration execution*/
}
```

El caso más común para usar un bucle `for` es ejecutar sentencias un número específico de veces. Por ejemplo, considere lo siguiente:

```
for(int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

Un bucle válido también es:

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {
    std::cout << a << " " << b << " " << c << std::endl;
}
```

Un ejemplo de ocultar variables declaradas antes de un bucle es:

```
int i = 99; //i = 99
for(int i = 0; i < 10; i++) { //we declare a new variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 99
```

Pero si desea utilizar la variable ya declarada y no ocultarla, omita la parte de declaración:

```
int i = 99; //i = 99
for(i = 0; i < 10; i++) { //we are using already declared variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 10
```

Notas:

- Las instrucciones de inicialización e incremento pueden realizar operaciones no relacionadas con la declaración de condición, o nada en absoluto, si así lo desea. Pero por

razones de legibilidad, es una buena práctica realizar solo operaciones directamente relevantes para el bucle.
- Una variable declarada en la declaración de inicialización es visible solo dentro del alcance del bucle `for` y se libera al finalizar el bucle.
- No olvide que la variable que se declaró en la `initialization statement` se puede modificar durante el bucle, así como la variable verificada en la `condition` .

Ejemplo de un bucle que cuenta de 0 a 10:

```
for (int counter = 0; counter <= 10; ++counter)
{
    std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

Explicación de los fragmentos de código:

- `int counter = 0` inicializa la variable `counter` a 0. (Esta variable solo se puede usar dentro del bucle `for`).
- `counter <= 10` es una condición booleana que verifica si el `counter` es menor o igual a 10. Si es `true` , el bucle se ejecuta. Si es `false` , el bucle termina.
- `++counter` es una operación de incremento que incrementa el valor de `counter` en 1 antes de la siguiente verificación de condición.

Al dejar todas las declaraciones en blanco, puede crear un bucle infinito:

```
// infinite loop
for (;;)
    std::cout << "Never ending!\n";
```

El `while` de bucle equivalente de la anterior es:

```
// infinite loop
while (true)
    std::cout << "Never ending!\n";
```

Sin embargo, aún se puede dejar un bucle infinito utilizando las instrucciones `break` , `goto` o `return` o lanzando una excepción.

El siguiente ejemplo común de iteración sobre todos los elementos de una colección STL (por ejemplo, un `vector`) sin usar el encabezado `<algorithm>` es:

```
std::vector<std::string> names = {"Albert Einstein", "Stephen Hawking", "Michael Ellis"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Mientras bucle

Un `while` de bucle ejecuta las instrucciones varias veces hasta que la condición dada se evalúa

como `false` . Esta declaración de control se usa cuando no se sabe, de antemano, cuántas veces se debe ejecutar un bloque de código.

Por ejemplo, para imprimir todos los números del 0 al 9, se puede usar el siguiente código:

```
int i = 0;
while (i < 10)
{
    std::cout << i << " ";
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; "0 1 2 3 4 5 6 7 8 9" is printed to the console
```

C ++ 17

Tenga en cuenta que desde C ++ 17, las 2 primeras declaraciones pueden combinarse

```
while (int i = 0; i < 10)
//... The rest is the same
```

Para crear un bucle infinito, se puede usar la siguiente construcción:

```
while (true)
{
    // Do something forever (however, you can exit the loop by calling 'break'
}
```

Hay otra variante de `while` bucles, a saber, `do...while` construct. Vea el ejemplo del bucle do-while para más información.

Declaración de variables en condiciones.

En la condición de la `for` y `while` bucles, también se permitió declarar un objeto. Se considerará que este objeto está en el alcance hasta el final del bucle, y persistirá en cada iteración del bucle:

```
for (int i = 0; i < 5; ++i) {
    do_something(i);
}
// i is no longer in scope.

for (auto& a : some_container) {
    a.do_something();
}
// a is no longer in scope.

while(std::shared_ptr<Object> p = get_object()) {
   p->do_something();
}
// p is no longer in scope.
```

Sin embargo, no está permitido hacer lo mismo con un bucle `do...while` while; en su lugar, declare la variable antes del bucle, y (opcionalmente) encierre tanto la variable como el bucle dentro de un ámbito local si desea que la variable salga del alcance después de que finalice el bucle:

```
//This doesn't compile
do {
    s = do_something();
} while (short s > 0);

// Good
short s;
do {
    s = do_something();
} while (s > 0);
```

Esto se debe a que la parte de *instrucción* de un bucle `do...while` while (el cuerpo del bucle) se evalúa antes de llegar a la parte de *expresión* (`while`), y por lo tanto, cualquier declaración en la *expresión* no será visible durante la primera iteración de lazo.

Bucle Do-while

Un bucle *do-while* es muy similar a un bucle *while,* excepto que la condición se comprueba al final de cada ciclo, no al principio. Por lo tanto, se garantiza que el bucle se ejecuta al menos una vez.

El siguiente código imprimirá `0` , ya que la condición se evaluará como `false` al final de la primera iteración:

```
int i =0;
do
{
    std::cout << i;
    ++i; // Increment counter
}
while (i < 0);
std::cout << std::endl; // End of line; 0 is printed to the console
```

Nota: No olvide el punto y coma al final de `while(condition);` , que se necesita en la construcción *do-while* .

En contraste con el bucle *do-* while, lo siguiente no imprimirá nada, porque la condición se evalúa como `false` al comienzo de la primera iteración:

```
int i =0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; nothing is printed to the console
```

Nota: Un bucle *while* se puede salir sin la condición de convertirse en falsa utilizando una `break` , `goto` , o `return` comunicado.

```
int i = 0;
do
{
    std::cout << i;
```

```
    ++i; // Increment counter
    if (i > 5)
    {
        break;
    }
}
while (true);
std::cout << std::endl; // End of line; 0 1 2 3 4 5 is printed to the console
```

Un bucle *do-while* trivial también es ocasionalmente usado para escribir macros que requieren su propio ámbito (en cuyo caso el punto y coma final se omite de la definición de la macro y requiere ser proporcionado por el usuario):

```
#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// Only the call to f1 is protected by the condition here
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// All calls are protected here
if (cond) GOOD_MACRO(var);
```

Declaraciones de control de bucle: romper y continuar

Las instrucciones de control de bucle se utilizan para cambiar el flujo de ejecución desde su secuencia normal. Cuando la ejecución deja un ámbito, se destruyen todos los objetos automáticos que se crearon en ese ámbito. El `break` y `continue` son instrucciones de control de bucle.

La `break` comunicado termina un bucle sin ninguna consideración adicional.

```
for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // this will immediately exit our loop
    std::cout << i << '\n';
}
```

El código de arriba se imprimirá:

```
1
2
3
```

La declaración de `continue` no sale inmediatamente del bucle, sino que se salta el resto del cuerpo del bucle y va a la parte superior del bucle (incluida la verificación de la condición).

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // evaluates to true if i is even
        continue; // this will immediately go back to the start of the loop
    /* the next line will only be reached if the above "continue" statement
```

```
       does not execute  */
    std::cout << i << " is an odd number\n";
}
```

El código de arriba se imprimirá:

```
1 is an odd number
3 is an odd number
5 is an odd number
```

Debido a que tales cambios en el flujo de control a veces son difíciles de entender para los humanos, los `break` y los `continue` se usan con moderación. Una implementación más sencilla suele ser más fácil de leer y entender. Por ejemplo, el primer bucle `for` con la `break` anterior podría reescribirse como:

```
for (int i = 0; i < 4; i++)
{
    std::cout << i << '\n';
}
```

El segundo ejemplo con `continue` podría reescribirse como:

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 != 0) {
        std::cout << i << " is an odd number\n";
    }
}
```

Rango-para sobre un sub-rango

Usando bucles de rango de base, puede recorrer una subparte de un contenedor u otro rango dado generando un objeto proxy que califica para bucles basados en rango.

```
template<class Iterator, class Sentinel=Iterator>
struct range_t {
  Iterator b;
  Sentinel e;
  Iterator begin() const { return b; }
  Sentinel end() const { return e; }
  bool empty() const { return begin()==end(); }
  range_t without_front( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {std::next(b, count), e};
  }
  range_t without_back( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {b, std::prev(e, count)};
```

```
  }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinal e ) {
  return {b,e};
}
template<class Iterable>
auto range( Iterable& r ) {
  using std::begin; using std::end;
  return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
  auto r = range(c);
  if (r.empty()) return r;
  return r.without_front();
}
```

Ahora podemos hacer:

```
std::vector<int> v = {1,2,3,4};

for (auto i : except_first(v))
  std::cout << i << '\n';
```

e imprimir

```
2
3
4
```

Tenga en cuenta que los objetos intermedios generados en la parte `for(:range_expression)` del bucle `for` habrán caducado antes `for` comience el bucle `for` .

Lea Bucles en línea: https://riptutorial.com/es/cplusplus/topic/589/bucles

Capítulo 13: Búsqueda de nombre dependiente del argumento

Examples

Que funciones se encuentran

Las funciones se encuentran al recopilar primero un conjunto de *"clases asociadas"* y *"espacios de nombres asociados"* que incluyen uno o más de los siguientes, según el tipo de argumento `T` Primero, mostremos las reglas para los nombres de especialización de clases, enumeración y de plantilla de clase.

- Si `T` es una clase anidada, enumeración de miembros, entonces la clase circundante.
- Si `T` es una enumeración (que *también* puede ser un miembro de la clase!), El espacio de nombres más interna de la misma.
- Si `T` es una clase (que *también* se pueden anidar!), Todas sus clases base y la propia clase. El espacio de nombres más interno de todas las clases asociadas.
- Si `T` es un `ClassTemplate<TemplateArguments>` (¡esto *también* es una clase!), Las clases y los espacios de nombres asociados con los argumentos de tipo de plantilla, el espacio de nombres de cualquier argumento de plantilla de plantilla y la clase circundante de cualquier argumento de plantilla de plantilla, si un argumento de plantilla es una plantilla de miembro.

Ahora hay algunas reglas para los tipos incorporados también

- Si `T` es un puntero a `U` o matriz de `U` , las clases y los espacios de nombres asociados con `U` Ejemplo: `void (*fptr)(A); f(fptr);` , incluye los espacios de nombres y las clases asociadas con `void(A)` (ver la siguiente regla).
- Si `T` es un tipo de función, las clases y los espacios de nombres asociados con los tipos de parámetros y de retorno. Ejemplo: `void(A)` incluiría los espacios de nombres y las clases asociadas con `A`
- Si `T` es un puntero a miembro, las clases y los espacios de nombres asociados con el tipo de miembro (pueden aplicarse tanto a puntero a funciones de miembro como puntero a miembro de datos). Ejemplo: `BA::*p; void (A::*pf)(B); f(p); f(pf);` incluye los espacios de nombres y las clases asociadas con `A` , `B` , `void(B)` (que aplica la viñeta anterior para los tipos de función).

Todas las funciones y plantillas dentro de todos los espacios de nombres asociados se encuentran por búsqueda dependiente del argumento. Además, *se encuentran funciones de amigo en el ámbito del espacio de nombres declaradas en clases asociadas* , que normalmente no son visibles. Sin embargo, se ignoran las directivas de uso.

Todas las siguientes llamadas de ejemplo son válidas, sin calificar `f` por el nombre del espacio de nombres en la llamada.

```
namespace A {
```

```
   struct Z { };
   namespace I { void g(Z); }
   using namespace I;

   struct X { struct Y { }; friend void f(Y) { } };
   void f(X p) { }
   void f(std::shared_ptr<X> p) { }
}

// example calls
f(A::X());
f(A::X::Y());
f(std::make_shared<A::X>());

g(A::Z()); // invalid: "using namespace I;" is ignored!
```

Lea Búsqueda de nombre dependiente del argumento en línea:
https://riptutorial.com/es/cplusplus/topic/5163/busqueda-de-nombre-dependiente-del-argumento

Capítulo 14: C ++ Streams

Observaciones

El constructor predeterminado de `std::istream_iterator` construye un iterador que representa el final de la secuencia. Por lo tanto, `std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(), ...` significa copiar desde la posición actual en `ifs` hasta el final.

Examples

Corrientes de cuerda

`std::ostringstream` es una clase cuyos objetos se parecen a un flujo de salida (es decir, puede escribir en ellos a través del `operator<<`), pero en realidad almacena los resultados de escritura y los proporciona en forma de un flujo.

Considere el siguiente código corto:

```
#include <sstream>
#include <string>


using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

La línea

```
ostringstream ss;
```

crea tal objeto. Este objeto se manipula primero como un flujo regular:

```
ss << "the answer to everything is " << 42;
```

Después de eso, sin embargo, el flujo resultante se puede obtener de esta manera:

```
const string result = ss.str();
```

(el `result` la cadena será igual a `"the answer to everything is 42"`).

Esto es principalmente útil cuando tenemos una clase para la cual se ha definido la serialización de flujo y para la que queremos una forma de cadena. Por ejemplo, supongamos que tenemos

alguna clase

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

Para obtener la representación de cadena de un objeto `foo` ,

```
foo f;
```

podríamos usar

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Entonces el `result` contiene la representación de cadena del objeto `foo` .

Leyendo un archivo hasta el final.

Leyendo un archivo de texto línea por línea

Una forma adecuada de leer un archivo de texto línea por línea hasta el final generalmente no está clara en la documentación de `ifstream` . Consideremos algunos errores comunes cometidos por los programadores principiantes de C ++ y una forma adecuada de leer el archivo.

Líneas sin caracteres de espacios en blanco.

En aras de la simplicidad, supongamos que cada línea del archivo no contiene símbolos de espacios en blanco.

`ifstream` tiene el `operator bool()` , que devuelve verdadero cuando una secuencia no tiene errores y está lista para leer. Además, `ifstream::operator >>` devuelve una referencia al flujo mismo, de modo que podemos leer y verificar EOF (así como errores) en una línea con una sintaxis muy elegante:

```
std::ifstream ifs("1.txt");
std::string s;
while(ifs >> s) {
    std::cout << s << std::endl;
}
```

Líneas con caracteres de espacio en blanco.

`ifstream::operator >>` lee la secuencia hasta que `ifstream::operator >>` cualquier carácter de espacio en blanco, por lo que el código anterior imprimirá las palabras de una línea en líneas separadas. Para leer todo hasta el final de la línea, use `std::getline` lugar de `ifstream::operator >>` . `getline` devuelve la referencia al hilo con el que trabajó, por lo que está disponible la misma sintaxis:

```
while(std::getline(ifs, s)) {
    std::cout << s << std::endl;
}
```

Obviamente, `std::getline` también debe usarse para leer un archivo de una sola línea hasta el final.

Leyendo un archivo en un búfer a la vez

Finalmente, leamos el archivo desde el principio hasta el final sin detenerse en ningún carácter, incluidos los espacios en blanco y las nuevas líneas. Si sabemos que el tamaño exacto del archivo o el límite superior de la longitud es aceptable, podemos cambiar el tamaño de la cadena y luego leer:

```
s.resize(100);
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    s.begin());
```

De lo contrario, necesitamos insertar cada carácter al final de la cadena, por lo que `std::back_inserter` es lo que necesitamos:

```
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::back_inserter(s));
```

Alternativamente, es posible inicializar una colección con datos de flujo, usando un constructor con argumentos de rango de iterador:

```
std::vector v(std::istreambuf_iterator<char>(ifs),
    std::istreambuf_iterator<char>());
```

Tenga en cuenta que estos ejemplos también son aplicables si `ifs` se abre como archivo binario:

```
std::ifstream ifs("1.txt", std::ios::binary);
```

Copiando arroyos

Un archivo se puede copiar en otro archivo con secuencias e iteradores:

```
std::ofstream ofs("out.file");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
```

```
    std::ostream_iterator<char>(ofs));
ofs.close();
```

o redirigido a cualquier otro tipo de flujo con una interfaz compatible. Por ejemplo, el flujo de red Boost.Asio:

```
boost::asio::ip::tcp::iostream stream;
stream.connect("example.com", "http");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::ostream_iterator<char>(stream));
stream.close();
```

Arrays

Como los iteradores pueden considerarse como una generalización de los punteros, los contenedores STL en los ejemplos anteriores pueden reemplazarse con matrices nativas. Aquí es cómo analizar números en una matriz:

```
int arr[100];
std::copy(std::istream_iterator<char>(ifs), std::istream_iterator<char>(), arr);
```

Tenga cuidado con el desbordamiento del búfer, ya que las matrices no se pueden redimensionar sobre la marcha después de que se asignaron. Por ejemplo, si el código anterior se alimenta con un archivo que contiene más de 100 números enteros, intentará escribir fuera de la matriz y se ejecutará en un comportamiento indefinido.

Imprimiendo colecciones con iostream

Impresión básica

`std::ostream_iterator` permite imprimir el contenido de un contenedor STL en cualquier flujo de salida sin ciclos explícitos. El segundo argumento del constructor `std::ostream_iterator` establece el delimitador. Por ejemplo, el siguiente código:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

imprimirá

```
1 ! 2 ! 3 ! 4 !
```

Tipo implícito de reparto

`std::ostream_iterator` permite emitir implícitamente el tipo de contenido del contenedor. Por

ejemplo, sintonicemos `std::cout` para imprimir valores de punto flotante con 3 dígitos después del punto decimal:

```
std::cout << std::setprecision(3);
std::fixed(std::cout);
```

e instanciar `std::ostream_iterator` con `float` , mientras que los valores contenidos permanecen `int` :

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

por lo que el código de arriba rinde

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

A pesar de `std::vector` hold `int` s.

Generación y transformación.

`std::generate` funciones `std::generate` , `std::generate_n` y `std::transform` proporcionan una herramienta muy poderosa para la manipulación de datos sobre la marcha. Por ejemplo, tener un vector:

```
std::vector<int> v = {1,2,3,4,8,16};
```

podemos imprimir fácilmente el valor booleano de "x es par" para cada elemento:

```
std::boolalpha(std::cout); // print booleans alphabetically
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),
[](int val) {
    return (val % 2) == 0;
});
```

o imprimir el elemento cuadrado:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Impresión de N números al azar delimitados por espacios:

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Arrays

Al igual que en la sección sobre la lectura de archivos de texto, casi todas estas consideraciones pueden aplicarse a matrices nativas. Por ejemplo, imprimamos valores cuadrados de una matriz nativa:

```
int v[] = {1,2,3,4,8,16};
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Análisis de archivos

Análisis de archivos en contenedores STL

`istream_iterator` s son muy útiles para leer secuencias de números u otros datos analizables en contenedores STL sin bucles explícitos en el código.

Usando tamaño de contenedor explícito:

```
std::vector<int> v(100);
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    v.begin());
```

o con la inserción de iterador:

```
std::vector<int> v;
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    std::back_inserter(v));
```

Tenga en cuenta que los números en el archivo de entrada se pueden dividir por cualquier número de caracteres de espacio en blanco y líneas nuevas.

Análisis de tablas de texto heterogéneas

Como `istream::operator>>` lee el texto hasta que aparezca un símbolo de espacio en blanco, se puede usar en la condición `while` para analizar tablas de datos complejas. Por ejemplo, si tenemos un archivo con dos números reales seguidos por una cadena (sin espacios) en cada línea:

```
1.12 3.14 foo
2.1 2.2 barr
```

se puede analizar de esta manera:

```
std::string s;
double a, b;
while(ifs >> a >> b >> s) {
    std::cout << a << " " << b << " " << s << std::endl;
}
```

Transformación

Cualquier función de manipulación de rangos puede usarse con rangos `std::istream_iterator` . Uno de ellos es `std::transform` , que permite procesar datos sobre la marcha. Por ejemplo, leamos valores enteros, multiplíquelos por 3.14 y almacenemos el resultado en un contenedor de punto flotante:

```
std::vector<double> v(100);
std::transform(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
v.begin(),
[](int val) {
    return val * 3.14;
});
```

Lea C ++ Streams en línea: https://riptutorial.com/es/cplusplus/topic/7660/c-plusplus-streams

Capítulo 15: Campos de bits

Introducción

Los campos de bits empaquetan las estructuras C y C ++ para reducir el tamaño. Esto parece indoloro: especifique el número de bits para los miembros, y el compilador hace el trabajo de mezclar bits. La restricción es la incapacidad de tomar la dirección de un miembro de campo de bit, ya que se almacena de forma combinada. `sizeof()` también está deshabilitado.

El costo de los campos de bits es un acceso más lento, ya que la memoria debe recuperarse y las operaciones a nivel de bits deben aplicarse para extraer o modificar los valores de los miembros. Estas operaciones también se agregan al tamaño ejecutable.

Observaciones

¿Qué tan caras son las operaciones bitwise? Supongamos una estructura de campo no bit simple:

```
struct foo {
    unsigned x;
    unsigned y;
}
static struct foo my_var;
```

En algún código posterior:

```
my_var.y = 5;
```

Si `sizeof (unsigned) == 4` , entonces x se almacena al inicio de la estructura, y se almacena y 4 bytes. El código de ensamblaje generado puede parecerse a:

```
loada register1,#myvar     ; get the address of the structure
storei register1[4],#0x05  ; put the value '5' at offset 4, e.g., set y=5
```

Esto es sencillo porque x no está mezclado con y. Pero imagina redefinir la estructura con campos de bits:

```
struct foo {
    unsigned x : 4; /* Range 0-0x0f, or 0 through 15 */
    unsigned y : 4;
}
```

Tanto a `x` como a `y` se les asignarán 4 bits, compartiendo un solo byte. Por lo tanto, la estructura ocupa 1 byte, en lugar de 8. Considere el ensamblaje para establecer `y` ahora, asumiendo que termina en el mordisco superior:

```
loada  register1,#myvar        ; get the address of the structure
loadb  register2,register1[0]  ; get the byte from memory
andb   register2,#0x0f         ; zero out y in the byte, leaving x alone
orb    register2,#0x50         ; put the 5 into the 'y' portion of the byte
stb    register1[0],register2  ; put the modified byte back into memory
```

Esto puede ser una buena compensación si tenemos miles o millones de estas estructuras, y ayuda a mantener la memoria en la memoria caché o evita el intercambio, o podría inflar el ejecutable para empeorar estos problemas y demorar el procesamiento. Como con todas las cosas, usa el buen juicio.

Uso del controlador de dispositivo: evite los campos de bits como una estrategia de implementación inteligente para los controladores de dispositivo. Los diseños de almacenamiento de campo de bits no son necesariamente coherentes entre los compiladores, por lo que tales implementaciones no son portátiles. La lectura-modificación-escritura para establecer valores puede no hacer lo que los dispositivos esperan, causando comportamientos inesperados.

Examples

Declaración y uso

```
struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};
```

Aquí, cada uno de estos dos campos ocupará 1 bit en la memoria. Se especifica mediante `: 1` expresión después de los nombres de las variables. El tipo de base del campo de bits podría ser cualquier tipo integral (int de 8 bits a int de 64 bits). Se recomienda usar el tipo `unsigned` , de lo contrario pueden surgir sorpresas.

Si se requieren más bits, reemplace "1" con la cantidad de bits requeridos. Por ejemplo:

```
struct Date
{
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
    unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
    unsigned int Day:   5;  // 32
};
```

Toda la estructura está utilizando tan solo 22 bits, y con la configuración normal del compilador, `sizeof` esta estructura sería de 4 bytes.

El uso es bastante simple. Solo declara la variable, y úsala como una estructura ordinaria.

```
Date d;

d.Year = 2016;
d.Month = 7;
```

```
d.Day =  22;

std::cout << "Year:" << d.Year << std::endl <<
        "Month:" << d.Month << std::endl <<
        "Day:" << d.Day << std::endl;
```

Lea Campos de bits en línea: https://riptutorial.com/es/cplusplus/topic/2710/campos-de-bits

Capítulo 16: Categorías de valor

Examples

Significados de la categoría de valor

A las expresiones en C ++ se les asigna una categoría de valor particular, en función del resultado de esas expresiones. Las categorías de valor para las expresiones pueden afectar la resolución de sobrecarga de la función C ++.

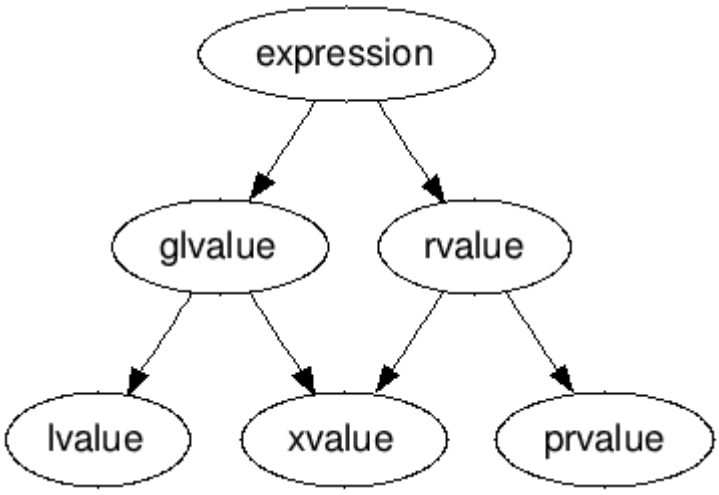
Las categorías de valor determinan dos propiedades importantes pero separadas de una expresión. Una propiedad es si la expresión tiene identidad. Una expresión tiene identidad si se refiere a un objeto que tiene un nombre de variable. Es posible que el nombre de la variable no esté involucrado en la expresión, pero el objeto aún puede tener uno.

La otra propiedad es si es legal moverse implícitamente del valor de la expresión. O más específicamente, si la expresión, cuando se usa como un parámetro de función, se unirá a los tipos de parámetros de valor r o no.

C ++ define 3 categorías de valores que representan la combinación útil de estas propiedades: lvalue (expresiones con identidad pero que no se pueden mover desde), xvalue (expresiones con identidad que se pueden mover desde), y prvalue (expresiones sin identidad que se pueden mover desde). C ++ no tiene expresiones que no tengan identidad y no puedan moverse.

C ++ define otras dos categorías de valores, cada una basada únicamente en una de estas propiedades: glvalue (expresiones con identidad) y rvalue (expresiones desde las que se puede mover). Estos actúan como agrupaciones útiles de las categorías anteriores.

Este gráfico sirve como ilustración:



prvalue

Una expresión prvalue (pure-rvalue) es una expresión que carece de identidad, cuya evaluación

se usa normalmente para inicializar un objeto y que se puede mover de forma implícita. Estos incluyen, pero no se limitan a:

- Expresiones que representan objetos temporales, como `std::string("123")` .
- Una expresión de llamada de función que no devuelve una referencia.
- Un literal (*excepto* un literal de cadena - esos son valores de l), como tiene `1` , `true` , `0.5f` , o `'a'`
- Una expresión lambda

La dirección incorporada del operador (`&`) no se puede aplicar a estas expresiones.

xvalor

Una expresión xvalue (valor eXpiring) es una expresión que tiene identidad y representa un objeto desde el cual se puede mover implícitamente. La idea general con las expresiones xvalue es que el objeto que representan se destruirá pronto (de ahí la parte "inspiradora"), y por lo tanto, mudarse de forma implícita está bien.

Dado:

```
struct X { int n; };
extern X x;

4;                    // prvalue: does not have an identity
x;                    // lvalue
x.n;                  // lvalue
std::move(x);         // xvalue
std::forward<X&>(x); // lvalue
X{4};                 // prvalue: does not have an identity
X{4}.n;               // xvalue: does have an identity and denotes resources
                      // that can be reused
```

valor

Una expresión lvalue es una expresión que tiene identidad, pero no se puede mover implícitamente. Entre ellas se encuentran las expresiones que consisten en un nombre de variable, un nombre de función, expresiones que son usos de operadores de desreferencia incorporados y expresiones que se refieren a referencias de valores.

El lvalue típico es simplemente un nombre, pero los lvalues también pueden venir en otros sabores:

```
struct X { ... };

X x;         // x is an lvalue
X* px = &x;  // px is an lvalue
*px = X{};   // *px is also an lvalue, X{} is a prvalue

X* foo_ptr();  // foo_ptr() is a prvalue
X& foo_ref();  // foo_ref() is an lvalue
```

Además, mientras que la mayoría de los literales (por ejemplo, `4` , `'x'` , etc.) son prvalores, los

literales de cadenas son valores l.

glvalue

Una expresión glvalue (un "lvalue generalizado") es cualquier expresión que tiene identidad, independientemente de si se puede mover o no. Esta categoría incluye lvalues (expresiones que tienen identidad pero no se pueden mover) y xvalues (expresiones que tienen identidad y se pueden mover desde), pero excluye prvalues (expresiones sin identidad).

Si una expresión tiene un *nombre* , es un glvalue:

```
struct X { int n; };
X foo();

X x;
x; // has a name, so it's a glvalue
std::move(x); // has a name (we're moving from "x"), so it's a glvalue
              // can be moved from, so it's an xvalue not an lvalue

foo(); // has no name, so is a prvalue, not a glvalue
X{};   // temporary has no name, so is a prvalue, not a glvalue
X{}.n; // HAS a name, so is a glvalue. can be moved from, so it's an xvalue
```

valor

Una expresión de rvalor es cualquier expresión a la que se puede mover implícitamente, independientemente de si tiene identidad.

Más precisamente, las expresiones rvalue se pueden usar como argumento para una función que toma un parámetro de tipo `T &&` (donde `T` es el tipo de `expr`). *Solo* se pueden dar expresiones de valor como argumentos a tales parámetros de función; Si se usa una expresión sin valor, la resolución de sobrecarga seleccionará cualquier función que no use un parámetro de referencia de valor. Y si no existe ninguno, entonces obtienes un error.

La categoría de expresiones rvalue incluye todas las expresiones xvalue y prvalue, y solo esas expresiones.

La función de biblioteca estándar `std::move` existe para transformar explícitamente una expresión sin valor en un valor. Más específicamente, convierte la expresión en un xvalor, ya que incluso si antes era una expresión de prvalor sin identidad, al pasarla como parámetro a `std::move` , gana identidad (el nombre del parámetro de la función) y se convierte en un xvalor.

Considera lo siguiente:

```
std::string str("init");                      //1
std::string test1(str);                       //2
std::string test2(std::move(str));            //3

str = std::string("new value");               //4
std::string &&str_ref = std::move(str);       //5
std::string test3(str_ref);                   //6
```

`std::string` tiene un constructor que toma un solo parámetro de tipo `std::string&&` , comúnmente llamado "constructor de movimiento". Sin embargo, la categoría de valor de la expresión `str` no es un rvalue (específicamente es un lvalue), por lo que no puede llamar a esa sobrecarga del constructor. En su lugar, llama a `const std::string&` overload, el constructor de copia.

La línea 3 cambia las cosas. El valor de retorno de `std::move` es un `T&&` , donde `T` es el tipo base del parámetro pasado. Así que `std::move(str)` devuelve `std::string&&` . Una llamada de función cuyo valor de retorno es una referencia rvalue es una expresión rvalue (específicamente un valor x), por lo que puede llamar al constructor de movimiento de `std::string` . Después de la línea 3, `str` se ha movido desde (los contenidos de quién ahora están indefinidos).

La línea 4 pasa un operador temporal al operador de asignación de `std::string` . Esto tiene una sobrecarga que lleva un `std::string&&` . La expresión `std::string("new value")` es una expresión rvalue (específicamente un prvalue), por lo que puede llamar a esa sobrecarga. Por lo tanto, el temporal se mueve a `str` , reemplazando los contenidos indefinidos con contenidos específicos.

La línea 5 crea una referencia `str_ref` llamada `str_ref` que se refiere a `str` . Aquí es donde las categorías de valor se vuelven confusas.

Vea, mientras que `str_ref` es una referencia de rvalue a `std::string` , la categoría de valor de la expresión `str_ref` *no es un valor de r* . Es una expresión lvalue. Sí, en serio. Debido a esto, no se puede llamar al constructor de movimiento de `std::string` con la expresión `str_ref` . La línea 6, por lo tanto, *copia* el valor de `str` en `test3` .

Para moverlo, tendríamos que emplear `std::move` nuevamente.

Lea Categorías de valor en línea: https://riptutorial.com/es/cplusplus/topic/763/categorias-de-valor

Capítulo 17: Clases / Estructuras

Sintaxis

- variable.member_var = constante;
- variable.member_function ();
- variable_pointer-> member_var = constante;
- variable_pointer-> miembro_función ();

Observaciones

Tenga en cuenta que la ***única*** diferencia entre las palabras clave `struct` y `class` es que, de forma predeterminada, las variables miembro, las funciones miembro y las clases base de una `struct` son `public` , mientras que en una `class` son `private` . Los programadores de C ++ tienden a llamarlo una clase si tienen constructores y destructores, y la capacidad de imponer sus propios invariantes; o una estructura si es solo una simple colección de valores, pero el lenguaje C ++ en sí mismo no hace distinción.

Examples

Conceptos básicos de clase

Una *clase* es un tipo definido por el usuario. Se introduce una clase con la palabra clave `class` , `struct` o `union` . En el uso coloquial, el término "clase" generalmente se refiere solo a las clases no sindicalizadas.

Una clase es una colección de *miembros* de la *clase* , que puede ser:

- variables miembro (también llamadas "campos"),
- funciones miembro (también llamadas "métodos"),
- tipos de miembros o typedefs (por ejemplo, "clases anidadas"),
- Plantillas de miembros (de cualquier tipo: plantilla de variable, función, clase o alias)

Las palabras clave `class` y `struct` , llamadas *claves de clase* , son en gran medida intercambiables, excepto que el especificador de acceso predeterminado para miembros y bases es "privado" para una clase declarada con la clave de `class` y "público" para una clase declarada con la clave `struct` o `union` (cf. modificadores de acceso).

Por ejemplo, los siguientes fragmentos de código son idénticos:

```
struct Vector
{
    int x;
    int y;
    int z;
};
```

```
// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

Al declarar una clase` se agrega un nuevo tipo a su programa, y es posible crear una instancia de los objetos de esa clase mediante

```
Vector my_vector;
```

Se accede a los miembros de una clase usando la sintaxis de puntos.

```
my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my:vector.z = 7;
```

Especificadores de acceso

Hay tres palabras clave que actúan como **especificadores de acceso** . Estos limitan el acceso a los miembros de la clase siguiendo el especificador, hasta que otro especificador cambie de nuevo el nivel de acceso:

Palabra clave	Descripción
`public`	Todos tienen acceso
`protected`	Sólo la clase en sí, las clases derivadas y los amigos tienen acceso.
`private`	Sólo la clase en sí y los amigos tienen acceso.

Cuando el tipo se define con la palabra clave de `class` , el especificador de acceso predeterminado es `private` , pero si el tipo se define con la palabra clave `struct` , el especificador de acceso predeterminado es `public` :

```
struct MyStruct { int x; };
class MyClass { int x; };

MyStruct s;
s.x = 9; // well formed, because x is public

MyClass c;
c.x = 9; // ill-formed, because x is private
```

Los especificadores de acceso se usan principalmente para limitar el acceso a los campos y métodos internos, y obligan al programador a usar una interfaz específica, por ejemplo, para forzar el uso de captadores y definidores en lugar de hacer referencia a una variable

directamente:

```
class MyClass {

public: /* Methods: */

    int x() const noexcept { return m_x; }
    void setX(int const x) noexcept { m_x = x; }

private: /* Fields: */

    int m_x;

};
```

Usar `protected` es útil para permitir que cierta funcionalidad del tipo solo sea accesible para las clases derivadas, por ejemplo, en el siguiente código, el método `calculateValue()` solo es accesible para las clases derivadas de la clase base `Plus2Base` , como `FortyTwo` :

```
struct Plus2Base {
    int value() noexcept { return calculateValue() + 2; }
protected: /* Methods: */
    virtual int calculateValue() noexcept = 0;
};
struct FortyTwo: Plus2Base {
protected: /* Methods: */
    int calculateValue() noexcept final override { return 40; }
};
```

Tenga en cuenta que la palabra clave `friend` se puede usar para agregar excepciones de acceso a funciones o tipos para acceder a miembros protegidos y privados.

Las palabras clave `public` , `protected` y `private` también se pueden usar para otorgar o limitar el acceso a subobjetos de clase base. Ver el ejemplo de herencia .

Herencia

Las clases / estructuras pueden tener relaciones de herencia.

Si una clase / estructura `B` hereda de una clase / estructura `A` , esto significa que `B` tiene como padre `A` Decimos que `B` es una clase / estructura derivada de `A` , y `A` es la clase / estructura base.

```
struct A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};

//Make B inherit publicly (default) from A
struct B : A
```

```
{
};
```

Hay 3 formas de herencia para una clase / estructura:

- `public`
- `private`
- `protected`

Tenga en cuenta que la herencia predeterminada es la misma que la visibilidad predeterminada de los miembros: `public` si usa la palabra clave `struct` , y `private` para la palabra clave `class` .

Incluso es posible tener una `class` derivada de una `struct` (o viceversa). En este caso, la herencia predeterminada está controlada por el hijo, por lo que una `struct` que se deriva de una `class` se establecerá de forma predeterminada como herencia pública, y una `class` que se deriva de una `struct` tendrá herencia privada de forma predeterminada.

herencia `public` :

```
struct B : public A // or just `struct B : A`
{
    void foo()
    {
        p1 = 0; //well formed, p1 is public in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //well formed, p1 is public
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

herencia `private`

```
struct B : private A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is private in B
        p2 = 0; //well formed, p2 is private in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is private
b.p2 = 1; //ill formed, p2 is private
b.p3 = 1; //ill formed, p3 is inaccessible
```

herencia `protected` :

```
struct B : protected A
{
```

```
    void foo()
    {
        p1 = 0; //well formed, p1 is protected in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is protected
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

Tenga en cuenta que aunque se permite la herencia `protected` , el uso real de la misma es raro. Una instancia de cómo se usa la herencia `protected` en la aplicación es en la especialización de clase base parcial (generalmente denominada "polimorfismo controlado").

Cuando la POO era relativamente nueva, se decía con frecuencia que la herencia (pública) modelaba una relación "IS-A". Es decir, la herencia pública es correcta solo si una instancia de la clase derivada *es también una* instancia de la clase base.

Más tarde, esto se refinó en el Principio de Sustitución de Liskov : la herencia pública solo debe usarse cuando / si una instancia de la clase derivada puede sustituirse por una instancia de la clase base bajo cualquier circunstancia posible (y aún tiene sentido).

En general, se dice que la herencia privada incorpora una relación completamente diferente: "se implementa en términos de" (a veces se denomina relación "HAS-A"). Por ejemplo, una clase de `Stack` podría heredar de forma privada de una clase `Vector` . La herencia privada tiene una similitud mucho mayor con la agregación que con la herencia pública.

La herencia protegida casi nunca se usa, y no hay un acuerdo general sobre qué tipo de relación abarca.

Herencia virtual

Al usar la herencia, puede especificar la palabra clave `virtual` :

```
struct A{};
struct B: public virtual A{};
```

Cuando la clase `B` tiene una base virtual `A` , significa que `A` **residirá en la mayoría de la clase derivada** del árbol de herencia y, por lo tanto, la mayoría de la clase derivada también es responsable de inicializar esa base virtual:

```
struct A
{
    int member;
    A(int param)
    {
        member = param;
    }
};
```

```
struct B: virtual A
{
    B(): A(5){}
};

struct C: B
{
    C(): /*A(88)*/ {}
};

void f()
{
    C object; //error since C is not initializing it's indirect virtual base `A`
}
```

Si anulamos el comentario `/*A(88)*/` no obtendremos ningún error ya que `C` ahora está inicializando su base virtual indirecta `A`

También tenga en cuenta que cuando estamos creando un `object` variable, la mayoría de la clase derivada es `C` , por lo que `C` es responsable de crear (llamar al constructor de) `A` y, por lo tanto, el valor de `A::member` es `88` , no `5` (como lo sería si fuésemos creando objeto de tipo `B`).

Es útil para resolver el problema del diamante .

```
  A                                   A   A
 / \                                  |   |
B   C                                 B   C
 \ /                                   \ /
  D                                     D
virtual inheritance                 normal inheritance
```

`B` y `C` heredan de `A` , y `D` hereda de `B` y `C` , por lo que **hay 2 instancias de `A` en `D` !** Esto se traduce en ambigüedad cuando se accede al miembro de `A` a `D` , ya que el compilador no tiene forma de saber de qué clase desea acceder a ese miembro (¿el que `B` hereda o el que hereda `C` ?) .

La herencia virtual resuelve este problema: como la base virtual reside solo en la mayoría de los objetos derivados, solo habrá una instancia de `A` en `D`

```
struct A
{
    void foo() {}
};

struct B : public /*virtual*/ A {};
struct C : public /*virtual*/ A {};

struct D : public B, public C
{
    void bar()
    {
        foo(); //Error, which foo? B::foo() or C::foo()? - Ambiguous
    }
};
```

Eliminar los comentarios resuelve la ambigüedad.

Herencia múltiple

Aparte de la herencia única:

```
class A {};
class B : public A {};
```

También puede tener herencia múltiple:

```
class A {};
class B {};
class C : public A, public B {};
```

`C` ahora tendrá herencia de `A` y `B` al mismo tiempo.

Nota: esto puede generar ambigüedad si se usan los mismos nombres en varias `class` o `struct` heredadas. ¡Ten cuidado!

La ambigüedad en la herencia múltiple

La herencia múltiple puede ser útil en ciertos casos pero, a veces, algún tipo de problema se encuentra mientras se usa la herencia múltiple.

Por ejemplo: dos clases base tienen funciones con el mismo nombre que no se anulan en la clase derivada y si escribe código para acceder a esa función utilizando el objeto de la clase derivada, el compilador muestra error porque no puede determinar a qué función llamar. Aquí hay un código para este tipo de ambigüedad en herencia múltiple.

```
class base1
{
  public:
     void funtion( )
     { //code for base1 function }
};
class base2
{
    void function( )
     { // code for base2 function }
};

class derived : public base1, public base2
{

};

int main()
{
    derived obj;

  // Error because compiler can't figure out which function to call
  //either function( ) of base1 or base2 .
    obj.function( )
```

```
}
```

Pero, este problema se puede resolver utilizando la función de resolución de alcance para especificar qué función se debe clasificar como base1 o base2:

```
int main()
{
    obj.base1::function( );  // Function of class base1 is called.
    obj.base2::function( );  // Function of class base2 is called.
}
```

Acceso a los miembros de la clase

Para acceder a las variables y funciones miembro de un objeto de una clase, el `.` el operador es usado:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
// Accessing member variable a in var.
std::cout << var.a << std::endl;
// Assigning member variable b in var.
var.b = 1;
// Calling a member function.
var.foo();
```

Cuando se accede a los miembros de una clase a través de un puntero, el operador `->` se usa comúnmente. Alternativamente, la instancia puede ser anulada y la `.` Operador utilizado, aunque esto es menos común:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
SomeStruct *p = &var;
// Accessing member variable a in var via pointer.
std::cout << p->a << std::endl;
std::cout << (*p).a << std::endl;
// Assigning member variable b in var via pointer.
p->b = 1;
(*p).b = 1;
// Calling a member function via a pointer.
p->foo();
(*p).foo();
```

Cuando se accede a miembros de la clase estática, se utiliza el operador `::` , pero en el nombre de la clase en lugar de una instancia de la misma. Alternativamente, se puede acceder al

miembro estático desde una instancia o un puntero a una instancia usando el `.` o `->` operador, respectivamente, con la misma sintaxis que para acceder a miembros no estáticos.

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}

  static int c;
  static void bar() {}
};
int SomeStruct::c;

SomeStruct var;
SomeStruct* p = &var;
// Assigning static member variable c in struct SomeStruct.
SomeStruct::c = 5;
// Accessing static member variable c in struct SomeStruct, through var and p.
var.a = var.c;
var.b = p->c;
// Calling a static member function.
SomeStruct::bar();
var.bar();
p->bar();
```

Fondo

El operador `->` es necesario porque el operador de acceso miembro `.` Tiene prioridad sobre el operador de desreferenciación `*` .

Uno esperaría que `*pa` desreferenciara `p` (resultando en una referencia al objeto `p` que apunta) y luego accediendo a su miembro `a` . Pero, de hecho, intenta acceder al miembro `a` de `p` y luego desreferenciarlo. Ie `*pa` es equivalente a `*(pa)` . En el ejemplo anterior, esto daría como resultado un error del compilador debido a dos hechos: Primero, `p` es un puntero y no tiene un miembro `a` . En segundo lugar, `a` es un número entero y, por lo tanto, no se puede eliminar la referencia.

La solución poco común a este problema sería controlar explícitamente la prioridad: `(*p).a`

En cambio, el operador `->` casi siempre se usa. Es una mano corta para desreferenciar primero el puntero y luego acceder a él. Ie `(*p).a` es exactamente lo mismo que `p->a` .

El operador `::` es el operador de alcance, que se utiliza de la misma manera que para acceder a un miembro de un espacio de nombres. Esto se debe a que se considera que un miembro de la clase estática está dentro del alcance de esa clase, pero no se considera un miembro de las instancias de esa clase. El uso de lo normal `.` y `->` también está permitido para miembros estáticos, a pesar de que no sean miembros de instancia, por razones históricas; esto es útil para escribir código genérico en plantillas, ya que la persona que llama no necesita preocuparse de si una función miembro dada es estática o no estática.

Herencia privada: restringiendo la interfaz de clase base

La herencia privada es útil cuando se requiere restringir la interfaz pública de la clase:

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move();  // compile error
b.turn();  // OK
```

Este enfoque evita de manera eficiente el acceso a los métodos públicos A mediante el envío al puntero o referencia A:

```
B b;
A& a = static_cast<A&>(b); // compile error
```

En el caso de la herencia pública, dicha conversión proporcionará acceso a todos los métodos públicos A, a pesar de que existen formas alternativas de evitar esto en B derivada, como la ocultación:

```
class B : public A {
private:
    int move();
};
```

o privado utilizando:

```
class B : public A {
private:
    using A::move;
};
```

Entonces para ambos casos es posible:

```
B b;
A& a = static_cast<A&>(b); // OK for public inheritance
a.move(); // OK
```

Clases finales y estructuras.

C ++ 11

Derivar una clase puede estar prohibido con especificador `final` . Vamos a declarar una clase final:

```
class A final {
};
```

Ahora cualquier intento de subclase causará un error de compilación:

```
// Compilation error: cannot derive from final class:
class B : public A {
};
```

La clase final puede aparecer en cualquier lugar de la jerarquía de clases:

```
class A {
};

// OK.
class B final : public A {
};

// Compilation error: cannot derive from final class B.
class C : public B {
};
```

Amistad

La palabra clave `friend` se utiliza para otorgar acceso a otras clases y funciones a miembros privados y protegidos de la clase, incluso a través de su definición fuera del alcance de la clase.

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // A common use for a friend function is to overload the operator<< for streaming.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(Animal animal)
{
    std::cout << animal.weight << "\n";
}

class AnimalPrinter
{
public:
    void print(const Animal& animal)
    {
        // Because of the `friend class AnimalPrinter;" declaration, we are
        // allowed to access private members here.
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, Animal animal)
{
    os << "Animal height: " << animal.height << "\n";
    return os;
}
```

```
int main() {
    Animal animal = {10, 5};
    printWeight(animal);

    AnimalPrinter aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

```
10
10, 5
Animal height: 5
```

Clases / Estructuras Anidadas

Una `class` o `struct` también puede contener otra definición de `class` / `struct` dentro de sí misma, que se denomina "clase anidada"; en esta situación, la clase contenedora se conoce como la "clase adjunta". La definición de clase anidada se considera un miembro de la clase adjunta, pero por lo demás es separada.

```
struct Outer {
    struct Inner { };
};
```

Desde fuera de la clase adjunta, se accede a las clases anidadas mediante el operador de alcance. Sin embargo, desde el interior de la clase adjunta, las clases anidadas se pueden usar sin calificadores:

```
struct Outer {
    struct Inner { };

    Inner in;
};

// ...

Outer o;
Outer::Inner i = o.in;
```

Al igual que con una `class` / `struct` no anidada, las funciones miembro y las variables estáticas se pueden definir ya sea dentro de una clase anidada o en el espacio de nombres adjunto. Sin embargo, no se pueden definir dentro de la clase adjunta, debido a que se considera que es una clase diferente a la clase anidada.

```
// Bad.
struct Outer {
    struct Inner {
        void do_something();
    };

    void Inner::do_something() {}
```

```
};


// Good.
struct Outer {
    struct Inner {
        void do_something();
    };

};

void Outer::Inner::do_something() {}
```

Al igual que con las clases no anidadas, las clases anidadas se pueden declarar y definir posteriormente, siempre que se definan antes de usarlas directamente.

```
class Outer {
    class Inner1;
    class Inner2;

    class Inner1 {};

    Inner1 in1;
    Inner2* in2p;

  public:
    Outer();
    ~Outer();
};

class Outer::Inner2 {};

Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}
Outer::~Outer() {
    if (in2p) { delete in2p; }
}
```

C ++ 11

Antes de C ++ 11, las clases anidadas solo tenían acceso a nombres de tipo, miembros `static` y enumeradores de la clase adjunta; todos los demás miembros definidos en la clase adjunta estaban fuera de los límites.

C ++ 11

A partir de C ++ 11, las clases anidadas y sus miembros se tratan como si fueran `friend` de la clase adjunta y pueden acceder a todos sus miembros, de acuerdo con las reglas de acceso habituales; si los miembros de la clase anidada requieren la capacidad de evaluar uno o más miembros no estáticos de la clase adjunta, por lo tanto, se les debe pasar una instancia:

```
class Outer {
    struct Inner {
        int get_sizeof_x() {
            return sizeof(x); // Legal (C++11): x is unevaluated, so no instance is required.
        }
```

```
        int get_x() {
            return x; // Illegal: Can't access non-static member without an instance.
        }

        int get_x(Outer& o) {
            return o.x; // Legal (C++11): As a member of Outer, Inner can access private
members.
        }
    };

    int x;
};
```

A la inversa, la clase adjunta *no se* trata como un amigo de la clase anidada y, por lo tanto, no puede acceder a sus miembros privados sin un permiso explícito.

```
class Outer {
    class Inner {
        // friend class Outer;

        int x;
    };

    Inner in;

  public:
    int get_x() {
        return in.x; // Error: int Outer::Inner::x is private.
        // Uncomment "friend" line above to fix.
    }
};
```

Los amigos de una clase anidada no se consideran automáticamente amigos de la clase adjunta; Si también necesitan ser amigos de la clase adjunta, esto debe declararse por separado. A la inversa, como la clase adjunta no se considera automáticamente un amigo de la clase anidada, tampoco se considerarán amigos de la clase anexa amigos de la clase anidada.

```
class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Good.
    int o = out.o; // Error: int Outer::o is private.
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Error: int Outer::Inner::i is private.
```

```
    int o = out.o; // Good.
}
```

Al igual que con todos los demás miembros de la clase, las clases anidadas solo pueden nombrarse desde fuera de la clase si tienen acceso público. Sin embargo, puede acceder a ellos sin importar el modificador de acceso, siempre y cuando no los nombre explícitamente.

```
class Outer {
    struct Inner {
        void func() { std::cout << "I have no private taboo.\n"; }
    };

  public:
    static Inner make_Inner() { return Inner(); }
};

// ...

Outer::Inner oi; // Error: Outer::Inner is private.

auto oi = Outer::make_Inner(); // Good.
oi.func();                     // Good.
Outer::make_Inner().func();    // Good.
```

También puede crear un alias de tipo para una clase anidada. Si un alias de tipo está contenido en la clase adjunta, el tipo anidado y el alias de tipo pueden tener diferentes modificadores de acceso. Si el alias de tipo está fuera de la clase adjunta, requiere que la clase anidada, o un `typedef` misma, sea pública.

```
class Outer {
    class Inner_ {};

  public:
    typedef Inner_ Inner;
};

typedef Outer::Inner  ImOut; // Good.
typedef Outer::Inner_ ImBad; // Error.

// ...

Outer::Inner  oi; // Good.
Outer::Inner_ oi; // Error.
ImOut         oi; // Good.
```

Al igual que con otras clases, las clases anidadas pueden derivar o derivarse de otras clases.

```
struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

Esto puede ser útil en situaciones en las que la clase adjunta se deriva de otra clase, al permitir que el programador actualice la clase anidada según sea necesario. Esto se puede combinar con un typedef para proporcionar un nombre coherente para cada clase anidada de la clase envolvente:

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

  public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---

class DerivedOuter : public BaseOuter {
    // Note the use of the qualified typedef; BaseOuter::BaseInner_ is private.
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

  public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...

// Calls BaseOuter::BaseInner_::do_something();
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// Calls DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();
```

En el caso anterior, tanto `BaseOuter` como `DerivedOuter` suministran el tipo de miembro `Inner` , como `BaseInner_` y `DerivedInner_` , respectivamente. Esto permite que los tipos anidados se deriven sin romper la interfaz de la clase envolvente, y permite que el tipo anidado se utilice de forma polimórfica.

Tipos de miembros y alias

Una `class` o `struct` también puede definir alias de tipo de miembro, que son alias de tipo contenidos dentro de la clase y tratados como miembros de la misma clase.

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};
```

Al igual que los miembros estáticos, se accede a estos typedefs usando el operador de alcance, `::`.

```
IHaveATypedef::MyTypedef i = 5; // i is an int.

IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

Al igual que con los alias de tipo normal, cada alias de tipo miembro puede referirse a cualquier tipo definido o con alias antes, pero no después, de su definición. Del mismo modo, un typedef fuera de la definición de clase puede referirse a cualquier typedef accesible dentro de la definición de clase, siempre que venga después de la definición de clase.

```
template<typename T>
struct Helper {
    T get() const { return static_cast<T>(42); }
};

struct IHaveTypedefs {
//    typedef MyTypedef NonLinearTypedef; // Error if uncommented.
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IHaveTypedefs::MyTypedef        i; // x_i is an int.
IHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.

typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii;                  // ii is an int.
```

Los alias de tipo de miembro se pueden declarar con cualquier nivel de acceso y respetarán el modificador de acceso apropiado.

```
class TypedefAccessLevels {
    typedef int PrvInt;

  protected:
    typedef int ProInt;

  public:
    typedef int PubInt;
```

```
};

TypedefAccessLevels::PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
TypedefAccessLevels::ProInt pro_i; // Error: TypedefAccessLevels::ProInt is protected.
TypedefAccessLevels::PubInt pub_i; // Good.

class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
    ProInt pro_i; // Good.
    PubInt pub_i; // Good.
};
```

Esto se puede usar para proporcionar un nivel de abstracción, permitiendo que el diseñador de una clase cambie su funcionamiento interno sin romper el código que se basa en él.

```
class Something {
    friend class SomeComplexType;

    short s;
    // ...

  public:
    typedef SomeComplexType MyHelper;

    MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();
```

En esta situación, si la clase auxiliar se cambia de `SomeComplexType` a algún otro tipo, solo será necesario modificar el `typedef` y la declaración de `friend` ; Siempre que la clase auxiliar proporcione la misma funcionalidad, cualquier código que lo use como `Something::MyHelper` lugar de especificarlo por su nombre, por lo general funcionará sin ninguna modificación. De esta manera, minimizamos la cantidad de código que debe modificarse cuando se cambia la implementación subyacente, de modo que el nombre de tipo solo necesita cambiarse en una ubicación.

Esto también se puede combinar con `decltype` , si uno lo desea.

```
class SomethingElse {
    AnotherComplexType<bool, int, SomeThirdClass> helper;

  public:
    typedef decltype(helper) MyHelper;

  private:
    InternalVariable<MyHelper> ivh;

    // ...
```

```
  public:
    MyHelper& get_helper() const { return helper; }

    // ...
};
```

En esta situación, cambiar la implementación de `SomethingElse::helper` cambiará automáticamente el typedef para nosotros, debido a `decltype` . Esto minimiza el número de modificaciones necesarias cuando queremos cambiar de `helper` , lo que minimiza el riesgo de error humano.

Como con todo, sin embargo, esto puede ser llevado demasiado lejos. Si el nombre de tipo solo se usa una o dos veces internamente y cero veces externamente, por ejemplo, no es necesario proporcionar un alias para él. Si se usa cientos o miles de veces a lo largo de un proyecto, o si tiene un nombre lo suficientemente largo, entonces puede ser útil proporcionarlo como un typedef en lugar de usarlo siempre en términos absolutos. Hay que equilibrar la compatibilidad y la conveniencia con la cantidad de ruido innecesario creado.

Esto también se puede utilizar con clases de plantilla, para proporcionar acceso a los parámetros de la plantilla desde fuera de la clase.

```
template<typename T>
class SomeClass {
    // ...

  public:
    typedef T MyParam;
    MyParam getParam() { return static_cast<T>(42); }
};

template<typename T>
typename T::MyParam some_func(T& t) {
    return t.getParam();
}

SomeClass<int> si;
int i = some_func(si);
```

Esto se usa comúnmente con los contenedores, que normalmente proporcionarán su tipo de elemento, y otros tipos de ayuda, como alias de tipo de miembro. La mayoría de los contenedores en la biblioteca estándar de C ++, por ejemplo, proporcionan los siguientes 12 tipos de ayuda, junto con cualquier otro tipo especial que puedan necesitar.

```
template<typename T>
class SomeContainer {
    // ...

  public:
    // Let's provide the same helper types as most standard containers.
    typedef T                                   value_type;
    typedef std::allocator<value_type>          allocator_type;
    typedef value_type&                         reference;
    typedef const value_type&                   const_reference;
    typedef value_type*                         pointer;
```

```
    typedef const value_type*                       const_pointer;
    typedef MyIterator<value_type>                  iterator;
    typedef MyConstIterator<value_type>             const_iterator;
    typedef std::reverse_iterator<iterator>         reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t                                  size_type;
    typedef ptrdiff_t                               difference_type;
};
```

Antes de C ++ 11, también se usaba comúnmente para proporcionar un tipo de "plantilla `typedef` ", ya que la característica aún no estaba disponible; se han vuelto un poco menos comunes con la introducción de plantillas de alias, pero siguen siendo útiles en algunas situaciones (y se combinan con plantillas de alias en otras situaciones, lo que puede ser muy útil para obtener componentes individuales de un tipo complejo, como un puntero de función).). Comúnmente usan el `type` nombre para su alias de tipo.

```
template<typename T>
struct TemplateTypedef {
    typedef T type;
}

TemplateTypedef<int>::type i; // i is an int.
```

Esto se usaba a menudo con tipos con múltiples parámetros de plantilla, para proporcionar un alias que define uno o más de los parámetros.

```
template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
};

OneDArray<int, 3>::type     arr1i; // arr1i is an Array<int, 3, 1>.
TwoDArray<short, 5>::type   arr2s; // arr2s is an Array<short, 5, 2>.
MonoDisplayLine<char>::type arr3c; // arr3c is an Array<char, 80, 1>.
```

Miembros de la clase estatica

Una clase también puede tener miembros `static` , que pueden ser variables o funciones. Se considera que estos están dentro del alcance de la clase, pero no se tratan como miembros normales; tienen una duración de almacenamiento estático (existen desde el inicio del programa hasta el final), no están vinculados a una instancia particular de la clase y solo existe una copia

para toda la clase.

```
class Example {
    static int num_instances;      // Static data member (static member variable).
    int i;                         // Non-static member variable.

  public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func();      // Static member function.

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int         Example::num_instances;
std::string Example::static_str = "Hello.";

// ...

Example one, two, three;
// Each Example has its own "i", such that:
//  (&one.i != &two.i)
//  (&one.i != &three.i)
//  (&two.i != &three.i).
// All three Examples share "num_instances", such that:
//  (&one.num_instances == &two.num_instances)
//  (&one.num_instances == &three.num_instances)
//  (&two.num_instances == &three.num_instances)
```

Las variables miembro estáticas no se consideran definidas dentro de la clase, solo se declaran, y por lo tanto tienen su definición fuera de la definición de la clase; el programador puede, pero no es obligatorio, inicializar variables estáticas en su definición. Al definir las variables miembro, se omite la palabra clave `static` .

```
class Example {
    static int num_instances;              // Declaration.

  public:
    static std::string static_str;         // Declaration.

    // ...
};

int         Example::num_instances;         // Definition.  Zero-initialised.
std::string Example::static_str = "Hello."; // Definition.
```

Debido a esto, las variables estáticas pueden ser tipos incompletos (aparte del `void`), siempre que se definan más adelante como un tipo completo.

```
struct ForwardDeclared;

class ExIncomplete {
    static ForwardDeclared fd;
    static ExIncomplete    i_contain_myself;
    static int             an_array[];
};
```

```
struct ForwardDeclared {};

ForwardDeclared ExIncomplete::fd;
ExIncomplete    ExIncomplete::i_contain_myself;
int             ExIncomplete::an_array[5];
```

Las funciones miembro estáticas se pueden definir dentro o fuera de la definición de clase, como ocurre con las funciones miembro normales. Al igual que con las variables miembro estáticas, la palabra clave `static` se omite al definir funciones miembro estáticas fuera de la definición de clase.

```
// For Example above, either...
class Example {
    // ...

  public:
    static int static_func() { return num_instances; }

    // ...

    void set_str(const std::string& str) { static_str = str; }
};

// Or...

class Example { /* ... */ };

int  Example::static_func() { return num_instances; }
void Example::set_str(const std::string& str) { static_str = str; }
```

Si una variable miembro estática se declara `const` pero no es `volatile` , y es de tipo integral o de enumeración, se puede inicializar en declaración, dentro de la definición de clase.

```
enum E { VAL = 5 };

struct ExConst {
    const static int ci = 5;              // Good.
    static const E ce = VAL;              // Good.
    const static double cd = 5;           // Error.
    static const volatile int cvi = 5;    // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};

const double ExConst::good_cd = 5;        // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

C ++ 11

A partir de C ++ 11, las variables miembro estáticas de tipos `LiteralType` (tipos que pueden construirse en tiempo de compilación, de acuerdo con `constexpr` reglas `constexpr`) también pueden declararse como `constexpr` ; si es así, deben inicializarse dentro de la definición de clase.

```
struct ExConstexpr {
    constexpr static int ci = 5;                       // Good.
    static constexpr double cd = 5;                    // Good.
    constexpr static int carr[] = { 1, 1, 2 };         // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;                       // Error.
};

constexpr int ExConstexpr::bad_ci = 5;                 // Still an error.
```

Si una variable de miembro estática `const` o `constexpr` se *usa de* forma *estándar* (de manera informal, si tiene su dirección tomada o está asignada a una referencia), todavía debe tener una definición separada, fuera de la definición de clase. Esta definición no puede contener un inicializador.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

Como los miembros estáticos no están vinculados a una instancia determinada, se puede acceder a ellos utilizando el operador de alcance, `::` .

```
std::string str = Example::static_str;
```

También se puede acceder a ellos como si fueran miembros normales, no estáticos. Esto es de importancia histórica, pero se usa con menos frecuencia que el operador de alcance para evitar confusiones sobre si un miembro es estático o no estático.

```
Example ex;
std::string rts = ex.static_str;
```

Los miembros de la clase pueden acceder a miembros estáticos sin calificar su alcance, al igual que con los miembros de clase no estáticos.

```
class ExTwo {
    static int num_instances;
    int my_num;

  public:
    ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

No pueden ser `mutable` , ni deberían serlo; como no están vinculados a ninguna instancia dada, si una instancia es o no const no afecta a los miembros estáticos.

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

    ExDontNeedMutable() : immuta(-5), muta(-5) {}
};
int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5;   // Good.  Mutable fields of const objects can be written.
dnm.i = 5;      // Good.  Static members can be written regardless of an instance's const-
ness.
```

Los miembros estáticos respetan los modificadores de acceso, al igual que los miembros no estáticos.

```
class ExAccess {
    static int prv_int;

  protected:
    static int pro_int;

  public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
int x3 = ExAccess::pub_int; // Good.
```

Como no están vinculados a una instancia determinada, las funciones miembro estáticas no tienen `this` puntero; Debido a esto, no pueden acceder a las variables miembro no estáticas a menos que se pase una instancia.

```
class ExInstanceRequired {
    int i;

  public:
    ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }                          // Error.
```

```
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};
```

Debido a que no tienen un puntero de `this` , sus direcciones no se pueden almacenar en funciones de punteros a miembros y, en cambio, se almacenan en punteros a funciones normales.

```
struct ExPointer {
           void nsfunc() {}
    static void  sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
    f_ptr p_sf = &ExPointer::sfunc; // Good.
```

Debido a no tener un `this` puntero, sino que también no pueden ser `const` o `volatile` , ni pueden tener Ref-calificadores. Tampoco pueden ser virtuales.

```
struct ExCVQualifiersAndVirtual {
    static void   func()                {} // Good.
    static void  cfunc() const          {} // Error.
    static void  vfunc() volatile       {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void  rfunc() &              {} // Error.
    static void rvfunc() &&             {} // Error.

    virtual static void vsfunc()        {} // Error.
    static virtual void svfunc()        {} // Error.
};
```

Como no están vinculados a una instancia determinada, las variables miembro estáticas se tratan efectivamente como variables globales especiales; se crean cuando se inicia el programa y se destruyen cuando se sale, independientemente de si existe alguna instancia de la clase. Solo existe una copia de cada variable miembro estática (a menos que la variable se declare `thread_local` (C ++ 11 o posterior), en cuyo caso hay una copia por subproceso).

Las variables miembro estáticas tienen el mismo enlace que la clase, ya sea que la clase tenga enlace externo o interno. Las clases locales y las clases sin nombre no tienen permitido tener miembros estáticos.

Funciones miembro no estáticas

Una clase puede tener funciones miembro no estáticas , que operan en instancias individuales de la clase.

```
class CL {
  public:
    void member_function() {}
};
```

Estas funciones se llaman en una instancia de la clase, como así:

```
CL instance;
instance.member_function();
```

Se pueden definir dentro o fuera de la definición de clase; si se definen fuera, se especifican como dentro del alcance de la clase.

```
struct ST {
    void  defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

Pueden ser calificados para CV y / o ref , lo que afecta la forma en que ven la instancia a la que se les solicita; la función considerará que la instancia tiene el calificador (es) cv especificado, si corresponde. La versión que se llame se basará en los calificadores cv de la instancia. Si no hay una versión con los mismos calificadores de CV que la instancia, se llamará una versión más calificada de CV, si está disponible.

```
struct CVQualifiers {
    void func()                   {} // 1: Instance is non-cv-qualified.
    void func() const             {} // 2: Instance is const.

    void cv_only() const volatile {}
};

CVQualifiers       non_cv_instance;
const CVQualifiers      c_instance;

non_cv_instance.func(); // Calls #1.
c_instance.func();      // Calls #2.

non_cv_instance.cv_only(); // Calls const volatile version.
c_instance.cv_only();      // Calls const volatile version.
```

C ++ 11

Los ref-qualifiers de la función miembro indican si la función se debe llamar o no en instancias de rvalue, y usan la misma sintaxis que la función cv-qualifiers.

```
struct RefQualifiers {
    void func() &  {} // 1: Called on normal instances.
    void func() && {} // 2: Called on rvalue (temporary) instances.
};

RefQualifiers rf;
rf.func();              // Calls #1.
RefQualifiers{}.func(); // Calls #2.
```

CV-calificadores y ref-calificadores también se pueden combinar si es necesario.

```
struct BothCVAndRef {
```

```
    void func() const& {} // Called on normal instances.  Sees instance as const.
    void func() &&     {} // Called on temporary instances.
};
```

También pueden ser virtuales ; esto es fundamental para el polimorfismo, y permite que una (s) clase (s) infantil (es) proporcione la misma interfaz que la clase primaria, al tiempo que proporciona su propia funcionalidad.

```
struct Base {
    virtual void func() {}
};
struct Derived {
    virtual void func() {}
};

Base* bp = new Base;
Base* dp = new Derived;
bp.func(); // Calls Base::func().
dp.func(); // Calls Derived::func().
```

Para más información, ver aquí .

Estructura / clase sin nombre

Se permite la *`struct`* sin nombre (el tipo no tiene nombre)

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

o

```
struct Circle
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

y después

```
Circle circle;
circle.center.x = 42.f;
```

pero NO *`struct`* *anónima* (tipo sin nombre y objeto sin nombre)

```
struct InvalidCircle
{
    struct /* No name */ {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

Nota: algunos compiladores permiten `struct` *anónimas* como *extensión* .

C ++ 11

- *lamdba* puede ser visto como una `struct` especial *sin nombre* .
- `decltype` permite recuperar el tipo de `struct` *sin nombre* :

  ```
  decltype(circle.point) otherPoint;
  ```

- *La* instancia de `struct` *sin nombre* puede ser un parámetro del método de plantilla:

  ```
  void print_square_coordinates()
  {
      const struct {float x; float y;} points[] = {
          {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
      };

      // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
      for (const auto& point : points) {
          std::cout << "{" << point.x << ", " << point.y << "}\n";
      }

      decltype(points[0]) topRightCorner{1, 1};
      auto it = std::find(points, points + 4, topRightCorner);
      std::cout << "top right corner is the "
                << 1 + std::distance(points, it) << "th\n";
  }
  ```

Lea Clases / Estructuras en línea: https://riptutorial.com/es/cplusplus/topic/508/clases---estructuras

Capítulo 18: Clasificación

Observaciones

La familia de funciones `std::sort` se encuentra en la biblioteca de `algorithm` .

Examples

Clasificación de contenedores de secuencia con orden específico

Si los valores en un contenedor ya tienen ciertos operadores sobrecargados, `std::sort` puede usarse con funtores especializados para ordenar en orden ascendente o descendente:

C ++ 11

```
#include <vector>
#include <algorithm>
#include <functional>

std::vector<int> v = {5,1,2,4,3};

//sort in ascending order (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// Or just:
std::sort(v.begin(), v.end());

//sort in descending order (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

//Or just:
std::sort(v.rbegin(), v.rend());
```

C ++ 14

En C ++ 14, no necesitamos proporcionar el argumento de plantilla para los objetos de la función de comparación y, en su lugar, dejar que el objeto deduzca en función de lo que se pasa:

```
std::sort(v.begin(), v.end(), std::less<>());     // ascending order
std::sort(v.begin(), v.end(), std::greater<>());  // descending order
```

Clasificación de contenedores de secuencia por sobrecargado menos operador

Si no se pasa ninguna función de ordenación, `std::sort` ordenará los elementos llamando al `operator<` en pares de elementos, que deben devolver un tipo convertible contextualmente a `bool` (o simplemente `bool`). Los tipos básicos (enteros, flotadores, punteros, etc.) ya se han construido en los operadores de comparación.

Podemos sobrecargar a este operador para que la llamada de `sort` predeterminada funcione en tipos definidos por el usuario.

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    // Use variable to provide total order operator less
    //`this` always represents the left-hand side of the compare.
    bool operator<(const Base &b) const {
        return this->variable < b.variable;
    }

    int variable;
};

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using operator<(const Base &b) function
    std::sort(vector.begin(), vector.end());
    std::sort(deque.begin(), deque.end());
    // List must be sorted differently due to its design
    list.sort();

    return 0;
}
```

Clasificación de contenedores de secuencia utilizando la función de comparación

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using comparing function
    std::sort(vector.begin(), vector.end(), compare);
    std::sort(deque.begin(), deque.end(), compare);
    list.sort(compare);

    return 0;
}
```

Ordenando los contenedores de secuencias usando expresiones lambda (C ++ 11)

C ++ 11

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>
#include <array>
```

```
#include <forward_list>

// Include sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};


int main() {
    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // We're using C++11, so let's use initializer lists to insert items.
    std::vector <Base> vector = {a, b};
    std::deque <Base> deque = {a, b};
    std::list <Base> list = {a, b};
    std::array <Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // We can sort data using an inline lambda expression
    std::sort(std::begin(vector), std::end(vector),
      [](const Base &a, const Base &b) { return a.variable < b.variable;});

    // We can also pass a lambda object as the comparator
    // and reuse the lambda multiple times
    auto compare = [](const Base &a, const Base &b) {
                     return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}
```

Clasificación y secuenciación de contenedores.

`std::sort` , que se encuentra en el `algorithm` encabezado de biblioteca estándar, es un algoritmo de biblioteca estándar para ordenar un rango de valores, definido por un par de iteradores. `std::sort` toma como último parámetro un funtor utilizado para comparar dos valores; Así es como determina el orden. Tenga en cuenta que `std::sort` no es estable .

La función de comparación *debe* imponer un ordenamiento estricto y débil en los elementos. Una simple comparación menor que (o mayor que) será suficiente.

Un contenedor con iteradores de acceso aleatorio se puede ordenar utilizando el algoritmo `std::sort` :

C ++ 11

```
#include <vector>
#include <algorithm>

std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort` requiere que sus iteradores sean iteradores de acceso aleatorio. Los contenedores de secuencias `std::list` y `std::forward_list` (que requieren C ++ 11) no proporcionan iteradores de acceso aleatorio, por lo que no pueden usarse con `std::sort` . Sin embargo, tienen funciones de miembro de `sort` que implementan un algoritmo de clasificación que funciona con sus propios tipos de iteradores.

C ++ 11

```
#include <list>
#include <algorithm>

std::list<int> MyList = {3, 1, 2}

//Default comparison of <
//Whole list only.
MyList.sort();
```

Sus funciones de `sort` miembros siempre ordenan la lista completa, por lo que no pueden ordenar un sub-rango de elementos. Sin embargo, dado que `list` y `forward_list` tienen operaciones de empalme rápidas, puede extraer los elementos que se ordenarán de la lista, ordenarlos y luego volver a guardarlos donde fueron más eficientemente así:

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //extract and sort half-open sub range denoted by start and end iterator
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //re-insert range at the point we extracted it from
    list.splice(end, tmp);
}
```

clasificación con std :: map (ascendente y descendente)

Este ejemplo ordena los elementos en orden **ascendente** de una **clave** utilizando un mapa. Puede usar cualquier tipo, incluida la clase, en lugar de `std::string` , en el siguiente ejemplo.

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // Sort the names of the planets according to their size
    sorted_map.insert(std::make_pair(0.3829, "Mercury"));
```

```
    sorted_map.insert(std::make_pair(0.9499, "Venus"));
    sorted_map.insert(std::make_pair(1,      "Earth"));
    sorted_map.insert(std::make_pair(0.532,  "Mars"));
    sorted_map.insert(std::make_pair(10.97,  "Jupiter"));
    sorted_map.insert(std::make_pair(9.14,   "Saturn"));
    sorted_map.insert(std::make_pair(3.981,  "Uranus"));
    sorted_map.insert(std::make_pair(3.865,  "Neptune"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " of Earth's radius)" << '\n';
    }
}
```

Salida:

```
Mercury (0.3829 of Earth's radius)
Mars (0.532 of Earth's radius)
Venus (0.9499 of Earth's radius)
Earth (1 of Earth's radius)
Neptune (3.865 of Earth's radius)
Uranus (3.981 of Earth's radius)
Saturn (9.14 of Earth's radius)
Jupiter (10.97 of Earth's radius)
```

Si son posibles entradas con claves iguales, use el `multimap` lugar del `map` (como en el siguiente ejemplo).

Para ordenar los elementos de forma **descendente** , declare el mapa con un funtor de comparación adecuado (`std::greater<>`):

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // Sort the names of animals in descending order of the number of legs
    sorted_map.insert(std::make_pair(6,   "bug"));
    sorted_map.insert(std::make_pair(4,   "cat"));
    sorted_map.insert(std::make_pair(100, "centipede"));
    sorted_map.insert(std::make_pair(2,   "chicken"));
    sorted_map.insert(std::make_pair(0,   "fish"));
    sorted_map.insert(std::make_pair(4,   "horse"));
    sorted_map.insert(std::make_pair(8,   "spider"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (has " << entry.first << " legs)" << '\n';
    }
}
```

Salida

```
centipede (has 100 legs)
```

```
spider (has 8 legs)
bug (has 6 legs)
cat (has 4 legs)
horse (has 4 legs)
chicken (has 2 legs)
fish (has 0 legs)
```

Clasificación de matrices incorporadas

El algoritmo de `sort` ordena una secuencia definida por dos iteradores. Esto es suficiente para ordenar una matriz integrada (también conocida como c-style).

C ++ 11

```
int arr1[] = {36, 24, 42, 60, 59};

// sort numbers in ascending order
sort(std::begin(arr1), std::end(arr1));

// sort numbers in descending order
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

Antes de C ++ 11, el final de la matriz debía "calcularse" utilizando el tamaño de la matriz:

C ++ 11

```
// Use a hard-coded number for array size
sort(arr1, arr1 + 5);

// Alternatively, use an expression
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

Lea Clasificación en línea: https://riptutorial.com/es/cplusplus/topic/1675/clasificacion

Capítulo 19: Comparaciones lado a lado de ejemplos clásicos de C ++ resueltos a través de C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17

Examples

Buceando a través de un contenedor

En C ++, el bucle a través de un contenedor de secuencia `c` se puede hacer usando índices de la siguiente manera:

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

Aunque simples, tales escritos están sujetos a errores semánticos comunes, como un operador de comparación incorrecto o una variable de indexación incorrecta:

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
                     ^~~~~~~~~~~~~~^
```

El bucle también se puede lograr para todos los contenedores que utilizan iteradores, con inconvenientes similares:

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

C ++ 11 introdujo rango basado en bucles y palabras clave `auto` , permitiendo que el código se convierta en:

```
for(auto& x : c) x = 0;
```

Aquí los únicos parámetros son el contenedor `c` y una variable `x` para mantener el valor actual. Esto evita los errores semánticos apuntados previamente.

De acuerdo con el estándar C ++ 11, la implementación subyacente es equivalente a:

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

En dicha implementación, la expresión `auto begin = c.begin(), end = c.end();` las fuerzas `begin` y `end` para ser del mismo tipo, mientras que el `end` nunca se incrementa, ni se elimina la referencia. Por lo tanto, el bucle for basado en rango solo funciona para contenedores definidos por un par iterador / iterador. El estándar C ++ 17 relaja esta restricción al cambiar la implementación a:

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

Aquí se permite que el `begin` y el `end` sean de diferentes tipos, siempre que se puedan comparar por desigualdad. Esto permite recorrer más contenedores, por ejemplo, un contenedor definido por un par iterador / centinela.

Lea Comparaciones lado a lado de ejemplos clásicos de C ++ resueltos a través de C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17 en línea: https://riptutorial.com/es/cplusplus/topic/7134/comparaciones-lado-a-lado-de-ejemplos-clasicos-de-c-plusplus-resueltos-a-traves-de-c-plusplus-vs-c-plusplus-11-vs-c-plusplus-14-vs-c-plusplus-17

Capítulo 20: Compilando y construyendo

Introducción

Los programas escritos en C ++ deben compilarse antes de que puedan ejecutarse. Hay una gran variedad de compiladores disponibles dependiendo de su sistema operativo.

Observaciones

La mayoría de los sistemas operativos se envían sin un compilador, y deben instalarse más adelante. Algunas opciones comunes de compiladores son:

- GCC, la colección de compiladores GNU g ++
- clang: una interfaz familiar de lenguaje C para LLVM clang ++
- MSVC, Microsoft Visual C ++ (incluido en Visual Studio) visual-c ++
- C ++ Builder, Embarcadero C ++ Builder (incluido en RAD Studio) c ++ builder

Por favor, consulte el manual del compilador apropiado, sobre cómo compilar un programa C ++.

Otra opción para usar un compilador específico con su propio sistema de compilación específico, es posible permitir que los sistemas de compilación genéricos configuren el proyecto para un compilador específico o para el instalado por defecto.

Examples

Compilando con GCC

Asumiendo un único archivo fuente llamado `main.cpp` , el comando para compilar y vincular un ejecutable no optimizado es el siguiente (Compilar sin optimización es útil para el desarrollo inicial y la depuración, aunque oficialmente se recomienda `-Og` para las versiones más recientes de GCC).

```
g++ -o app -Wall main.cpp -O0
```

Para producir un ejecutable optimizado para su uso en la producción, utilizar una de las `-O` opciones (véase: `-O1` , `-O2` , `-O3` , `-Os` , `-Ofast`):

```
g++ -o app -Wall -O2 main.cpp
```

Si se omite la opción -O, se utiliza -O0, que significa que no hay optimizaciones, como valor predeterminado (la especificación de -O sin un número se resuelve en -O1).

Alternativamente, use las marcas de optimización de los grupos `O` (o más optimizaciones experimentales) directamente. El siguiente ejemplo construye con la optimización de `-O2` , más un indicador del nivel de optimización de `-O3` :

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

Para producir un ejecutable optimizado específico de la plataforma (para su uso en producción en la máquina con la misma arquitectura), use:

```
g++ -o app -Wall -O2 -march=native main.cpp
```

Cualquiera de los anteriores producirá un archivo binario que puede ejecutarse con `.\app.exe` en Windows y `./app` en Linux, Mac OS, etc.

La bandera `-o` también se puede omitir. En este caso, GCC creará una salida ejecutable predeterminada `a.exe` en Windows y `a.out` en sistemas similares a Unix. Para compilar un archivo sin vincularlo, use la opción `-c` :

```
g++ -o file.o -Wall -c file.cpp
```

Esto produce un archivo de objeto llamado `file.o` que luego puede vincularse con otros archivos para producir un binario:

```
g++ -o app file.o otherfile.o
```

Puede encontrar más información sobre las opciones de optimización en gcc.gnu.org . De particular interés son: `-Og` (optimización con énfasis en la experiencia de depuración, recomendada para el ciclo estándar de edición-compilación-depuración) y `-Ofast` (todas las optimizaciones, incluidas las que no tienen en cuenta el estricto cumplimiento de estándares).

La bandera `-Wall` habilita advertencias para muchos errores comunes y siempre debe usarse. Para mejorar la calidad del código, a menudo también se recomienda utilizar `-Wextra` y otros indicadores de advertencia que no están habilitados automáticamente por `-Wall` y `-Wextra` .

Si el código espera un estándar de C ++ específico, especifique qué estándar usar incluyendo la `-std=` . Los valores admitidos corresponden al año de finalización de cada versión del estándar ISO C ++. A partir de GCC 6.1.0, los valores válidos para el indicador `std=` son `c++98` / `c++03` , `c++11` , `c++14` y `c++17` / `c++1z` . Los valores separados por una barra diagonal son equivalentes.

```
g++ -std=c++11 <file>
```

GCC incluye algunas extensiones específicas del compilador que están deshabilitadas cuando entran en conflicto con un estándar especificado por el `-std=` . Para compilar con todas las extensiones habilitadas, se puede utilizar el valor `gnu++XX` , donde `XX` es cualquiera de los años utilizados por los valores de `c++` enumerados anteriormente.

El estándar predeterminado se utilizará si no se especifica ninguno. Para las versiones de GCC anteriores a 6.1.0, el valor predeterminado es `-std=gnu++03` ; en GCC 6.1.0 y superior, el valor predeterminado es `-std=gnu++14` .

Tenga en cuenta que debido a errores en GCC, el indicador `-pthread` debe estar presente en la

compilación y enlace para que GCC admita la funcionalidad de subprocesamiento estándar C ++ introducida con C ++ 11, como `std::thread` y `std::wait_for` . Omitirlo cuando se usan funciones de subprocesamiento puede generar advertencias pero resultados no válidos en algunas plataformas.

Vinculación con bibliotecas:

Use la opción `-l` para pasar el nombre de la biblioteca:

```
g++ main.cpp -lpcre2-8
#pcre2-8 is the PCRE2 library for 8bit code units (UTF-8)
```

Si la biblioteca no está en la ruta de la biblioteca estándar, agregue la ruta con la opción `-L` :

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

Se pueden vincular varias bibliotecas entre sí:

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

Si una biblioteca depende de otra, ponga la biblioteca dependiente **antes de** la biblioteca independiente:

```
g++ main.cpp -lchild-lib -lbase-lib
```

O deje que el enlazador determine el pedido a través de `--start-group` y `--end-group` (nota: esto tiene un costo de rendimiento significativo):

```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Compilando con Visual C ++ (Línea de Comando)

Para los programadores que vienen de GCC o Clang a Visual Studio, o los programadores que se sienten más cómodos con la línea de comandos en general, puede usar el compilador de Visual C ++ desde la línea de comandos así como el IDE.

Si desea compilar su código desde la línea de comandos en Visual Studio, primero debe configurar el entorno de línea de comandos. Esto se puede hacer abriendo la línea de `Visual Studio Command Prompt` / `Developer Command Prompt` / `x86 Native Tools Command Prompt` / `x64 Native Tools Command Prompt` o similar (según lo provisto por su versión de Visual Studio), o en la línea de comandos, navegando a el subdirectorio `VC` directorio de instalación del compilador (normalmente `\Program Files (x86)\Microsoft Visual Studio x\VC` , donde `x` es el número de versión (como `10.0` para 2010 o `14.0` para 2015) y ejecuta el archivo por lotes `VCVARSALL` con una parámetro de línea de comando especificado aquí .

Tenga en cuenta que a diferencia de GCC, Visual Studio no proporciona un front-end para el

vinculador (`link.exe`) a través del compilador (`cl.exe`), sino que proporciona el vinculador como un programa separado, al que el compilador llama cuando sale. `cl.exe` y `link.exe` se pueden usar por separado con diferentes archivos y opciones, o se puede indicar a `cl` que pase archivos y opciones para `link` si ambas tareas se realizan juntas. Cualquier opción de enlace especificada para `cl` se convertirá en opciones para `link` , y cualquier archivo que no sea procesado por `cl` pasará directamente al `link` . Como esta es principalmente una guía simple para compilar con la línea de comandos de Visual Studio, los argumentos para el `link` no se describirán en este momento; Si necesita una lista, vea aquí .

Tenga en cuenta que los argumentos para `cl` distinguen entre mayúsculas y minúsculas, mientras que los argumentos para `link` no lo son.

[Tenga en cuenta que algunos de los siguientes ejemplos utilizan la variable "directorio actual" del shell de Windows, `%cd%` , al especificar nombres de ruta absolutos. Para cualquier persona que no esté familiarizada con esta variable, se expande al directorio de trabajo actual. Desde la línea de comandos, será el directorio en el que estaba cuando ejecutó `cl` , y se especifica en el símbolo del sistema de manera predeterminada (si su símbolo del sistema es `C:\src>` , por ejemplo, entonces `%cd%` es `C:\src\`).]

Suponiendo que un solo archivo de origen denominado `main.cpp` en la carpeta actual, el comando para compilar y vincular un ejecutable no optimizado (útil para el desarrollo inicial y la depuración) es (use uno de los siguientes):

```
cl main.cpp
// Generates object file "main.obj".
// Performs linking with "main.obj".
// Generates executable "main.exe".

cl /Od main.cpp
// Same as above.
// "/Od" is the "Optimisation: disabled" option, and is the default when no /O is specified.
```

Suponiendo un archivo fuente adicional "niam.cpp" en el mismo directorio, use lo siguiente:

```
cl main.cpp niam.cpp
// Generates object files "main.obj" and "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

También puedes usar comodines, como es de esperar:

```
cl main.cpp src\*.cpp
// Generates object file "main.obj", plus one object file for each ".cpp" file in folder
//  "%cd%\src".
// Performs linking with "main.obj", and every additional object file generated.
// All object files will be in the current folder.
// Generates executable "main.exe".
```

Para renombrar o reubicar el ejecutable, use uno de los siguientes:

```
cl /o name main.cpp
// Generates executable named "name.exe".

cl /o folder\ main.cpp
// Generates executable named "main.exe", in folder "%cd%\folder".

cl /o folder\name main.cpp
// Generates executable named "name.exe", in folder "%cd%\folder".

cl /Fename main.cpp
// Same as "/o name".

cl /Fefolder\ main.cpp
// Same as "/o folder\".

cl /Fefolder\name main.cpp
// Same as "/o folder\name".
```

Tanto `/o` como `/Fe` pasan su parámetro (llamémoslo `o-param`) para `link` como `/OUT:o-param` , agregando la extensión apropiada (generalmente `.exe` o `.dll`) para "nombrar" `o-param` según sea necesario. Mientras tanto `/o` y `/Fe` son, en mi opinión, idénticas en funcionalidad, esta última es la preferida para Visual Studio. `/o` está marcado como obsoleto y parece que se proporciona principalmente para programadores más familiarizados con GCC o Clang.

Tenga en cuenta que si bien el espacio entre `/o` y la carpeta y / o el nombre especificados es opcional, no *puede* haber un espacio entre `/Fe` y la carpeta y / o el nombre especificados.

De manera similar, para producir un ejecutable optimizado (para uso en producción), use:

```
cl /O1 main.cpp
// Optimise for executable size.  Produces small programs, at the possible expense of slower
//  execution.

cl /O2 main.cpp
// Optimise for execution speed.  Produces fast programs, at the possible expense of larger
//  file size.

cl /GL main.cpp other.cpp
// Generates special object files used for whole-program optimisation, which allows CL to
//  take every module (translation unit) into consideration during optimisation.
// Passes the option "/LTCG" (Link-Time Code Generation) to LINK, telling it to call CL during
//  the linking phase to perform additional optimisations.  If linking is not performed at
this
//  time, the generated object files should be linked with "/LTCG".
// Can be used with other CL optimisation options.
```

Finalmente, para producir un ejecutable optimizado específico de la plataforma (para su uso en la producción en la máquina con la arquitectura especificada), elija el indicador de comando o el parámetro `VCVARSALL` para la plataforma de destino. `link` debe detectar la plataforma deseada desde los archivos de objetos; si no, use la opción `/MACHINE` para especificar explícitamente la plataforma de destino.

```
// If compiling for x64, and LINK doesn't automatically detect target platform:
```

```
cl main.cpp /link /machine:X64
```

Cualquiera de los anteriores producirá un ejecutable con el nombre especificado por `/o` o `/Fe` , o si no se proporciona ninguno, con un nombre idéntico al primer archivo de origen u objeto especificado para el compilador.

```
cl a.cpp b.cpp c.cpp
// Generates "a.exe".

cl d.obj a.cpp q.cpp
// Generates "d.exe".

cl y.lib n.cpp o.obj
// Generates "n.exe".

cl /o yo zp.obj pz.cpp
// Generates "yo.exe".
```

Para compilar un archivo (s) sin vincular, use:

```
cl /c main.cpp
// Generates object file "main.obj".
```

Esto le dice a `cl` que salga sin llamar al `link` , y produce un archivo objeto, que luego puede vincularse con otros archivos para producir un binario.

```
cl main.obj niam.cpp
// Generates object file "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".

link main.obj niam.obj
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

También hay otros parámetros de línea de comando valiosos, que sería muy útil para los usuarios saber:

```
cl /EHsc main.cpp
// "/EHsc" specifies that only standard C++ ("synchronous") exceptions will be caught,
//  and `extern "C"` functions will not throw exceptions.
// This is recommended when writing portable, platform-independent code.

cl /clr main.cpp
// "/clr" specifies that the code should be compiled to use the common language runtime,
//  the .NET Framework's virtual machine.
// Enables the use of Microsoft's C++/CLI language in addition to standard ("native") C++,
//  and creates an executable that requires .NET to run.

cl /Za main.cpp
// "/Za" specifies that Microsoft extensions should be disabled, and code should be
//  compiled strictly according to ISO C++ specifications.
// This is recommended for guaranteeing portability.
```

```
cl /Zi main.cpp
// "/Zi" generates a program database (PDB) file for use when debugging a program, without
//  affecting optimisation specifications, and passes the option "/DEBUG" to LINK.

cl /LD dll.cpp
// "/LD" tells CL to configure LINK to generate a DLL instead of an executable.
// LINK will output a DLL, in addition to an LIB and EXP file for use when linking.
// To use the DLL in other programs, pass its associated LIB to CL or LINK when compiling
those
//  programs.

cl main.cpp /link /LINKER_OPTION
// "/link" passes everything following it directly to LINK, without parsing it in any way.
// Replace "/LINKER_OPTION" with any desired LINK option(s).
```

Para cualquiera que esté más familiarizado con los sistemas * nix y / o GCC / Clang, `cl` , `link` y otras herramientas de línea de comandos de Visual Studio puede aceptar parámetros especificados con un guión (como `-c`) en lugar de una barra (como `/c`). Además, Windows reconoce una barra diagonal o una barra diagonal inversa como un separador de ruta válido, por lo que también se pueden usar las rutas estilo * nix. Esto facilita la conversión de líneas de comando del compilador simple de `g++` o `clang++` a `cl` , o viceversa, con cambios mínimos.

```
g++ -o app src/main.cpp
cl  -o app src/main.cpp
```

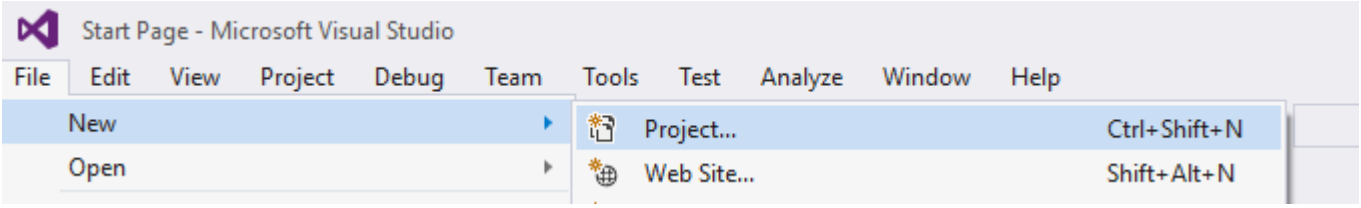
Por supuesto, cuando se transfieren líneas de comando que usan opciones más complejas de `g++` o `clang++` , debe buscar comandos equivalentes en la documentación del compilador correspondiente y / o en los sitios de recursos, pero esto hace que sea más fácil comenzar con un tiempo mínimo de aprendizaje. Nuevos compiladores.

En caso de que necesite funciones de idioma específicas para su código, se requiere una versión específica de MSVC. Desde Visual C ++ 2015 Update 3 , es posible elegir la versión del estándar para compilar a través de la bandera `/std` . Los valores posibles son `/std:c++14` y `/std:c++latest` (`/std:c++17` seguirá pronto).

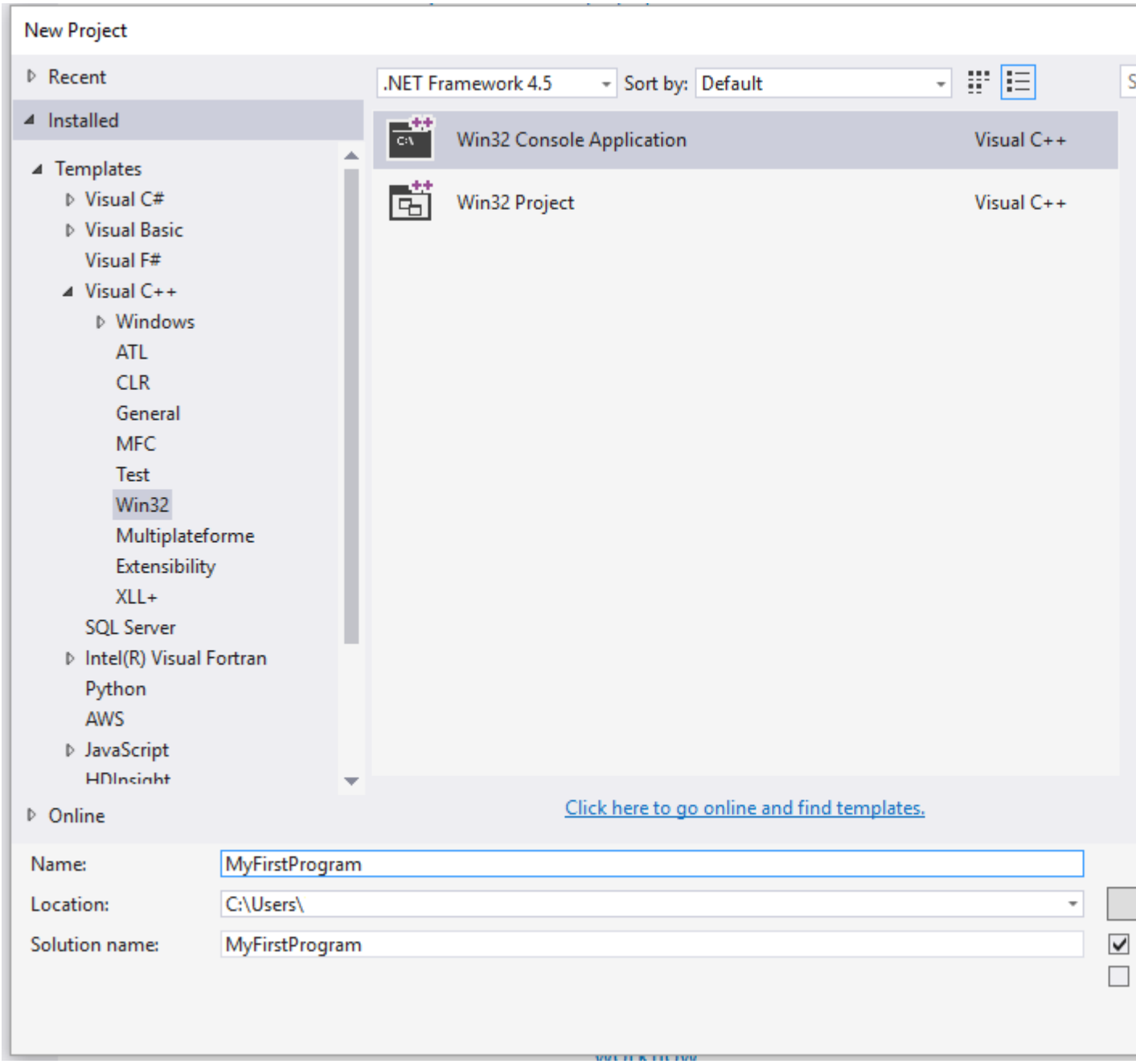
Nota: en versiones anteriores de este compilador, las características específicas estaban disponibles, sin embargo, esto se utilizaba principalmente para las vistas previas de nuevas funciones.

Compilación con Visual Studio (interfaz gráfica) - Hello World

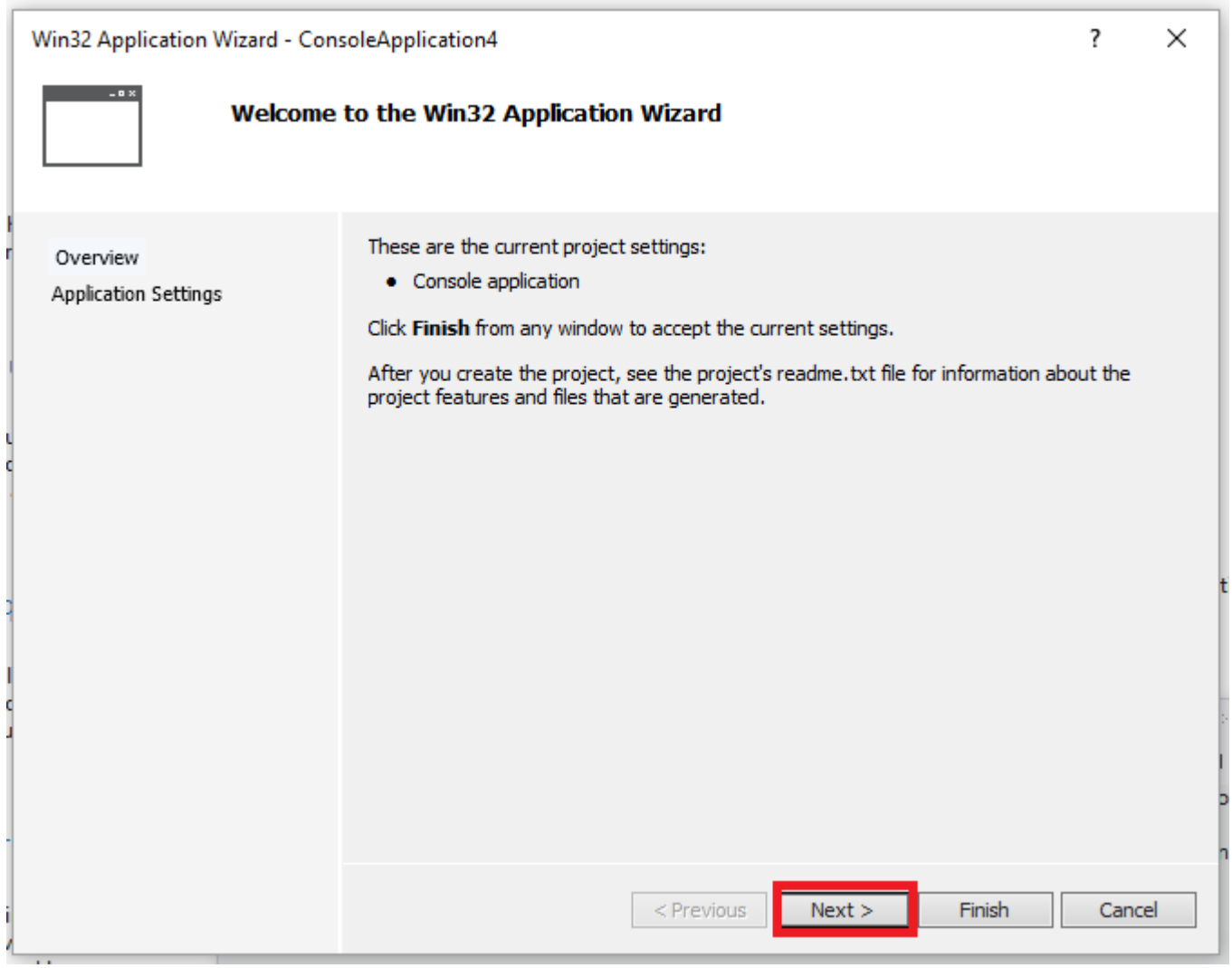
1. Descargue e instale Visual Studio Community 2015
2. Abrir la comunidad de Visual Studio
3. Haga clic en Archivo -> Nuevo -> Proyecto



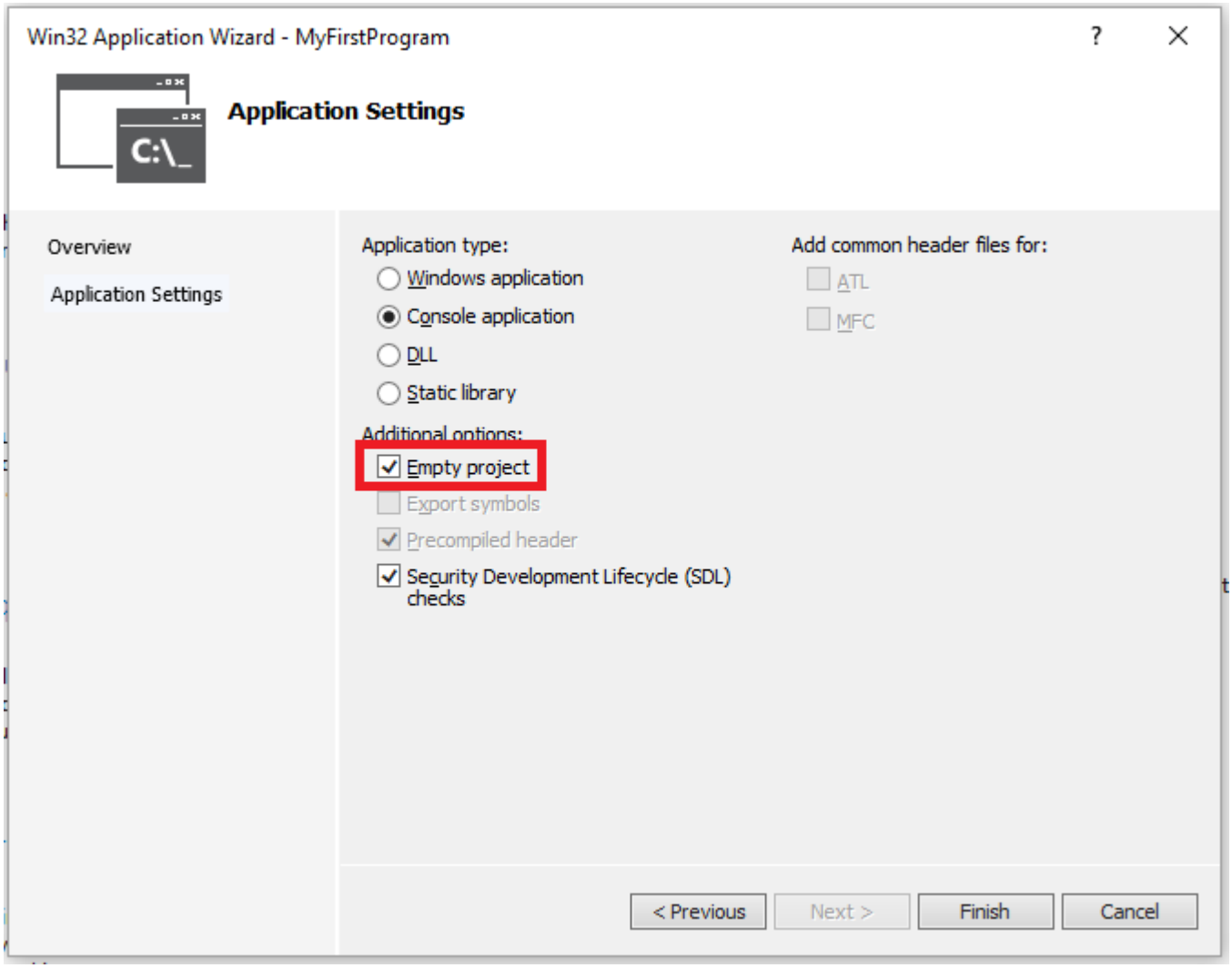
4. Haga clic en Plantillas -> Visual C ++ -> Aplicación de consola Win32 y luego nombre el proyecto **MyFirstProgram** .



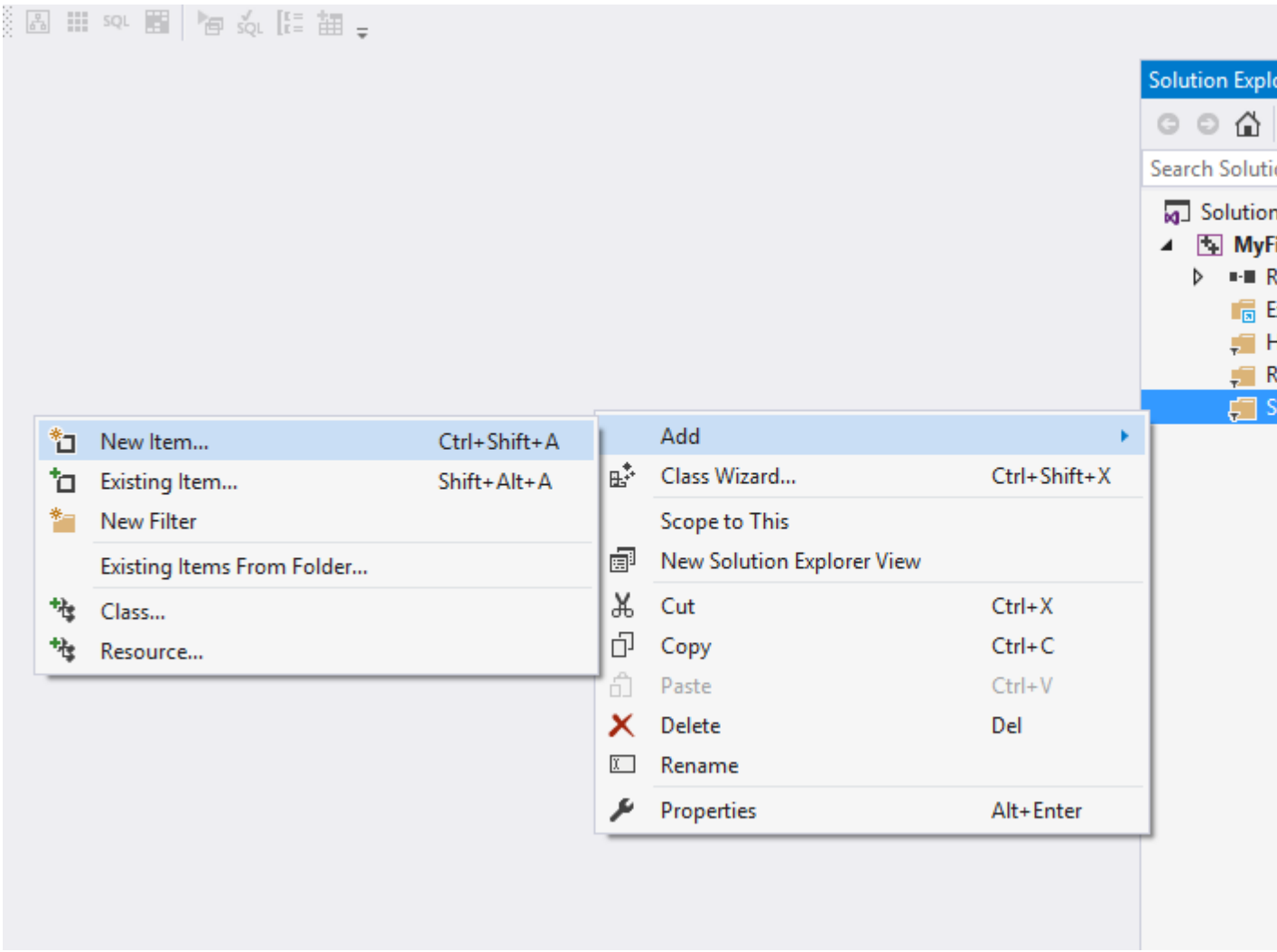
5. Haga clic en Aceptar
6. Haga clic en Siguiente en la siguiente ventana.



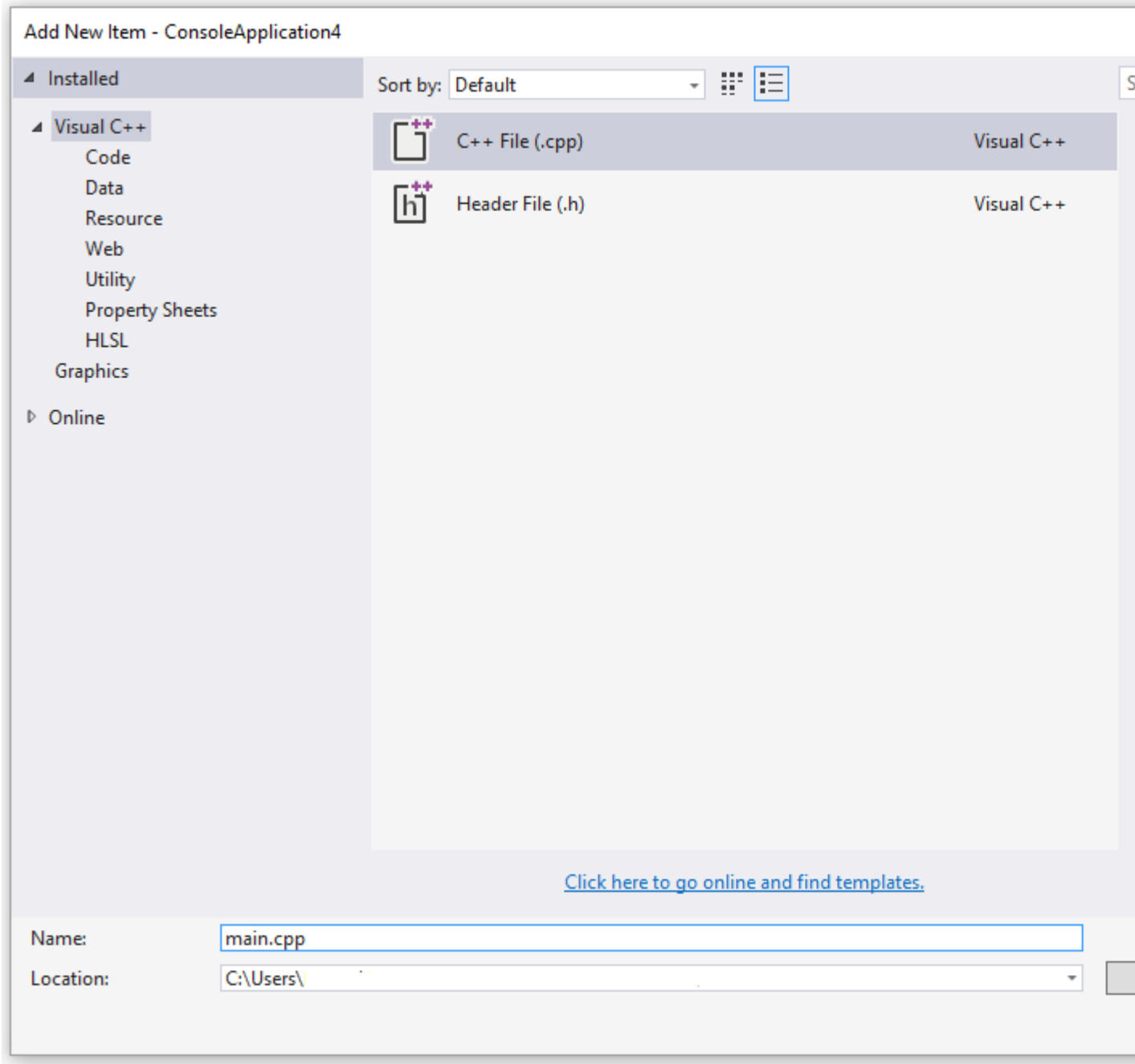
7. Marque la casilla `Empty project` y luego haga clic en Finalizar:



8. Haga clic derecho en la carpeta Archivo de origen y luego -> Agregar -> Nuevo elemento:



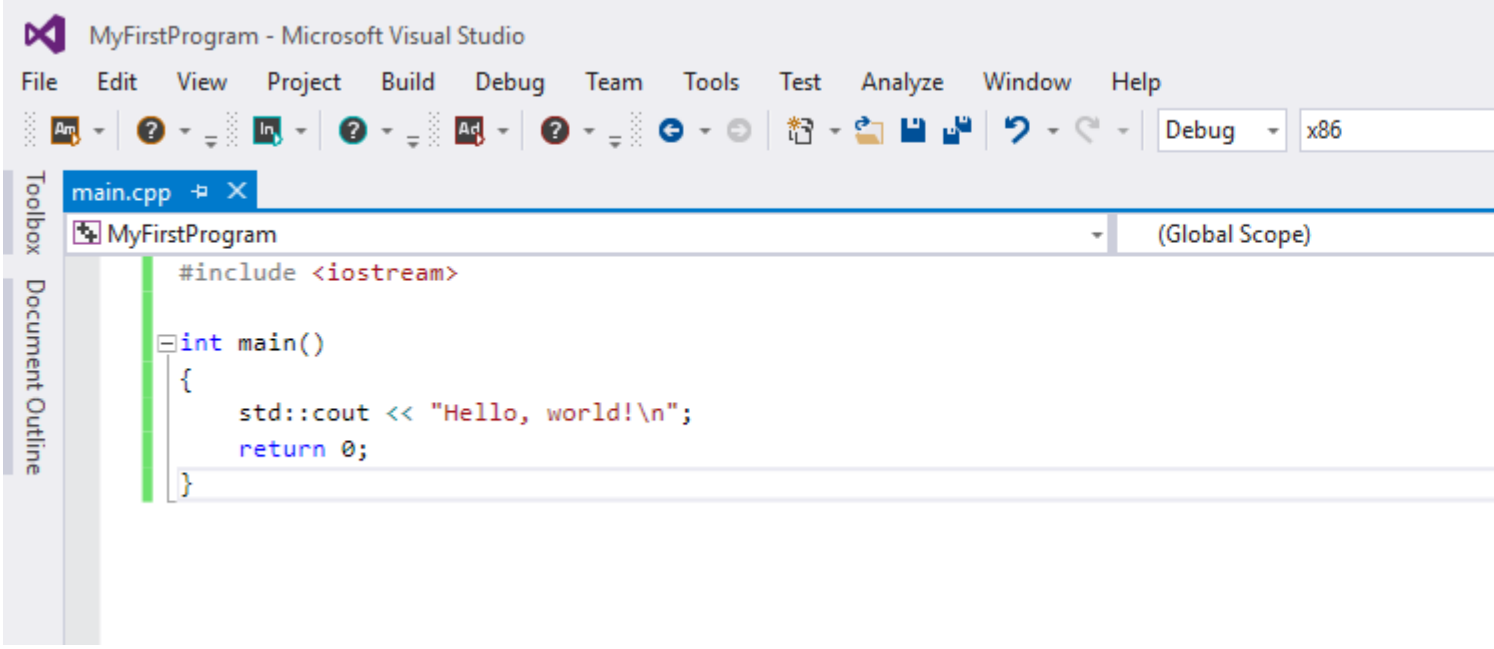
9. Seleccione Archivo C ++ y nombre el archivo main.cpp, luego haga clic en Agregar:



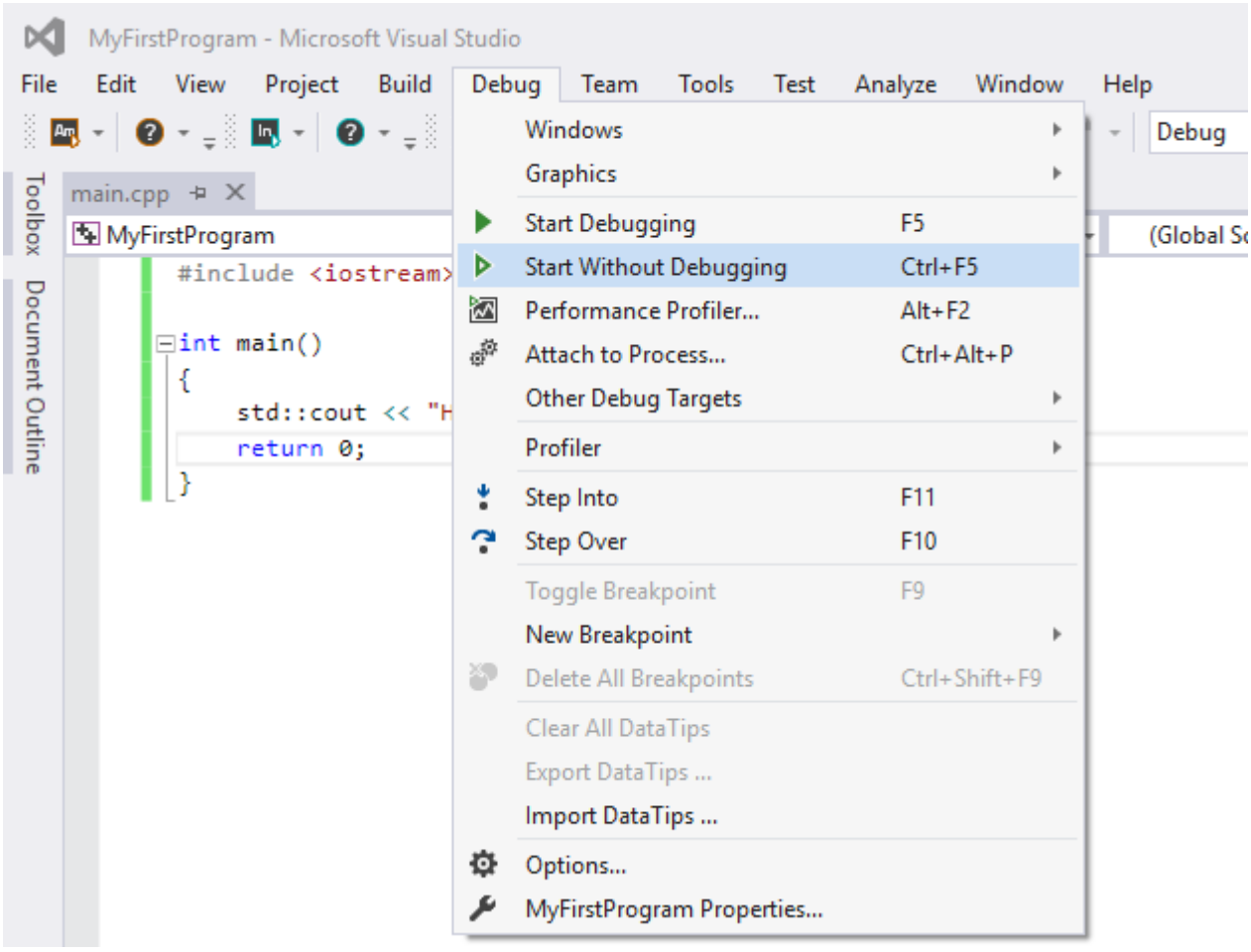
10: Copie y pegue el siguiente código en el nuevo archivo main.cpp:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

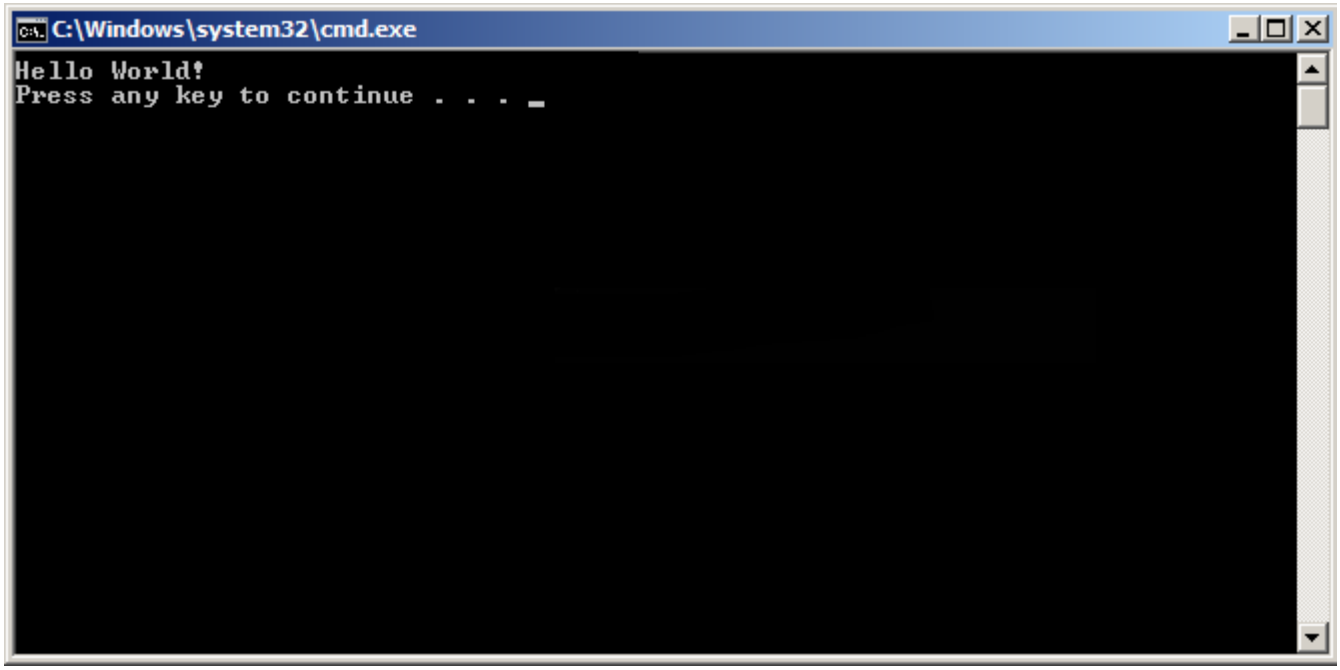
Tu entorno debe verse como:



11. Haga clic en Depurar -> Iniciar **sin** depurar (o presione ctrl + F5):



12. Hecho. Deberías obtener la siguiente salida de consola:



Compilando con Clang

Como la interfaz de Clang está diseñada para ser compatible con GCC, la mayoría de los programas que se pueden compilar a través de GCC se compilarán cuando intercambies `g++` por `clang++` en los scripts de compilación. Si no se `-std=version` , se utilizará gnu11.

Los usuarios de Windows que están acostumbrados a MSVC pueden intercambiar `cl.exe` con `clang-cl.exe` . De forma predeterminada, clang intenta ser compatible con la versión más alta de MSVC que se ha instalado.

En el caso de compilar con visual studio, clang-cl puede usarse cambiando el `Platform toolset` de `Platform toolset` la `Platform toolset` en las propiedades del proyecto.

En ambos casos, clang solo es compatible a través de su interfaz, aunque también intenta generar archivos de objetos compatibles binarios. Los usuarios de clang-cl deben tener en cuenta que la compatibilidad con MSVC aún no está completa .

Para usar clang o clang-cl, uno podría usar la instalación predeterminada en ciertas distribuciones de Linux o las que vienen con IDE (como XCode en Mac). Para otras versiones de este compilador o en plataformas que no lo tienen instalado, se puede descargar desde la página de descarga oficial .

Si está utilizando CMake para compilar su código, generalmente puede cambiar el compilador configurando las variables de entorno `CC` y `CXX` esta manera:

```
mkdir build
cd build
CC=clang CXX=clang++ cmake ..
cmake --build .
```

Véase también la introducción a Cmake .

Compiladores en linea

Varios sitios web proporcionan acceso en línea a compiladores de C ++. El conjunto de funciones del compilador en línea varía significativamente de un sitio a otro, pero generalmente permiten hacer lo siguiente:

- Pega tu código en un formulario web en el navegador.
- Seleccione algunas opciones del compilador y compile el código.
- Recoger compilador y / o salida del programa.

El comportamiento del sitio web del compilador en línea suele ser bastante restrictivo, ya que permite que cualquier persona ejecute compiladores y ejecute código arbitrario en su servidor, mientras que la ejecución de código arbitrario a distancia se considera una vulnerabilidad.

Los compiladores en línea pueden ser útiles para los siguientes propósitos:

- Ejecute un pequeño fragmento de código desde una máquina que carece de compilador de C ++ (teléfonos inteligentes, tabletas, etc.).
- Asegúrese de que el código se compile correctamente con diferentes compiladores y se ejecute de la misma manera, independientemente del compilador con el que se compiló.
- Aprender o enseñar conceptos básicos de C ++.
- Conozca las funciones modernas de C ++ (C ++ 14 y C ++ 17 en un futuro próximo) cuando el compilador C ++ actualizado no esté disponible en la máquina local.
- Detecta un error en tu compilador en comparación con un gran conjunto de otros compiladores. Compruebe si se solucionó un error del compilador en futuras versiones, que no están disponibles en su máquina.
- Resolver problemas de jueces en línea.

Para qué compiladores en línea **no se** debe utilizar:

- Desarrolle aplicaciones completas (incluso pequeñas) utilizando C ++. Por lo general, los compiladores en línea no permiten enlazar con bibliotecas de terceros o descargar artefactos de compilación.
- Realizar cálculos intensivos. Los recursos informáticos del lado del servidor son limitados, por lo que cualquier programa proporcionado por el usuario se eliminará después de unos segundos de ejecución. El tiempo de ejecución permitido suele ser suficiente para la prueba y el aprendizaje.
- Atacar el servidor del compilador o cualquier otro host de terceros en la red.

Ejemplos:

> Descargo de responsabilidad: los autores de la documentación no están afiliados a los recursos que se enumeran a continuación. Los sitios web están listados alfabéticamente.

- http://codepad.org/ Compilador en línea con código compartido. Editar código después de compilar con un código fuente de advertencia o error no funciona tan bien.
- http://coliru.stacked-crooked.com/ Compilador en línea para el que especifica la línea de

comando. Proporciona compiladores de GCC y Clang para su uso.

- http://cpp.sh/ - Compilador en línea con soporte para C ++ 14. No le permite editar la línea de comandos del compilador, pero algunas opciones están disponibles a través de los controles GUI.
- https://gcc.godbolt.org/ : proporciona una amplia lista de versiones de compilador, arquitecturas y resultados de desensamblaje. Muy útil cuando necesitas inspeccionar lo que compila tu código por diferentes compiladores. GCC, Clang, MSVC (`CL`), compilador Intel (`icc`), ELLCC y Zapcc están presentes, con uno o más de estos compiladores disponibles para ARM, ARMv8 (como ARM64), AVR de Atmel, MIPS, MIPS64, MSP430, PowerPC , x86, y x64 architecutres. Los argumentos de la línea de comando del compilador pueden ser editados.
- https://ideone.com/ : se utiliza ampliamente en la red para ilustrar el comportamiento del fragmento de código. Proporciona GCC y Clang para su uso, pero no le permite editar la línea de comandos del compilador.
- http://melpon.org/wandbox - Admite numerosas versiones de compilador Clang y GNU / GCC.
- http://onlinegdb.com/ : un IDE extremadamente minimalista que incluye un editor, un compilador (gcc) y un depurador (gdb).
- http://rextester.com/ : proporciona compiladores de Clang, GCC y Visual Studio para C y C ++ (junto con compiladores para otros idiomas), con la biblioteca Boost disponible para su uso.
- http://tutorialspoint.com/compile_cpp11_online.php : shell UNIX con todas las funciones con GCC y un explorador de proyectos fácil de usar.
- http://webcompiler.cloudapp.net/ - Compilador en línea de Visual Studio 2015, proporcionado por Microsoft como parte de RiSE4fun.

El proceso de compilación de C ++.

Cuando desarrolle un programa en C ++, el siguiente paso es compilar el programa antes de ejecutarlo. La compilación es el proceso que convierte el programa escrito en lenguaje legible por humanos como C, C ++, etc., en un código de máquina, entendido directamente por la Unidad Central de Procesamiento. Por ejemplo, si tiene un archivo de código fuente de C ++ llamado prog.cpp y ejecuta el comando de compilación,

```
g++ -Wall -ansi -o prog prog.cpp
```

Hay 4 etapas principales involucradas en la creación de un archivo ejecutable desde el archivo fuente.

1. El preprocesador de C ++ toma un archivo de código fuente de C ++ y se ocupa de los encabezados (#include), macros (#define) y otras directivas de preprocesador.

2. El archivo de código fuente expandido de C ++ producido por el preprocesador de C ++ se compila en el lenguaje ensamblador de la plataforma.

3. El código de ensamblador generado por el compilador se ensambla en el código de objeto para la plataforma.

4. El archivo de código de objeto producido por el ensamblador está vinculado entre sí con los archivos de código de objeto para cualquier función de biblioteca utilizada para producir una biblioteca o un archivo ejecutable.

Preprocesamiento

El preprocesador maneja las directivas del preprocesador, como #include y #define. Es un agnóstico de la sintaxis de C ++, por lo que debe usarse con cuidado.

Funciona en un archivo fuente de C ++ a la vez al reemplazar las directivas #include con el contenido de los archivos respectivos (que generalmente son solo declaraciones), al reemplazar las macros (# definir) y al seleccionar diferentes porciones de texto dependiendo de #if, #ifdef y #ifndef directivas.

El preprocesador trabaja en una secuencia de tokens de preprocesamiento. La sustitución de macros se define como la sustitución de tokens con otros tokens (el operador ## permite fusionar dos tokens cuando tiene sentido).

Después de todo esto, el preprocesador produce una salida única que es una secuencia de tokens resultantes de las transformaciones descritas anteriormente. También agrega algunos marcadores especiales que le dicen al compilador de dónde proviene cada línea para que pueda usarlos para producir mensajes de error razonables.

Algunos errores pueden producirse en esta etapa con un uso inteligente de las directivas #if y #error.

Al utilizar el indicador del compilador a continuación, podemos detener el proceso en la etapa de preprocesamiento.

```
g++ -E prog.cpp
```

Compilacion

El paso de compilación se realiza en cada salida del preprocesador. El compilador analiza el código fuente puro de C ++ (ahora sin directivas de preprocesador) y lo convierte en código de ensamblaje. Luego invoca el back-end subyacente (ensamblador en la cadena de herramientas) que ensambla ese código en un código de máquina produciendo un archivo binario real en algún formato (ELF, COFF, a.out, ...). Este archivo de objeto contiene el código compilado (en forma binaria) de los símbolos definidos en la entrada. Los símbolos en los archivos de objetos se denominan por nombre.

Los archivos de objetos pueden referirse a símbolos que no están definidos. Este es el caso cuando usa una declaración y no proporciona una definición para ella. Al compilador no le importa esto, y felizmente producirá el archivo de objeto siempre que el código fuente esté bien formado.

Los compiladores generalmente le permiten detener la compilación en este punto. Esto es muy útil porque con él puede compilar cada archivo de código fuente por separado. La ventaja que esto proporciona es que no necesita recompilar todo si solo cambia un solo archivo.

Los archivos de objetos producidos se pueden colocar en archivos especiales denominados bibliotecas estáticas, para luego reutilizarlos más fácilmente.

Es en esta etapa que se informan los errores "regulares" del compilador, como los errores de sintaxis o los errores de resolución de sobrecarga fallidos.

Para detener el proceso después del paso de compilación, podemos usar la opción -S:

```
g++ -Wall -ansi -S prog.cpp
```

Montaje

El ensamblador crea código objeto. En un sistema UNIX puede ver archivos con un sufijo .o (.OBJ en MSDOS) para indicar archivos de código de objeto. En esta fase, el ensamblador convierte esos archivos de objeto de código de ensamblaje en instrucciones a nivel de máquina y el archivo creado es un código de objeto reubicable. Por lo tanto, la fase de compilación genera el programa de objeto reubicable y este programa se puede utilizar en diferentes lugares sin tener que compilar nuevamente.

Para detener el proceso después del paso de ensamblaje, puede usar la opción -c:

```
g++ -Wall -ansi -c prog.cpp
```

Enlace

El enlazador es lo que produce la salida de compilación final de los archivos de objetos que produjo el ensamblador. Esta salida puede ser una biblioteca compartida (o dinámica) (y aunque el nombre es similar, no tienen mucho en común con las bibliotecas estáticas mencionadas anteriormente) o un archivo ejecutable.

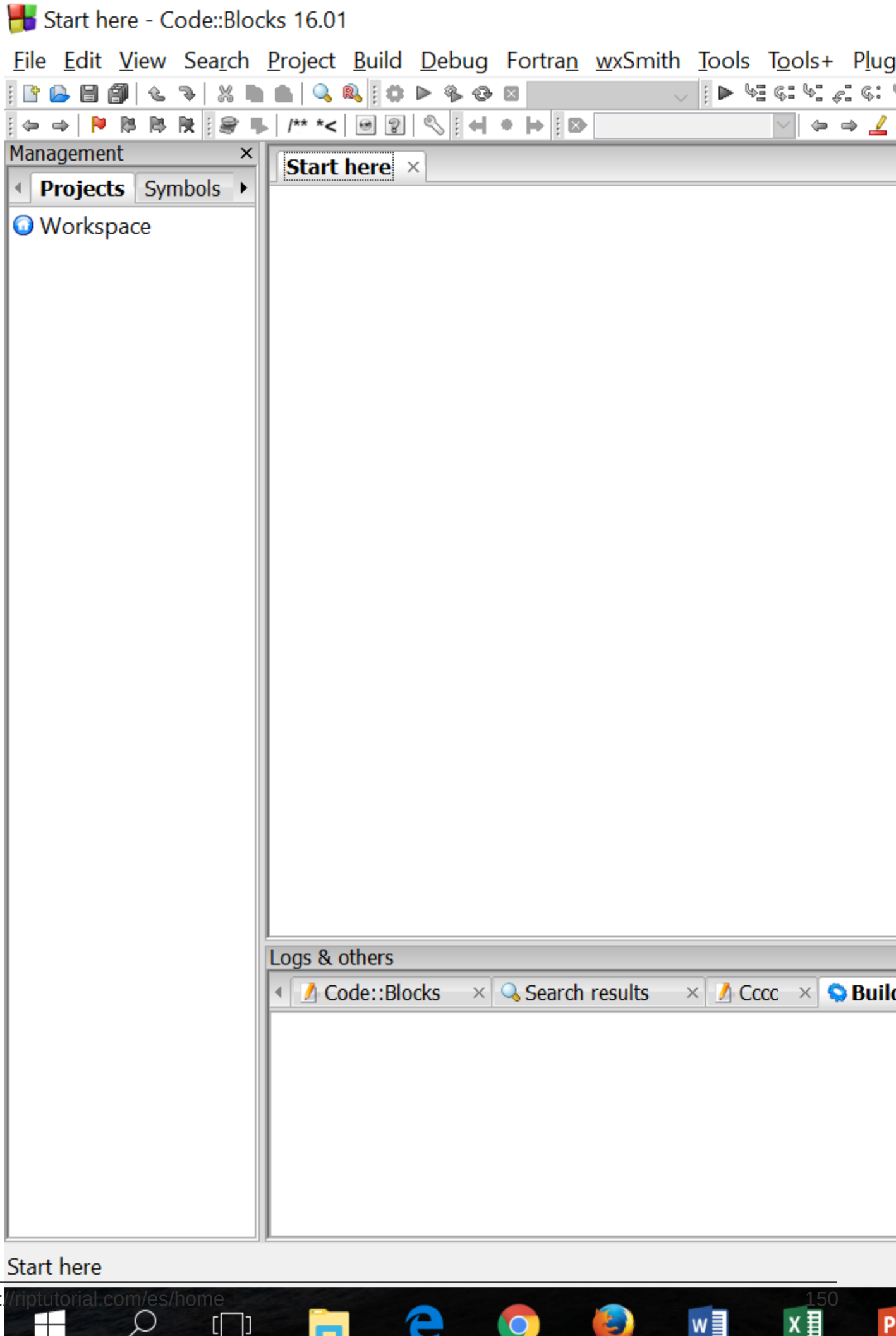
Vincula todos los archivos de objetos reemplazando las referencias a símbolos no definidos con las direcciones correctas. Cada uno de estos símbolos se puede definir en otros archivos de objetos o en bibliotecas. Si están definidas en bibliotecas distintas de la biblioteca estándar, debe informar al vinculador sobre ellas.

En esta etapa, los errores más comunes son las definiciones faltantes o las definiciones duplicadas. Lo primero significa que las definiciones no existen (es decir, no están escritas), o que los archivos de objetos o las bibliotecas donde residen no se entregaron al vinculador. Lo último es obvio: el mismo símbolo se definió en dos archivos de objetos o bibliotecas diferentes.

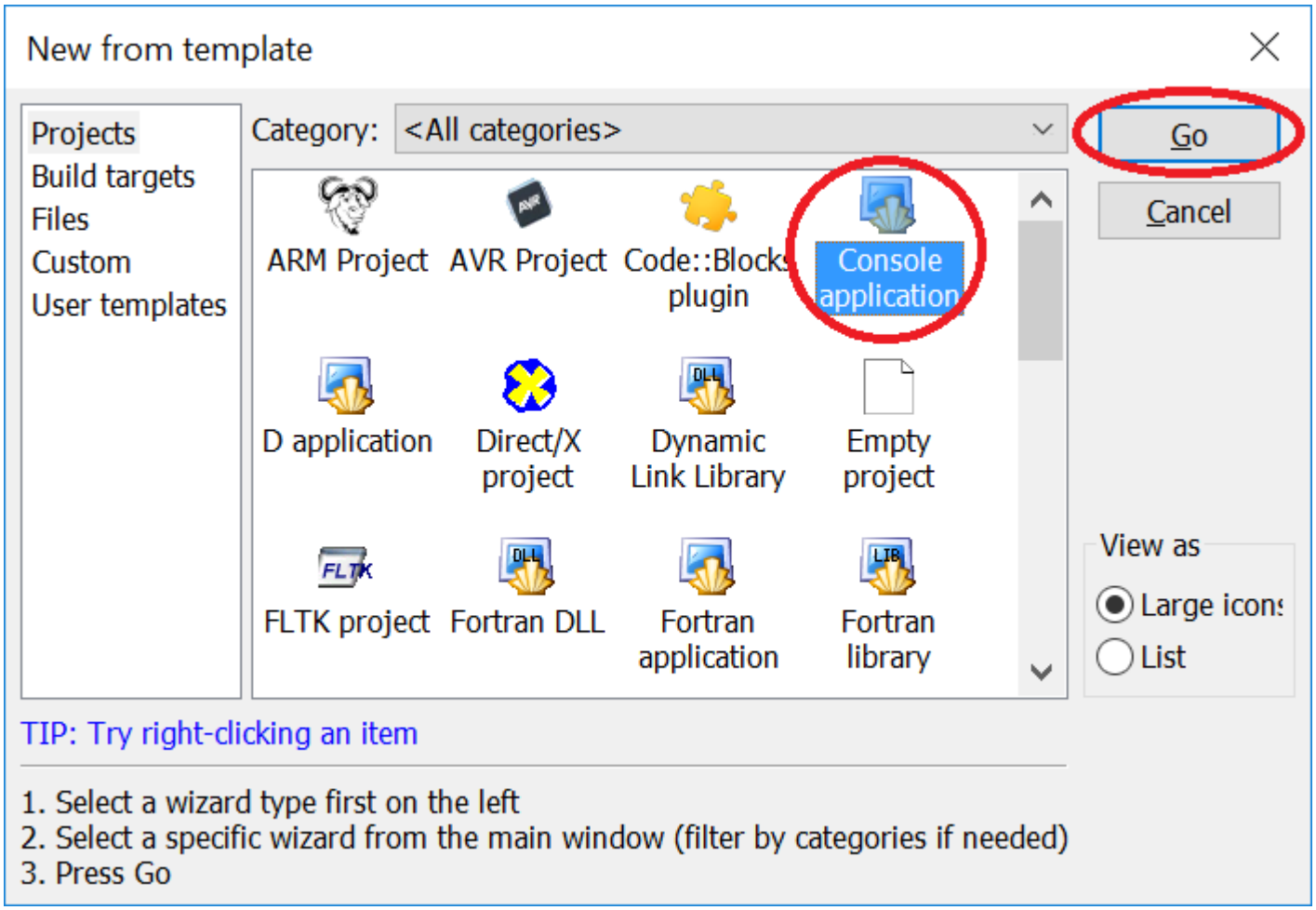
Compilando con Code :: Blocks (interfaz gráfica)

1. Descargue e instale Code :: Blocks aquí . Si está en Windows, tenga cuidado de seleccionar un archivo cuyo nombre contenga `mingw` , los otros archivos no instalarán ningún compilador.

2. Abra Code :: Blocks y haga clic en "Crear un nuevo proyecto":

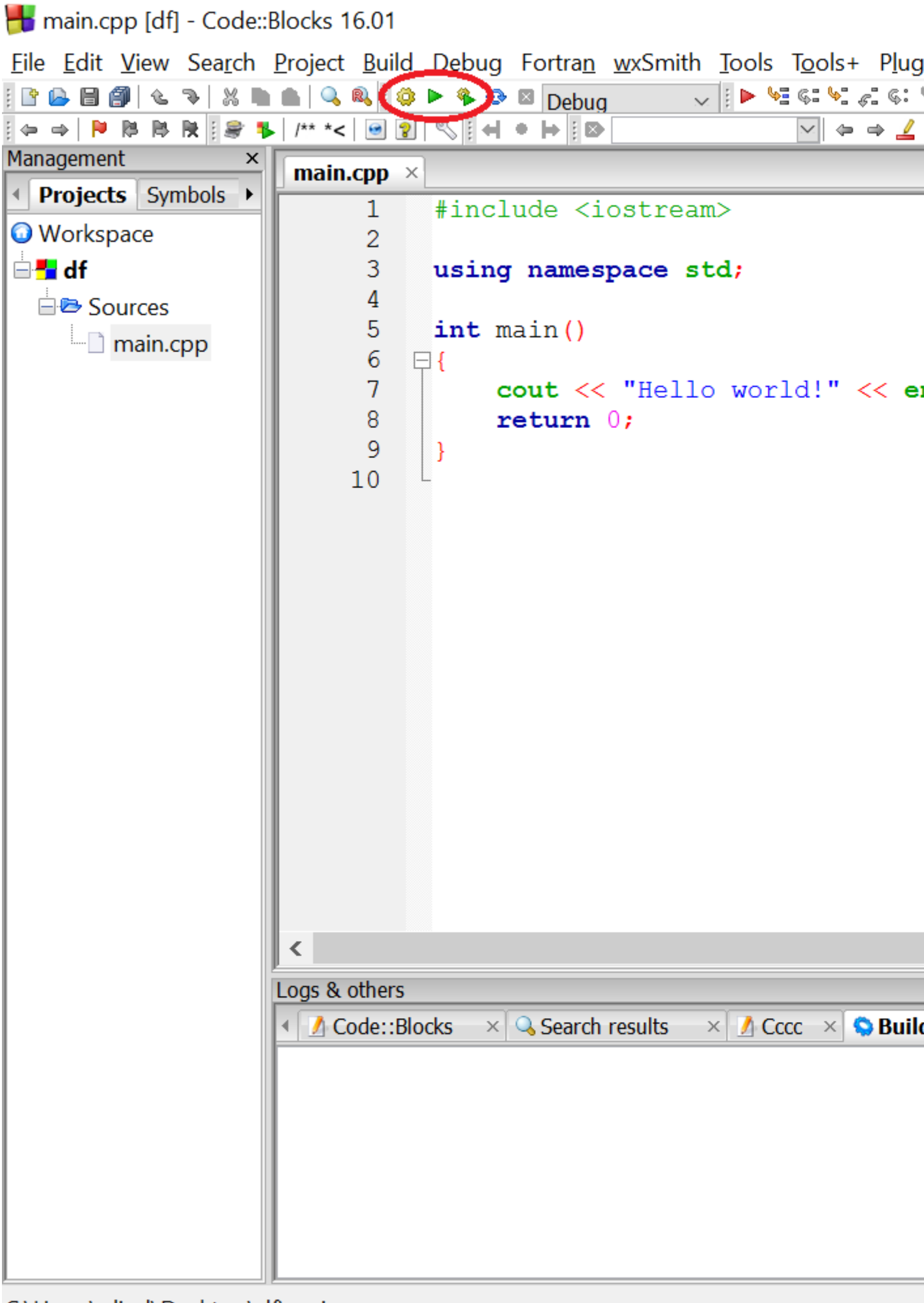

Start here - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Management
Projects
Symbols
Workspace
Start here
Logs & others
Code::Blocks
Search results
Cccc
Start here

3. Seleccione "Aplicación de consola" y haga clic en "Ir":



4. Haga clic en "Siguiente", seleccione "C ++", haga clic en "Siguiente", seleccione un nombre para su proyecto y elija una carpeta para guardarlo, haga clic en "Siguiente" y luego haga clic en "Finalizar".

5. Ahora puedes editar y compilar tu código. Un código predeterminado que imprime "¡Hola mundo!" En la consola ya está ahí. Para compilar y / o ejecutar su programa, presione uno de los tres botones de compilar / ejecutar en la barra de herramientas:

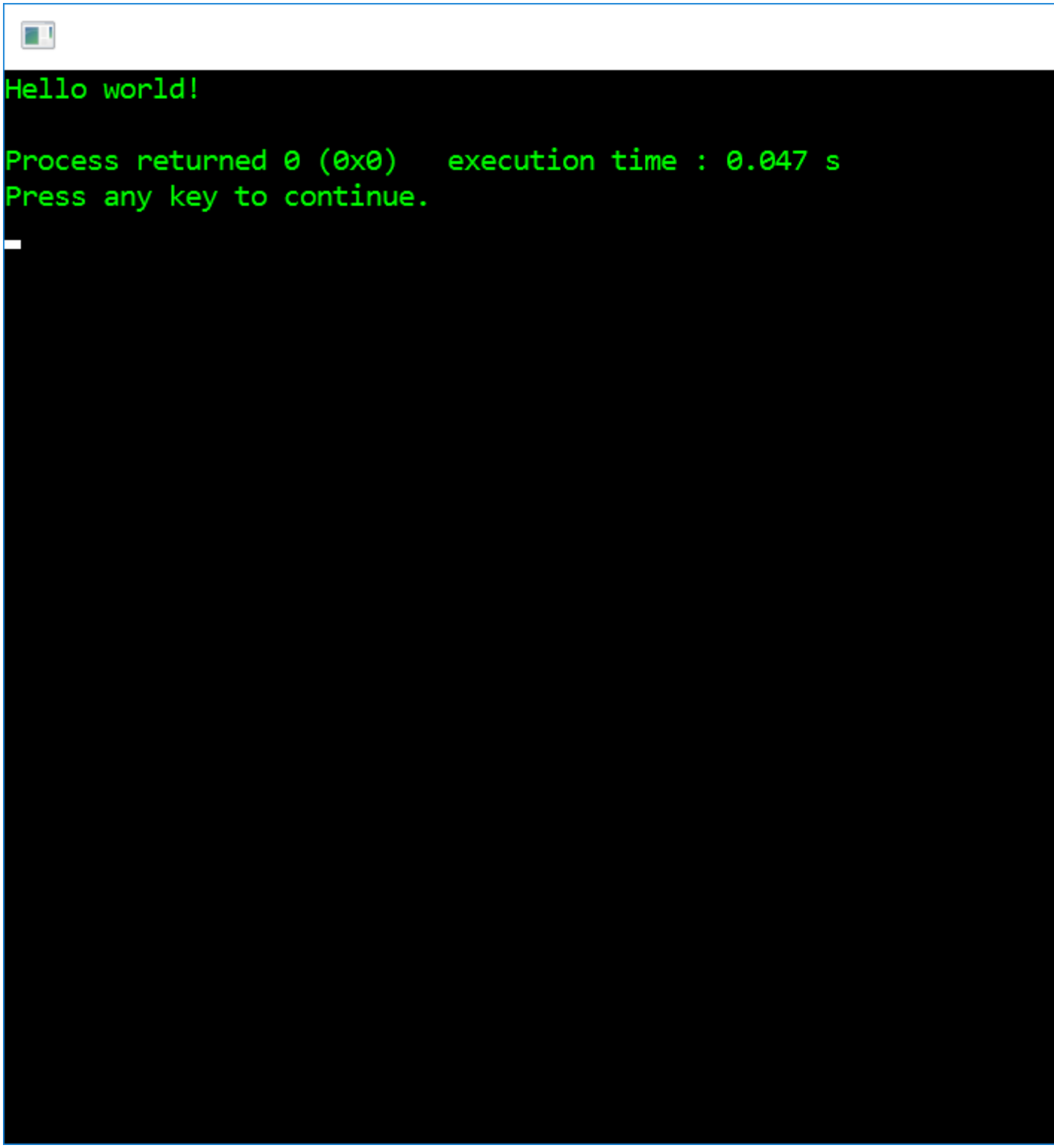

main.cpp [df] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Debug
Management
Projects Symbols
Workspace
df
Sources
main.cpp
main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7 cout << "Hello world!" << en
8 return 0;
9 }
10
Logs & others
Code::Blocks
Search results
Cccc
C:\Users\glind\Desktop\df\main.cpp

Para compilar sin correr, pulsa , para correr sin compilar nuevamente, presione y para compilar y luego correr, presiona .

y para compilar y luego correr, presiona .

y para compilar y luego correr, presiona .

Compilando y ejecutando el predeterminado "¡Hola mundo!" El código da el siguiente resultado:



Lea Compilando y construyendo en línea:
https://riptutorial.com/es/cplusplus/topic/4708/compilando-y-construyendo

Capítulo 21: Comportamiento definido por la implementación

Examples

Char puede estar sin firmar o firmado

El estándar no especifica si `char` debe estar firmado o sin firmar. Diferentes compiladores lo implementan de manera diferente, o podrían permitir cambiarlo usando un interruptor de línea de comando.

Tamaño de los tipos integrales.

Los siguientes tipos se definen como *tipos integrales* :

- `char`
- Tipos enteros firmados
- Tipos de enteros sin signo
- `char16_t` y `char32_t`
- `bool`
- `wchar_t`

Con la excepción de `sizeof(char)` / `sizeof(signed char)` / `sizeof(unsigned char)` , que se divide entre § 3.9.1.1 [basic.fundamental / 1] y § 5.3.3.1 [expr.sizeof], y `sizeof(bool)` , que está completamente definido por la implementación y no tiene un tamaño mínimo, los requisitos de tamaño mínimo de estos tipos se indican en la sección § 3.9.1 [basic.fundamental] de la norma, y se detallarán a continuación.

Tamaño de `char`

Todas las versiones de estándar el C ++ especifican, en § 5.3.3.1, que `sizeof` rendimientos `1` para `unsigned char` , `signed char` , y `char` (es definido por la implementación si el `char` tipo se `signed` o `unsigned`).

C ++ 14

`char` es lo suficientemente grande como para representar 256 valores diferentes, para ser adecuado para almacenar unidades de código UTF-8.

Tamaño de los tipos enteros con signo y sin signo

El estándar especifica, en el § 3.9.1.2, que en la lista de *tipos enteros con signo estándar* , que constan de caracteres con `signed char` , `short int` , `int` , `long int` `long long int` , cada tipo proporcionará al menos tanto almacenamiento como los anteriores en la lista. Además, como se especifica en el § 3.9.1.3, cada uno de estos tipos tiene un *tipo de entero sin signo estándar* correspondiente, `unsigned char` `unsigned short int` `unsigned int` , `unsigned short int` `unsigned int` , `unsigned int` `unsigned long int` , `unsigned long int` `unsigned long long int` , que tiene el mismo tamaño y alineación que Su correspondiente tipo firmado. Además, como se especifica en el § 3.9.1.1, `char` tiene los mismos requisitos de tamaño y alineación que el `signed char` y el `unsigned char` .

C ++ 11

Antes de C ++ 11, `long long` y `unsigned long long` no formaban parte oficialmente del estándar C ++. Sin embargo, después de su introducción a C, en C99, muchos compiladores admitieron `long long` como un *tipo entero con signo extendido* , y `unsigned long long` como un *tipo entero sin signo extendido* , con las mismas reglas que los tipos C.

La norma garantiza así que:

```
1 == sizeof(char)  == sizeof(signed char) == sizeof(unsigned char)
  <= sizeof(short) == sizeof(unsigned short)
  <= sizeof(int)   == sizeof(unsigned int)
  <= sizeof(long)  == sizeof(unsigned long)
```

C ++ 11

```
 <= sizeof(long long) == sizeof(unsigned long long)
```

Los tamaños mínimos específicos para cada tipo no están dados por la norma. En su lugar, cada tipo tiene un rango mínimo de valores que puede admitir, que, como se especifica en § 3.9.1.3, se hereda del estándar C, en §5.2.4.2.1. El tamaño mínimo de cada tipo se puede inferir aproximadamente de este rango, al determinar el número mínimo de bits requeridos; tenga en cuenta que para cualquier plataforma dada, el rango real admitido de cualquier tipo puede ser mayor que el mínimo. Tenga en cuenta que para los tipos con signo, los rangos corresponden al complemento de uno, no al complemento de dos de uso más común; esto es para permitir que una gama más amplia de plataformas cumpla con el estándar.

Tipo	Rango mínimo	Bits mínimos requeridos
`signed char`	-127 a 127 (- (2 ^{7} - 1) a (2 ^{7} - 1))	8
`unsigned char`	0 a 255 (0 a 2 ^{8} - 1)	8
`signed short`	-32,767 a 32,767 (- (2 ^{15} - 1) a (2 ^{15} - 1))	dieciséis
`unsigned short`	0 a 65,535 (0 a 2 ^{16} - 1)	dieciséis
`signed int`	-32,767 a 32,767 (- (2 ^{15} - 1) a (2 ^{15} - 1))	dieciséis

Tipo	Rango mínimo	Bits mínimos requeridos
`unsigned int`	0 a 65,535 (0 a $2^{16} - 1$)	dieciséis
`signed long`	-2,147,483,647 a 2,147,483,647 (- ($2^{31} - 1$) a ($2^{31} - 1$))	32
`unsigned long`	0 a 4,294,967,295 (0 a $2^{32} - 1$)	32

C ++ 11

Tipo	Rango mínimo	Bits mínimos requeridos
`signed long long`	-9,223,372,036,854,775,807 a 9,223,372,036,854,775,807 (- ($2^{63} - 1$) a ($2^{63} - 1$))	64
`unsigned long long`	0 a 18,446,744,073,709,551,615 (0 a $2^{64} - 1$)	64

Como se permite que cada tipo sea mayor que su requisito de tamaño mínimo, los tipos pueden diferir en tamaño entre las implementaciones. El ejemplo más notable de esto es con los modelos de datos de 64 bits LP64 y LLP64, donde los sistemas LLP64 (tales como Windows de 64 bits) tiene 32 bits `int`s y `long` s, y sistemas de LP64 (tales como Linux de 64 bits) tienen `int` s de 32 bits y s de 64 bits de `long` . Debido a esto, no se puede suponer que los tipos de enteros tengan un ancho fijo en todas las plataformas.

C ++ 11

Si se requieren tipos de enteros con ancho fijo, use tipos del encabezado `<cstdint>` , pero tenga en cuenta que el estándar hace que las implementaciones sean compatibles con los tipos de ancho exacto `int8_t` , `int16_t` , `int32_t` , `int64_t` , `intptr_t` , `uint8_t` , `uint16_t` , `uint32_t` , `uint64_t` y `uintptr_t` .

C ++ 11

Tamaño de `char16_t` y `char32_t`

Los tamaños de `char16_t` y `char32_t` están definidos por la implementación, como se especifica en el § 5.3.3.1, con las estipulaciones que figuran en el § 3.9.1.5:

- `char16_t` es lo suficientemente grande como para representar cualquier unidad de código UTF-16, y tiene el mismo tamaño, firmeza y alineación que `uint_least16_t` ; por lo tanto, se requiere que tenga al menos 16 bits de tamaño.
- `char32_t` es lo suficientemente grande como para representar cualquier unidad de código UTF-32, y tiene el mismo tamaño, firmeza y alineación que `uint_least32_t` ; por lo tanto, se requiere que tenga al menos 32 bits de tamaño.

Tamaño de `bool`

El tamaño de `bool` está definido en la implementación, y puede o no ser `1` .

Tamaño de wchar_t

`wchar_t` , como se especifica en § 3.9.1.5, es un tipo distinto, cuyo rango de valores puede representar cada unidad de código distinta del conjunto de caracteres extendido más grande entre los locales admitidos. Tiene el mismo tamaño, firmeza y alineación que uno de los otros tipos integrales, que se conoce como su *tipo subyacente* . El tamaño de este tipo está definido por la implementación, como se especifica en el § 5.3.3.1, y puede ser, por ejemplo, al menos 8, 16 o 32 bits; si un sistema admite Unicode, por ejemplo, se requiere que `wchar_t` tenga al menos 32 bits (una excepción a esta regla es Windows, donde `wchar_t` es de 16 bits por motivos de compatibilidad). Se hereda de la norma C90, ISO 9899: 1990 § 4.1.5, con solo una pequeña redacción.

Dependiendo de la implementación, el tamaño de `wchar_t` es a menudo, pero no siempre, de 8, 16 o 32 bits. Los ejemplos más comunes de estos son:

- En sistemas similares a Unix y Unix, `wchar_t` es de 32 bits, y generalmente se usa para UTF-32.
- En Windows, `wchar_t` es de 16 bits y se usa para UTF-16.
- En un sistema que solo tiene soporte de 8 bits, `wchar_t` es de 8 bits.

C ++ 11

Si se desea compatibilidad con Unicode, se recomienda usar `char` para UTF-8, `char16_t` para UTF-16 o `char32_t` para UTF-32, en lugar de usar `wchar_t` .

Modelos de datos

Como se mencionó anteriormente, los anchos de los tipos de enteros pueden diferir entre plataformas. Los modelos más comunes son los siguientes, con tamaños especificados en bits:

Modelo	`int`	`long`	puntero
LP32 (2/4/4)	dieciséis	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64
LP64 (4/8/8)	32	64	64

Fuera de estos modelos:

- Windows de 16 bits utiliza LP32.
- Los sistemas de 32 bits * nix (Unix, Linux, Mac OSX y otros sistemas operativos similares a Unix) y Windows usan ILP32.
- Windows de 64 bits utiliza LLP64.
- Los sistemas de 64 bits * nix utilizan LP64.

Tenga en cuenta, sin embargo, que estos modelos no se mencionan específicamente en la norma en sí.

Número de bits en un byte

En C ++, un *byte* es el espacio ocupado por un objeto `char` . La cantidad de bits en un byte viene dada por `CHAR_BIT` , que se define en `climits` y se requiere que sea al menos 8. Mientras que la mayoría de los sistemas modernos tienen bytes de 8 bits, y POSIX requiere que `CHAR_BIT` sea exactamente 8, hay algunos sistemas donde `CHAR_BIT` es mayor que 8, es decir, un solo byte puede estar compuesto por 8, 16, 32 o 64 bits.

Valor numérico de un puntero

El resultado de convertir un puntero a un entero usando `reinterpret_cast` está definido por la implementación, pero "... no es sorprendente para aquellos que conocen la estructura de direccionamiento de la máquina subyacente".

```
int x = 42;
int* p = &x;
long addr = reinterpret_cast<long>(p);
std::cout << addr << "\n"; // prints some numeric address,
                           // probably in the architecture's native address format
```

Del mismo modo, el puntero obtenido por conversión de un entero también está definido por la implementación.

La forma correcta de almacenar un puntero como un entero es usando los tipos `uintptr_t` o `intptr_t` :

```
// `uintptr_t` was not in C++03. It's in C99, in <stdint.h>, as an optional type
#include <stdint.h>

uintptr_t uip;
```

C ++ 11

```
// There is an optional `std::uintptr_t` in C++11
#include <cstdint>

std::uintptr_t uip;
```

C ++ 11 hace referencia a C99 para la definición `uintptr_t` (estándar C99, 6.3.2.3):

un tipo de entero sin signo con la propiedad de que cualquier puntero válido para `void` se puede convertir a este tipo, luego se puede volver a convertir en puntero a `void` , y el resultado se comparará igual al puntero original.

Si bien, para la mayoría de las plataformas modernas, puede asumir un espacio de direcciones plano y que la aritmética en `uintptr_t` es equivalente a la aritmética en `char *` , es totalmente posible que una implementación realice una transformación al `uintptr_t` `void *` a `uintptr_t` siempre que la transformación pueda se invierte cuando se devuelve desde `uintptr_t` a `void *` .

Tecnicismos

- En los sistemas `intptr_t` XSI (X / Open System Interfaces), se requieren los tipos `intptr_t` y `uintptr_t` , de lo contrario son **opcionales** .
- En el sentido del estándar C, las funciones no son objetos; el estándar C no garantiza que `uintptr_t` pueda contener un puntero de función. De todos modos, la conformidad con POSIX (2.12.3) requiere que:

  > Todos los tipos de punteros de función tendrán la misma representación que el puntero de tipo para anular. La conversión de un puntero de función a void * no alterará la representación. Un valor nulo * resultante de dicha conversión se puede convertir de nuevo al tipo de puntero de función original, utilizando una conversión explícita, sin pérdida de información.

- C99 §7.18.1:

  > Cuando se definen los nombres typedef que difieren solo en la ausencia o presencia de la u inicial, denotarán los tipos correspondientes firmados y no firmados como se describe en 6.2.5; una implementación que provea uno de estos tipos correspondientes también proporcionará el otro.

  `uintptr_t` podría tener sentido si quiere hacer cosas a los bits del puntero que no puede hacer con sensatez con un entero con signo.

Rangos de tipos numéricos

Los rangos de los tipos de enteros están definidos por la implementación. El encabezado `<limits>` proporciona la plantilla `std::numeric_limits<T>` que proporciona los valores mínimo y máximo de todos los tipos fundamentales. Los valores satisfacen las garantías proporcionadas por el estándar C a través de los `<climits>` y (> = C ++ 11) `<cinttypes>` .

- `std::numeric_limits<signed char>::min()` es igual a `SCHAR_MIN` , que es menor o igual que -127.
- `std::numeric_limits<signed char>::max()` es igual a `SCHAR_MAX` , que es mayor o igual a 127.
- `std::numeric_limits<unsigned char>::max()` es igual a `UCHAR_MAX` , que es mayor o igual a 255.
- `std::numeric_limits<short>::min()` es igual a `SHRT_MIN` , que es menor o igual que -32767.
- `std::numeric_limits<short>::max()` es igual a `SHRT_MAX` , que es mayor o igual que 32767.
- `std::numeric_limits<unsigned short>::max()` es igual a `USHRT_MAX` , que es mayor o igual a 65535.

- `std::numeric_limits<int>::min()` es igual a `INT_MIN` , que es menor o igual que -32767.
- `std::numeric_limits<int>::max()` es igual a `INT_MAX` , que es mayor o igual a 32767.
- `std::numeric_limits<unsigned int>::max()` es igual a `UINT_MAX` , que es mayor o igual a 65535.
- `std::numeric_limits<long>::min()` es igual a `LONG_MIN` , que es menor o igual a -2147483647.
- `std::numeric_limits<long>::max()` es igual a `LONG_MAX` , que es mayor o igual que 2147483647.
- `std::numeric_limits<unsigned long>::max()` es igual a `ULONG_MAX` , que es mayor o igual que 4294967295.

C ++ 11

- `std::numeric_limits<long long>::min()` es igual a `LLONG_MIN` , que es menor o igual que -9223372036854775807.
- `std::numeric_limits<long long>::max()` es igual a `LLONG_MAX` , que es mayor o igual que 9223372036854775807.
- `std::numeric_limits<unsigned long long>::max()` es igual a `ULLONG_MAX` , que es mayor o igual que 18446744073709551615.

Para los tipos de punto flotante `T` , `max()` es el valor finito máximo, mientras que `min()` es el valor normalizado positivo mínimo. Se proporcionan miembros adicionales para los tipos de punto flotante, que también están definidos por la implementación pero satisfacen ciertas garantías proporcionadas por el estándar C a través del encabezado `<cfloat>` .

- El `digits10` da el número de dígitos decimales de precisión.
  - `std::numeric_limits<float>::digits10` es igual a `FLT_DIG` , que es al menos 6.
  - `std::numeric_limits<double>::digits10` es igual a `DBL_DIG` , que es al menos 10.
  - `std::numeric_limits<long double>::digits10` es igual a `LDBL_DIG` , que es al menos 10.
- El miembro `min_exponent10` es el E negativo mínimo tal que 10 a la potencia E es normal.
  - `std::numeric_limits<float>::min_exponent10` es igual a `FLT_MIN_10_EXP` , que es como máximo -37.
  - `std::numeric_limits<double>::min_exponent10` es igual a `DBL_MIN_10_EXP` , que es como máximo -37. `std::numeric_limits<long double>::min_exponent10` es igual a `LDBL_MIN_10_EXP` , que es como máximo -37.
- El miembro `max_exponent10` es el E máximo, de modo que 10 para la potencia E es finito.
  - `std::numeric_limits<float>::max_exponent10` es igual a `FLT_MIN_10_EXP` , que es al menos 37.
  - `std::numeric_limits<double>::max_exponent10` es igual a `DBL_MIN_10_EXP` , que es al menos 37.
  - `std::numeric_limits<long double>::max_exponent10` es igual a `LDBL_MIN_10_EXP` , que es al menos 37.
- Si el miembro `is_iec559` es verdadero, el tipo cumple con IEC 559 / IEEE 754 y, por lo tanto, su rango está determinado por esa norma.

Representación del valor de los tipos de punto flotante

El estándar requiere que el `long double` proporcione al menos la misma precisión que el `double` , lo que proporciona al menos la misma precisión que el `float` ; y que un `long double` puede representar cualquier valor que un `double` pueda representar, mientras que un `double` puede

representar cualquier valor que un `float` pueda representar. Los detalles de la representación están, sin embargo, definidos por la implementación.

Para un tipo de punto flotante `T` , `std::numeric_limits<T>::radix` especifica el radix utilizado por la representación de `T`

Si `std::numeric_limits<T>::is_iec559` es verdadero, entonces la representación de `T` coincide con uno de los formatos definidos por IEC 559 / IEEE 754.

Desbordamiento al convertir de entero a entero con signo

Cuando un entero con signo o sin signo se convierte en un tipo de entero con signo y su valor no se puede representar en el tipo de destino, el valor producido se define por la implementación. Ejemplo:

```
// Suppose that on this implementation, the range of signed char is -128 to +127 and
// the range of unsigned char is 0 to 255
int x = 12345;
signed char sc = x;   // sc has an implementation-defined value
unsigned char uc = x; // uc is initialized to 57 (i.e., 12345 modulo 256)
```

Tipo subyacente (y, por tanto, tamaño) de una enumeración

Si el tipo subyacente no se especifica explícitamente para un tipo de enumeración sin ámbito, se determina de una manera definida por la implementación.

```
enum E {
    RED,
    GREEN,
    BLUE,
};
using T = std::underlying_type<E>::type; // implementation-defined
```

Sin embargo, el estándar requiere que el tipo subyacente de una enumeración no sea mayor que `int` menos que `int` y `unsigned int` no puedan representar todos los valores de la enumeración. Por lo tanto, en el código anterior, `T` podría ser `int` , `unsigned int` , o `short` , pero no `long long` , para dar algunos ejemplos.

Tenga en cuenta que una enumeración tiene el mismo tamaño (según lo devuelto por `sizeof`) que su tipo subyacente.

Lea Comportamiento definido por la implementación en línea:
https://riptutorial.com/es/cplusplus/topic/1363/comportamiento-definido-por-la-implementacion

Capítulo 22: Comportamiento indefinido

Introducción

¿Qué es el comportamiento indefinido (UB)? De acuerdo con la norma ISO C ++ (§1.3.24, N4296), es un "comportamiento por el que esta norma internacional no impone requisitos".

Esto significa que cuando un programa se encuentra con UB, se le permite hacer lo que quiera. A menudo, esto significa un choque, pero puede que simplemente no haga nada, haga que los demonios salgan volando por tu nariz , ¡o incluso *parece* que funciona correctamente!

No hace falta decir que debes evitar escribir código que invoque a UB.

Observaciones

Si un programa contiene un comportamiento indefinido, el estándar de C ++ no impone restricciones a su comportamiento.

- Puede parecer que funciona según lo previsto por el desarrollador, pero también puede fallar o producir resultados extraños.
- El comportamiento puede variar entre ejecuciones del mismo programa.
- Cualquier parte del programa puede funcionar mal, incluidas las líneas que vienen antes de la línea que contiene un comportamiento indefinido.
- La implementación no es necesaria para documentar el resultado de un comportamiento indefinido.

Una implementación *puede* documentar el resultado de una operación que produce un comportamiento indefinido de acuerdo con el estándar, pero un programa que depende de dicho comportamiento documentado no es portátil.

¿Por qué existe un comportamiento indefinido?

Intuitivamente, el comportamiento indefinido se considera algo malo, ya que tales errores no pueden manejarse con amabilidad mediante, por ejemplo, controladores de excepciones.

Pero dejar un comportamiento indefinido es en realidad una parte integral de la promesa de C ++ "no pagas por lo que no usas". El comportamiento indefinido permite que un compilador asuma que el desarrollador sabe lo que está haciendo y no introduce código para verificar los errores resaltados en los ejemplos anteriores.

Encontrar y evitar comportamientos indefinidos.

Algunas herramientas se pueden usar para descubrir un comportamiento indefinido durante el desarrollo:

- La mayoría de los compiladores tienen marcas de advertencia para advertir sobre algunos casos de comportamiento indefinido en tiempo de compilación.

- Las versiones más recientes de gcc y clang incluyen un indicador denominado "Desinfectante de comportamiento indefinido" (`-fsanitize=undefined`) que verificará el comportamiento indefinido en el tiempo de ejecución, a un costo de rendimiento.
- `lint` herramientas similares a `lint` pueden realizar un análisis de comportamiento indefinido más completo.

Comportamiento indefinido, no especificado y definido por la implementación

De la sección 1.9 (Ejecución del programa) de la norma C ++ 14 (ISO / IEC 14882: 2014):

1. Las descripciones semánticas en esta Norma Internacional definen una máquina abstracta no determinista parametrizada. [CORTAR]

2. Ciertos aspectos y operaciones de la máquina abstracta se describen en esta Norma Internacional como **definidos** por la **implementación** (por ejemplo, `sizeof(int)`). Estos constituyen *los parámetros de la máquina abstracta* . Cada implementación deberá incluir documentación que describa sus características y comportamiento en estos aspectos. [CORTAR]

3. Ciertos otros aspectos y operaciones de la máquina abstracta se describen en esta Norma Internacional como **no especificados** (por ejemplo, evaluación de expresiones en un *nuevo inicializador* si la función de asignación no puede asignar memoria). Donde sea posible, esta norma internacional define un conjunto de comportamientos permitidos. Estos definen los aspectos no deterministas de la máquina abstracta. Una instancia de la máquina abstracta puede tener más de una ejecución posible para un programa dado y una entrada dada.

4. Ciertas otras operaciones se describen en esta Norma Internacional como **indefinidas** (o ejemplo, el efecto de intentar modificar un objeto `const`). [*Nota* : esta Norma Internacional no impone requisitos sobre el comportamiento de los programas que contienen un comportamiento indefinido. - *nota final*]

Examples

Leer o escribir a través de un puntero nulo.

```
int *ptr = nullptr;
*ptr = 1; // Undefined behavior
```

Este es **un comportamiento indefinido** , porque un puntero nulo no apunta a ningún objeto válido, por lo que no hay ningún objeto en `*ptr` para escribir.

Aunque esto causa con mayor frecuencia un fallo de segmentación, no está definido y puede pasar cualquier cosa.

No hay declaración de retorno para una función con un tipo de retorno no

nulo

Omitir la declaración de `return` en una función que tiene un tipo de retorno que no es `void` es **un comportamiento indefinido** .

```
int function() {
    // Missing return statement
}

int main() {
    function(); //Undefined Behavior
}
```

La mayoría de los compiladores de hoy en día emiten una advertencia en el momento de la compilación para este tipo de comportamiento indefinido.

Nota: `main` es la única excepción a la regla. Si `main` no tiene una declaración de `return` , el compilador inserta automáticamente `return 0;` Para ti, para que puedas dejarlo fuera de forma segura.

Modificar un literal de cadena

C ++ 11

```
char *str = "hello world";
str[0] = 'H';
```

`"hello world"` es una cadena literal, por lo que modificarlo da un comportamiento indefinido.

La inicialización de `str` en el ejemplo anterior fue obsoleta formalmente (programada para su eliminación de una versión futura del estándar) en C ++ 03. Varios compiladores antes de 2003 podrían emitir una advertencia sobre esto (por ejemplo, una conversión sospechosa). Después de 2003, los compiladores suelen advertir sobre una conversión obsoleta.

C ++ 11

El ejemplo anterior es ilegal y da como resultado un diagnóstico del compilador, en C ++ 11 y versiones posteriores. Se puede construir un ejemplo similar para mostrar un comportamiento indefinido permitiendo explícitamente la conversión de tipo, como:

```
char *str = const_cast<char *>("hello world");
str[0] = 'H';
```

Accediendo a un índice fuera de límites

Es **un comportamiento indefinido** acceder a un índice que está fuera de los límites de una matriz (o el contenedor de la biblioteca estándar para esa materia, ya que todos se implementan utilizando una matriz sin *procesar*):

```
int array[] = {1, 2, 3, 4, 5};
array[5] = 0;  // Undefined behavior
```

Se *permite* tener un puntero que apunta al final de la matriz (en este caso, `array + 5`), simplemente no se puede eliminar la referencia, ya que no es un elemento válido.

```
const int *end = array + 5;  // Pointer to one past the last index
for (int *p = array; p != end; ++p)
  // Do something with `p`
```

En general, no se le permite crear un puntero fuera de límites. Un puntero debe apuntar a un elemento dentro de la matriz, o uno más allá del final.

División entera por cero

```
int x = 5 / 0;    // Undefined behavior
```

La división por `0` está definida matemáticamente y, como tal, tiene sentido que se trate de un comportamiento indefinido.

Sin embargo:

```
float x = 5.0f / 0.0f;   // x is +infinity
```

La mayoría de la implementación implementa IEEE-754, que define la división de punto flotante por cero para devolver `NaN` (si el numerador es `0.0f`), el `infinity` (si el numerador es positivo) o el `-infinity` (si el numerador es negativo).

Desbordamiento de enteros firmado

```
int x = INT_MAX + 1;

// x can be anything -> Undefined behavior
```

> Si durante la evaluación de una expresión, el resultado no está definido matemáticamente o no está en el rango de valores representables para su tipo, el comportamiento no está definido.

(C ++ 11 párrafo 5/4 estándar)

Este es uno de los más desagradables, ya que por lo general produce un comportamiento reproducible y sin interrupciones, por lo que los desarrolladores pueden verse tentados a confiar en gran medida en el comportamiento observado.

Por otra parte:

```
unsigned int x = UINT_MAX + 1;
```

```
// x is 0
```

está bien definido ya que:

> Los enteros sin signo, declarados sin firmar, obedecerán las leyes del módulo aritmético `2^n` donde `n` es el número de bits en la representación del valor de ese tamaño particular de entero.

(C ++ 11 párrafo 3.9.1 / 4)

A veces los compiladores pueden explotar un comportamiento indefinido y optimizar

```
signed int x ;
if(x > x + 1)
{
    //do something
}
```

Aquí, dado que no se define un desbordamiento de enteros con signo, el compilador es libre de asumir que nunca puede suceder y, por lo tanto, puede optimizar el bloque "if"

Usando una variable local sin inicializar

```
int a;
std::cout << a; // Undefined behavior!
```

Esto da como resultado **un comportamiento indefinido** , porque `a` no está inicializado.

A menudo, incorrectamente, se afirma que esto se debe a que el valor es "indeterminado", o "cualquier valor que haya en esa ubicación de memoria antes". Sin embargo, es el hecho de acceder al valor de `a` en el ejemplo anterior lo que da un comportamiento indefinido. En la práctica, la impresión de un "valor de basura" es un síntoma común en este caso, pero esa es solo una forma posible de comportamiento indefinido.

Aunque es muy poco probable en la práctica (ya que depende del soporte de hardware específico) el compilador podría electrocutar al programador al compilar el ejemplo de código anterior. Con un compilador y un soporte de hardware de este tipo, tal respuesta al comportamiento indefinido aumentaría notablemente el entendimiento promedio (vivo) del programador del verdadero significado del comportamiento indefinido, que es que el estándar no impone ninguna restricción al comportamiento resultante.

C ++ 14

El uso de un valor indeterminado de tipo de `unsigned char` no produce un comportamiento indefinido si el valor se utiliza como:

- el segundo o tercer operando del operador condicional ternario;
- el operando derecho del operador de coma incorporado;
- el operando de una conversión a caracteres `unsigned char` ;

- el operando derecho del operador de asignación, si el operando izquierdo también es de tipo `unsigned char` ;
- el inicializador para un objeto `unsigned char` ;

o si el valor es descartado. En tales casos, el valor indeterminado simplemente se propaga al resultado de la expresión, si corresponde.

Tenga en cuenta que una variable `static` **siempre se** inicializa con cero (si es posible):

```
static int a;
std::cout << a; // Defined behavior, 'a' is 0
```

Múltiples definiciones no idénticas (la regla de una definición)

Si una clase, enumeración, función en línea, plantilla o miembro de una plantilla tiene un enlace externo y se define en múltiples unidades de traducción, todas las definiciones deben ser idénticas o el comportamiento no está definido según la Regla de una definición (ODR) .

`foo.h` :

```
class Foo {
  public:
    double x;
  private:
    int y;
};

Foo get_foo();
```

`foo.cpp` :

```
#include "foo.h"
Foo get_foo() { /* implementation */ }
```

`main.cpp` :

```
// I want access to the private member, so I am going to replace Foo with my own type
class Foo {
  public:
    double x;
    int y;
};
Foo get_foo(); // declare this function ourselves since we aren't including foo.h
int main() {
    Foo foo = get_foo();
    // do something with foo.y
}
```

El programa anterior muestra un comportamiento indefinido porque contiene dos definiciones de la clase `::Foo` , que tiene un enlace externo, en diferentes unidades de traducción, pero las dos definiciones no son idénticas. A diferencia de la redefinición de una clase dentro de la *misma* unidad de traducción, este compilador no requiere que este problema sea diagnosticado.

Emparejamiento incorrecto de la asignación de memoria y desasignación

Un objeto solo puede ser desasignado por `delete` si fue asignado por `new` y no es una matriz. Si el argumento para `delete` no fue devuelto por `new` o es una matriz, el comportamiento no está definido.

Un objeto solo puede ser desasignado por `delete[]` si fue asignado por `new` y es una matriz. Si el argumento para `delete[]` no fue devuelto por `new` o no es una matriz, el comportamiento no está definido.

Si el argumento para `free` no fue devuelto por `malloc` , el comportamiento es indefinido.

```
int* p1 = new int;
delete p1;      // correct
// delete[] p1; // undefined
// free(p1);    // undefined

int* p2 = new int[10];
delete[] p2;    // correct
// delete p2;   // undefined
// free(p2);    // undefined

int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3);       // correct
// delete p3;   // undefined
// delete[] p3; // undefined
```

Dichos problemas se pueden evitar evitando completamente `malloc` y programas `free` en C ++, prefiriendo los punteros inteligentes de la biblioteca estándar sobre `new` y `delete` en bruto, y prefiriendo `std::vector` y `std::string` sobre `new` y `delete[]` .

Accediendo a un objeto como el tipo equivocado

En la mayoría de los casos, es ilegal acceder a un objeto de un tipo como si fuera un tipo diferente (sin tener en cuenta los calificadores cv). Ejemplo:

```
float x = 42;
int y = reinterpret_cast<int&>(x);
```

El resultado es un comportamiento indefinido.

Hay algunas excepciones a esta regla *estricta de aliasing* :

- Se puede acceder a un objeto de tipo de clase como si fuera de un tipo que es una clase base del tipo de clase real.
- Se puede acceder a cualquier tipo como `char` o `unsigned char` , pero lo contrario no es cierto: no se puede acceder a una matriz char como si fuera un tipo arbitrario.
- Se puede acceder a un tipo entero con signo como el tipo sin signo correspondiente y *viceversa* .

Una regla relacionada es que si se llama a una función miembro no estática en un objeto que en

realidad no tiene el mismo tipo que la clase que define la función, o una clase derivada, entonces ocurre un comportamiento indefinido. Esto es cierto incluso si la función no accede al objeto.

```
struct Base {
};
struct Derived : Base {
    void f() {}
};
struct Unrelated {};
Unrelated u;
Derived& r1 = reinterpret_cast<Derived&>(u); // ok
r1.f();                                      // UB
Base b;
Derived& r2 = reinterpret_cast<Derived&>(b); // ok
r2.f();                                      // UB
```

Desbordamiento de punto flotante

Si una operación aritmética que produce un tipo de punto flotante produce un valor que no está en el rango de valores representables del tipo de resultado, el comportamiento no está definido de acuerdo con el estándar C ++, pero puede definirse por otros estándares que la máquina pueda cumplir. tales como IEEE 754.

```
float x = 1.0;
for (int i = 0; i < 10000; i++) {
    x *= 10.0; // will probably overflow eventually; undefined behavior
}
```

Llamando (Puro) a los Miembros Virtuales del Constructor o Destructor

La Norma (10.4) establece:

> Las funciones miembro se pueden llamar desde un constructor (o destructor) de una clase abstracta; el efecto de hacer una llamada virtual (10.3) a una función virtual pura directa o indirectamente para el objeto que se está creando (o destruyendo) desde tal constructor (o destructor) no está definido.

De manera más general, algunas autoridades de C ++, por ejemplo, Scott Meyers, sugieren nunca llamar a funciones virtuales (incluso no puras) de constructores y constructores.

Considere el siguiente ejemplo, modificado desde el enlace anterior:

```
class transaction
{
public:
    transaction(){ log_it(); }
    virtual void log_it() const = 0;
};

class sell_transaction : public transaction
{
public:
    virtual void log_it() const { /* Do something */ }
```

```
};
```

Supongamos que creamos un objeto `sell_transaction` :

```
sell_transaction s;
```

Esto implícitamente llama al constructor de `sell_transaction` , que primero llama al constructor de la `transaction` . Sin embargo, cuando se llama al constructor de la `transaction` , el objeto todavía no es del tipo `sell_transaction` , sino más bien del tipo `transaction` .

En consecuencia, la llamada en `transaction::transaction()` a `log_it` , no hará lo que pueda parecer algo intuitivo, es decir, call `sell_transaction::log_it` .

- Si `log_it` es puramente virtual, como en este ejemplo, el comportamiento no está definido.
- Si `log_it` no es virtual puro, se `transaction::log_it` .

Eliminar un objeto derivado a través de un puntero a una clase base que no tiene un destructor virtual.

```
class base { };
class derived: public base { };

int main() {
    base* p = new derived();
    delete p; // The is undefined behavior!
}
```

En la sección [expr.delete] §5.3.5 / 3, el estándar dice que si se llama a `delete` en un objeto cuyo tipo estático no tiene un destructor `virtual` :

> Si el tipo estático del objeto que se va a eliminar es diferente de su tipo dinámico, el tipo estático será una clase base del tipo dinámico del objeto que se eliminará y el tipo estático tendrá un destructor virtual o el comportamiento es indefinido.

Este es el caso, independientemente de la pregunta sobre si la clase derivada agregó algún miembro de datos a la clase base.

Accediendo a una referencia colgante

Es ilegal acceder a una referencia a un objeto que ha quedado fuera del alcance o ha sido destruido. Se dice que dicha referencia está *colgando* ya que ya no se refiere a un objeto válido.

```
#include <iostream>
int& getX() {
    int x = 42;
    return x;
}
int main() {
    int& r = getX();
```

```
    std::cout << r << "\n";
}
```

En este ejemplo, la variable local `x` queda fuera del alcance cuando retorna `getX` . (Tenga en cuenta que la *extensión de vida útil* no puede extender la vida útil de una variable local más allá del alcance del bloque en el que está definida). Por lo tanto, `r` es una referencia pendiente. Este programa tiene un comportamiento indefinido, aunque puede parecer que funciona e imprime `42` en algunos casos.

Extendiendo el espacio de nombres `std` o` posix`

El estándar (17.6.4.2.1 / 1) generalmente prohíbe extender el `std` nombres estándar :

> El comportamiento de un programa en C ++ no está definido si agrega declaraciones o definiciones al espacio de nombres estándar o a un espacio de nombres dentro del espacio de nombres estándar, a menos que se especifique lo contrario.

Lo mismo ocurre con `posix` (17.6.4.2.2 / 1):

> El comportamiento de un programa de C ++ no está definido si agrega declaraciones o definiciones al espacio de nombres posix o a un espacio de nombres dentro del espacio de nombres posix a menos que se especifique lo contrario.

Considera lo siguiente:

```
#include <algorithm>

namespace std
{
    int foo(){}
}
```

Nada en el estándar prohíbe el `algorithm` (o uno de los encabezados que incluye) la definición de la misma definición, por lo que este código violaría la Regla de una definición .

Entonces, en general, esto está prohibido. Sin embargo, hay excepciones específicas permitidas . Tal vez lo más útil es que está permitido agregar especializaciones para los tipos definidos por el usuario. Entonces, por ejemplo, supongamos que su código tiene

```
class foo
{
    // Stuff
};
```

Entonces lo siguiente está bien

```
namespace std
{
    template<>
    struct hash<foo>
    {
```

```
    public:
        size_t operator()(const foo &f) const;
    };
}
```

Desbordamiento durante la conversión hacia o desde el tipo de punto flotante

Si, durante la conversión de:

- un tipo entero a un tipo de punto flotante,
- un tipo de punto flotante a un tipo entero, o
- un tipo de punto flotante a un tipo de punto flotante más corto,

El valor de origen está fuera del rango de valores que se pueden representar en el tipo de destino, el resultado es un comportamiento indefinido. Ejemplo:

```
double x = 1e100;
int y = x; // int probably cannot hold numbers that large, so this is UB
```

Conversión estática de base a derivada no válida

Si se utiliza `static_cast` para convertir un puntero (referencia de referencia) en una clase base en un puntero (referencia de referencia) en una clase derivada, pero el operando no apunta (referido de referencia) a un objeto del tipo de clase derivada, el comportamiento es indefinido. Ver Base a conversión derivada .

Función de llamada a través del tipo de puntero de función no coincidente

Para llamar a una función mediante un puntero de función, el tipo de puntero de función debe coincidir exactamente con el tipo de función. De lo contrario, el comportamiento es indefinido. Ejemplo:

```
int f();
void (*p)() = reinterpret_cast<void(*)()>(f);
p(); // undefined
```

Modificar un objeto const

Cualquier intento de modificar un objeto `const` produce un comportamiento indefinido. Esto se aplica a las variables `const` , miembros de objetos `const` y miembros de clase declarados `const` . (Sin embargo, un `mutable` miembro de un `const` objeto no es `const` .)

Tal intento se puede hacer a través de `const_cast` :

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

Un compilador usualmente alineará el valor de un objeto `const int` , por lo que es posible que este código compile e imprima `123` . Los compiladores también pueden colocar valores `const` objetos en la memoria de solo lectura, por lo que puede ocurrir una falla de segmentación. En cualquier caso, el comportamiento no está definido y el programa puede hacer cualquier cosa.

El siguiente programa oculta un error mucho más sutil:

```
#include <iostream>

class Foo* instance;

class Foo {
  public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
  private:
    Foo(int x, Foo*& this_ref): m_x(x) {
        this_ref = this;
    }
    int m_x;
    friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
    std::cout << foo.get_x() << '\n';
}
```

En este código, `getFoo` crea un singleton de tipo `const Foo` y su miembro `m_x` se inicializa en `123` . Luego se llama `do_evil` y el valor de `foo.m_x` aparentemente se cambia a 456. ¿Qué salió mal?

A pesar de su nombre, `do_evil` no hace nada particularmente malo; todo lo que hace es llamar a un setter a través de un `Foo*` . Pero ese puntero apunta a un objeto `const Foo` aunque no se usó `const_cast` . Este puntero se obtuvo a través del constructor de `Foo` . Un objeto `const` no se convierte en `const` hasta que se completa su inicialización, por `this` tiene el tipo `Foo*` , no `const Foo*` , dentro del constructor.

Por lo tanto, el comportamiento indefinido ocurre aunque no hay construcciones obviamente peligrosas en este programa.

Acceso a miembro inexistente a través de puntero a miembro

Cuando se accede a un miembro no estático de un objeto a través de un puntero a miembro, si el objeto no contiene realmente al miembro indicado por el puntero, el comportamiento no está

definido. (Este puntero a miembro se puede obtener a través de `static_cast`).

```
struct Base { int x; };
struct Derived : Base { int y; };
int Derived::*pdy = &Derived::y;
int Base::*pby = static_cast<int Base::*>(pdy);

Base* b1 = new Derived;
b1->*pby = 42; // ok; sets y in Derived object to 42
Base* b2 = new Base;
b2->*pby = 42; // undefined; there is no y member in Base
```

Conversión derivada a base no válida para punteros a miembros

Cuando `static_cast` se usa para convertir `TD::*` a `TB::*` , el miembro apuntado debe pertenecer a una clase que sea una clase base o una clase derivada de `B` De lo contrario el comportamiento es indefinido. Consulte Conversión derivada a base para punteros a miembros.

Aritmética de puntero no válido

Los siguientes usos de la aritmética de punteros provocan un comportamiento indefinido:

- Suma o resta de un entero, si el resultado no pertenece al mismo objeto de matriz que el operando de puntero. (Aquí, se considera que el elemento uno más allá del final todavía pertenece a la matriz.)

  ```
  int a[10];
  int* p1 = &a[5];
  int* p2 = p1 + 4; // ok; p2 points to a[9]
  int* p3 = p1 + 5; // ok; p2 points to one past the end of a
  int* p4 = p1 + 6; // UB
  int* p5 = p1 - 5; // ok; p2 points to a[0]
  int* p6 = p1 - 6; // UB
  int* p7 = p3 - 5; // ok; p7 points to a[5]
  ```

- Resta dos punteros si ambos no pertenecen al mismo objeto de matriz. (De nuevo, se considera que el elemento uno más allá del final pertenece a la matriz.) La excepción es que se pueden restar dos punteros nulos, lo que arroja 0.

  ```
  int a[10];
  int b[10];
  int *p1 = &a[8], *p2 = &a[3];
  int d1 = p1 - p2; // yields 5
  int *p3 = p1 + 2; // ok; p3 points to one past the end of a
  int d2 = p3 - p2; // yields 7
  int *p4 = &b[0];
  int d3 = p4 - p1; // UB
  ```

- Resta dos punteros si el resultado se desborda `std::ptrdiff_t` .

- Cualquier aritmética de punteros en la que el tipo de punto del operando no coincida con el tipo dinámico del objeto apuntado (ignorando la calificación cv). De acuerdo con el estándar,

"[en] en particular, un puntero a una clase base no se puede usar para la aritmética de punteros cuando la matriz contiene objetos de un tipo de clase derivada".

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
Base* p1 = &a[1];           // ok
Base* p2 = p1 + 1;          // UB; p1 points to Derived
Base* p3 = p1 - 1;          // likewise
Base* p4 = &a[2];           // ok
auto p5 = p4 - p1;          // UB; p4 and p1 point to Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // ok; cv-qualifiers don't matter
```

Desplazando por un número de posiciones no válido

Para el operador de cambio incorporado, el operando derecho debe ser no negativo y estrictamente menor que el ancho de bits del operando izquierdo promovido. De lo contrario, el comportamiento es indefinido.

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0;  // ok
const int d = a << 32; // UB if int is 32 bits or less
const int e = a >> 32; // also UB if int is 32 bits or less
const signed char f = 'x';
const int g = f << 10; // ok even if signed char is 10 bits or less;
                       // int must be at least 16 bits
```

Volviendo de una función [[noreturn]]

C ++ 11

Ejemplo de la Norma, [dcl.attr.noreturn]:

```
[[ noreturn ]] void f() {
  throw "error"; // OK
}
[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
  if (i > 0)
    throw "positive";
}
```

Destruyendo un objeto que ya ha sido destruido.

En este ejemplo, se invoca explícitamente un destructor para un objeto que luego se destruirá automáticamente.

```
struct S {
    ~S() { std::cout << "destroying S\n"; }
};
int main() {
```

```
    S s;
    s.~S();
} // UB: s destroyed a second time here
```

Se produce un problema similar cuando se `std::unique_ptr<T>` un `std::unique_ptr<T>` para apuntar a una `T` con duración de almacenamiento automática o estática.

```
void f(std::unique_ptr<S> p);
int main() {
    S s;
    std::unique_ptr<S> p(&s);
    f(std::move(p)); // s destroyed upon return from f
}                    // UB: s destroyed
```

Otra forma de destruir un objeto dos veces es tener dos `shared_ptr` gestionen el objeto sin compartir la propiedad entre ellos.

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);
int main() {
    S* p = new S;
    // I want to pass the same object twice...
    std::shared_ptr<S> sp1(p);
    std::shared_ptr<S> sp2(p);
    f(sp1, sp2);
} // UB: both sp1 and sp2 will destroy s separately
// NB: this is correct:
// std::shared_ptr<S> sp(p);
// f(sp, sp);
```

Recursión de plantilla infinita

Ejemplo de la Norma, [temp.inst] / 17:

```
template<class T> class X {
    X<T>* p; // OK
    X<T*> a; // implicit generation of X<T> requires
             // the implicit instantiation of X<T*> which requires
             // the implicit instantiation of X<T**> which ...
};
```

Lea Comportamiento indefinido en línea:
https://riptutorial.com/es/cplusplus/topic/1812/comportamiento-indefinido

Capítulo 23: Comportamiento no especificado

Observaciones

Si el comportamiento de una construcción no se especifica, entonces el estándar establece algunas restricciones en el comportamiento, pero deja cierta libertad a la implementación, que *no* es necesaria para documentar lo que sucede en una situación determinada. Contrasta con el comportamiento definido por la implementación, en el que *se* requiere que la implementación documente lo que sucede, y el comportamiento indefinido, en el que puede ocurrir cualquier cosa.

Examples

Orden de inicialización de globales a través de TU

Mientras que dentro de una unidad de traducción, se especifica el orden de inicialización de las variables globales, el orden de inicialización entre unidades de traducción no está especificado.

Así programa con los siguientes archivos

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

```
int main() {}
```

podría producir como salida:

```
foobar
```

o

```
barfoo
```

Eso puede llevar a *Fiasco Orden de Inicialización Estática* .

Valor de una enumeración fuera de rango

Si una enumeración de ámbito se convierte en un tipo integral que es demasiado pequeño para mantener su valor, el valor resultante no se especifica. Ejemplo:

```
enum class E {
    X = 1,
    Y = 1000,
};
// assume 1000 does not fit into a char
char c1 = static_cast<char>(E::X); // c1 is 1
char c2 = static_cast<char>(E::Y); // c2 has an unspecified value
```

Además, si un entero se convierte en una enumeración y el valor del entero está fuera del rango de los valores de la enumeración, el valor resultante no se especifica. Ejemplo:

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};
Color c = static_cast<Color>(4);
```

Sin embargo, en el siguiente ejemplo, el comportamiento *no* está *sin* especificar, ya que el valor de origen está dentro del *rango* de la enumeración, aunque es desigual para todos los enumeradores:

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};
Scale s = static_cast<Scale>(3);
```

Aquí `s` tendrá el valor 3 y será igual a `ONE` , `TWO` y `FOUR` .

Reparto estático a partir de un valor falso *

Si un valor `void*` se convierte en un puntero al tipo de objeto, `T*` , pero no se alinea correctamente para `T` , el valor del puntero resultante no se especifica. Ejemplo:

```
// Suppose that alignof(int) is 4
int x = 42;
void* p1 = &x;
// Do some pointer arithmetic...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

El valor de `p3` no está especificado porque `p2` no puede apuntar a un objeto de tipo `int` ; su valor no es una dirección correctamente alineada.

Resultado de algunas conversiones reinterpret_cast

El resultado de un `reinterpret_cast` de un tipo de puntero de función a otro, o un tipo de referencia de función a otro, no está especificado. Ejemplo:

```
int f();
auto fp = reinterpret_cast<int(*)(int)>(&f); // fp has unspecified value
```

C ++ 03

El resultado de un `reinterpret_cast` de un tipo de puntero de objeto a otro, o un tipo de referencia de objeto a otro, no se especifica. Ejemplo:

```
int x = 42;
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

Sin embargo, con la mayoría de los compiladores, esto era equivalente a `static_cast<char*>(static_cast<void*>(&x))` por lo que el puntero resultante `p` apuntaba al primer byte de `x` . Esto se hizo el comportamiento estándar en C ++ 11. Ver tipo de conversión de puntos para más detalles.

Resultado de algunas comparaciones de punteros

Si se comparan dos punteros utilizando `<` , `>` , `<=` o `>=` , el resultado no se especifica en los siguientes casos:

- Los punteros apuntan a diferentes matrices. (Un objeto sin matriz se considera una matriz de tamaño 1).

```
int x;
int y;
const bool b1 = &x < &y;            // unspecified
int a[10];
const bool b2 = &a[0] < &a[1];      // true
const bool b3 = &a[0] < &x;         // unspecified
const bool b4 = (a + 9) < (a + 10); // true
                                    // note: a+10 points past the end of the array
```

- Los punteros apuntan al mismo objeto, pero a los miembros con un control de acceso diferente.

```
class A {
  public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x comes before y
    bool f2() { return &x < &z; } // unspecified
  private:
    int z;
};
```

Espacio ocupado por una referencia.

Una referencia no es un objeto y, a diferencia de un objeto, no se garantiza que ocupe algunos bytes contiguos de memoria. El estándar deja sin especificar si una referencia requiere algún almacenamiento. Una serie de características del lenguaje conspiran para hacer que sea imposible examinar de manera portátil cualquier almacenamiento que la referencia pueda ocupar:

- Si se aplica `sizeof` a una referencia, devuelve el tamaño del tipo referenciado, por lo que no proporciona información sobre si la referencia ocupa algún almacenamiento.
- Las matrices de referencias son ilegales, por lo que no es posible examinar las direcciones de dos elementos consecutivos de una referencia hipotética de matrices para determinar el tamaño de una referencia.
- Si se toma la dirección de una referencia, el resultado es la dirección del referente, por lo que no podemos obtener un puntero a la referencia en sí.
- Si una clase tiene un miembro de referencia, el intento de extraer la dirección de ese miembro usando `offsetof` produce un comportamiento indefinido ya que dicha clase no es una clase de diseño estándar.
- Si una clase tiene un miembro de referencia, la clase ya no es un diseño estándar, por lo tanto, los intentos de acceder a los datos utilizados para almacenar los resultados de referencia en un comportamiento no definido o no especificado.

En la práctica, en algunos casos, una variable de referencia puede implementarse de manera similar a una variable de puntero y, por lo tanto, ocupar la misma cantidad de almacenamiento que un puntero, mientras que en otros casos una referencia puede no ocupar ningún espacio ya que puede optimizarse. Por ejemplo, en:

```
void f() {
    int x;
    int& r = x;
    // do something with r
}
```

el compilador es libre de simplemente tratar `r` como un alias para `x` y reemplazar todas las apariciones de `r` en el resto de la función `f` con `x` , y no asignar ningún almacenamiento para mantener `r` .

Orden de evaluacion de argumentos de funcion.

Si una función tiene múltiples argumentos, no se especifica en qué orden se evalúan. El siguiente código podría imprimir `x = 1, y = 2` o `x = 2, y = 1` pero no se especifica cuál.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
}
int get_val() {
    static int x = 0;
    return ++x;
}
int main() {
    f(get_val(), get_val());
```

```
}
```

C ++ 17

En C ++ 17, el orden de evaluación de los argumentos de la función permanece sin especificar.

Sin embargo, cada argumento de función se evalúa por completo, y se garantiza la evaluación del objeto llamante antes de que lo sean los argumentos de función.

```
struct from_int {
  from_int(int x) { std::cout << "from_int (" << x << ")\n"; }
};
int make_int(int x){ std::cout << "make_int (" << x << ")\n"; return x; }


void foo(from_int a, from_int b) {
}
void bar(from_int a, from_int b) {
}

auto which_func(bool b){
  std::cout << b?"foo":"bar" << "\n";
  return b?foo:bar;
}

int main(int argc, char const*const* argv) {
  which_func( true )( make_int(1), make_int(2) );
}
```

esto debe imprimir:

```
bar
make_int(1)
from_int(1)
make_int(2)
from_int(2)
```

o

```
bar
make_int(2)
from_int(2)
make_int(1)
from_int(1)
```

que *no se* imprima `bar` después de cualquiera de la `make` o `from` 's, y que no se imprima:

```
bar
make_int(2)
make_int(1)
from_int(2)
from_int(1)
```

o similar. Antes de C ++ 17, la `bar` impresión después de `make_int` s era legal, al igual que hacer

ambos `make_int` s antes de hacer cualquier `from_int` s.

Estado movido de la mayoría de las clases de biblioteca estándar

C ++ 11

Todos los contenedores de biblioteca estándar se dejan en un estado *válido pero no especificado* después de ser movidos. Por ejemplo, en el siguiente código, `v2` contendrá `{1, 2, 3, 4}` después del movimiento, pero no se garantiza que `v1` esté vacío.

```
int main() {
    std::vector<int> v1{1, 2, 3, 4};
    std::vector<int> v2 = std::move(v1);
}
```

Algunas clases tienen un estado movido desde exactamente definido. El caso más importante es el de `std::unique_ptr<T>` , que se garantiza que será nulo después de ser movido.

Lea Comportamiento no especificado en línea:
https://riptutorial.com/es/cplusplus/topic/4939/comportamiento-no-especificado

Capítulo 24: Concurrencia con OpenMP

Introducción

Este tema cubre los conceptos básicos de la concurrencia en C ++ utilizando OpenMP. OpenMP se documenta con más detalle en la etiqueta OpenMP .

Paralelismo o concurrencia implica la ejecución de código al mismo tiempo.

Observaciones

OpenMP no requiere encabezados ni bibliotecas especiales, ya que es una característica de compilador integrada. Sin embargo, si usa alguna de las funciones de la API de OpenMP como `omp_get_thread_num()` , deberá incluir `omp.h` su biblioteca.

Las instrucciones `pragma` OpenMP se ignoran cuando la opción OpenMP no está habilitada durante la compilación. Es posible que desee consultar la opción del compilador en el manual de su compilador.

- GCC utiliza `-fopenmp`
- Clang usa `-fopenmp`
- MSVC usa `/openmp`

Examples

OpenMP: Secciones paralelas

Este ejemplo ilustra los conceptos básicos de la ejecución de secciones de código en paralelo.

Como OpenMP es una característica de compilador incorporada, funciona en cualquier compilador compatible sin incluir bibliotecas. Si lo desea, puede incluir `omp.h` si desea usar cualquiera de las funciones de la API openMP.

Código de muestra

```
std::cout << "begin ";
//    This pragma statement hints the compiler that the
//    contents within the { } are to be executed in as
//    parallel sections using openMP, the compiler will
//    generate this chunk of code for parallel execution
#pragma omp parallel sections
{
    //    This pragma statement hints the compiler that
    //    this is a section that can be executed in parallel
    //    with other section, a single section will be executed
    //    by a single thread.
    //    Note that it is "section" as opposed to "sections" above
    #pragma omp section
```

```
    {
        std::cout << "hello " << std::endl;
        /** Do something **/
    }
    #pragma omp section
    {
        std::cout << "world " << std::endl;
        /** Do something **/
    }
}
//    This line will not be executed until all the
//    sections defined above terminates
std::cout << "end" << std::endl;
```

Salidas

Este ejemplo produce 2 salidas posibles y depende del sistema operativo y el hardware. La salida también ilustra un problema de **condición de carrera** que se produciría a partir de dicha implementación.

SALIDA A	SALIDA B
empieza hola fin mundo	comienza el mundo hola fin

OpenMP: Secciones paralelas

Este ejemplo muestra cómo ejecutar trozos de código en paralelo

```
std::cout << "begin ";
//    Start of parallel sections
#pragma omp parallel sections
{
    //    Execute these sections in parallel
    #pragma omp section
    {
        ... do something ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "forever ";
    }
}
//    end of parallel sections
std::cout << "end";
```

Salida

- comienza hola mundo para siempre termina

- comienza el mundo hola para siempre termina
- comience hola para siempre fin del mundo
- comienza por siempre hola fin del mundo

Como el orden de ejecución no está garantizado, puede observar cualquiera de los resultados anteriores.

OpenMP: Parallel For Loop

Este ejemplo muestra cómo dividir un bucle en partes iguales y ejecutarlas en paralelo.

```
//    Splits element vector into element.size() / Thread Qty
//    and allocate that range for each thread.
#pragma omp parallel for
for    (size_t i = 0; i < element.size(); ++i)
    element[i] = ...

//    Example Allocation (100 element per thread)
//    Thread 1 : 0 ~ 99
//    Thread 2 : 100 ~ 199
//    Thread 2 : 200 ~ 299
//    ...

//    Continue process
//    Only when all threads completed their allocated
//    loop job
...
```

* Tenga mucho cuidado de no modificar el tamaño del vector utilizado en paralelo para los bucles, ya que los **índices de rango asignado no se actualizan automáticamente** .

OpenMP: Recopilación paralela / Reducción

Este ejemplo ilustra un concepto para realizar una reducción o recopilación utilizando `std::vector` y OpenMP.

Suponemos que tenemos un escenario en el que queremos que varios subprocesos nos ayuden a generar un montón de cosas, `int` se usa aquí para simplificar y se puede reemplazar con otros tipos de datos.

Esto es particularmente útil cuando necesita combinar resultados de esclavos para evitar fallas de segmentos o violaciones de acceso a la memoria y no desea usar bibliotecas o bibliotecas personalizadas de contenedores de sincronización.

```
//    The Master vector
//    We want a vector of results gathered from slave threads
std::vector<int> Master;

//    Hint the compiler to parallelize this { } of code
//    with all available threads (usually the same as logical processor qty)
#pragma omp parallel
{
    //    In this area, you can write any code you want for each
```

```
    //    slave thread, in this case a vector to hold each of their results
    //    We don't have to worry about how many threads were spawn or if we need
    //    to repeat this declaration or not.
    std::vector<int> Slave;

    //    Tell the compiler to use all threads allocated for this parallel region
    //    to perform this loop in parts. Actual load appx = 1000000 / Thread Qty
    //    The nowait keyword tells the compiler that the slave threads don't
    //    have to wait for all other slaves to finish this for loop job
    #pragma omp for nowait
    for (size_t i = 0; i < 1000000; ++i
    {
        /* Do something */
        ....
        Slave.push_back(...);
    }

    //    Slaves that finished their part of the job
    //    will perform this thread by thread one at a time
    //    critical section ensures that only 0 or 1 thread performs
    //    the { } at any time
    #pragma omp critical
    {
        //    Merge slave into master
        //    use move iterators instead, avoid copy unless
        //    you want to use it for something else after this section
        Master.insert(Master.end(),
                      std::make_move_iterator(Slave.begin()),
                      std::make_move_iterator(Slave.end()));
    }
}

//    Have fun with Master vector
...
```

Lea Concurrencia con OpenMP en línea:
https://riptutorial.com/es/cplusplus/topic/8222/concurrencia-con-openmp

Capítulo 25: Const Correccion

Sintaxis

- class ClassOne {public: bool non_modifying_member_function () const {/ * ... * /}};
- int ClassTwo :: non_modifying_member_function () const {/ * ... * /}
- void ClassTwo :: modifying_member_function () {/ * ... * /}
- char non_param_modding_func (const ClassOne & one, const ClassTwo * two) {/ * ... * /}
- float parameters_modifying_function (ClassTwo & one, ClassOne * two) {/ * ... * /}
- short ClassThree :: non_modding_non_param_modding_f (const ClassOne &) const {/ * ... * /}

Observaciones

`const` corrección de `const` es una herramienta de solución de problemas muy útil, ya que le permite al programador determinar rápidamente qué funciones podrían estar modificando inadvertidamente el código. También evita que los errores involuntarios, como el que se muestra en `Const Correct Function Parameters` , se compilen correctamente y pasen desapercibidos.

Es mucho más fácil diseñar una clase para la corrección `const` , que luego agregar la corrección `const` a una clase preexistente. Si es posible, diseñar cualquier clase que *puede* ser `const` correcta para que *sea* `const` correcta, para salvar a sí mismo ya otros la molestia de tarde modificándolo.

Tenga en cuenta que esto también puede aplicarse a la corrección `volatile` si es necesario, con las mismas reglas que para la corrección `const` , pero esto se usa con mucha menos frecuencia.

Refrences:

ISO_CPP

Véndeme en constancia correcta

Tutorial de C ++

Examples

Los basicos

`const` corrección `const` es la práctica de diseñar código de modo que solo el código que *necesita* modificar una instancia *pueda* modificar una instancia (es decir, tenga acceso de escritura) y, a la inversa, que cualquier código que no necesite modificar una instancia no pueda hacerlo entonces (es decir, solo tiene acceso de lectura). Esto evita que la instancia se modifique involuntariamente, lo que hace que el código sea menos propenso a errores, y documenta si el código está destinado a cambiar el estado de la instancia o no. También permite que las instancias se traten como `const` siempre que no necesiten ser modificadas, o definidas como `const`

si no es necesario cambiarlas después de la inicialización, sin perder ninguna funcionalidad.

Esto se hace dando a los miembros las funciones `const` CV-qualifiers , y haciendo `const` parámetros de puntero / referencia, excepto en el caso de que necesiten acceso de escritura.

```
class ConstCorrectClass {
    int x;

  public:
    int getX() const { return x; } // Function is const: Doesn't modify instance.
    void setX(int i) { x = i; }    // Not const: Modifies instance.
};

// Parameter is const: Doesn't modify parameter.
int const_correct_reader(const ConstCorrectClass& c) {
    return c.getX();
}

// Parameter isn't const: Modifies parameter.
void const_correct_writer(ConstCorrectClass& c) {
    c.setX(42);
}

const ConstCorrectClass invariant; // Instance is const: Can't be modified.
ConstCorrectClass         variant; // Instance isn't const: Can be modified.

// ...

const_correct_reader(invariant); // Good.   Calling non-modifying function on const instance.
const_correct_reader(variant);   // Good.   Calling non-modifying function on modifiable
instance.

const_correct_writer(variant);   // Good.   Calling modifying function on modifiable instance.
const_correct_writer(invariant); // Error.  Calling modifying function on const instance.
```

Debido a la naturaleza de la corrección de const, esto comienza con las funciones de los miembros de la clase y se abre camino hacia afuera; Si intenta llamar a una función miembro no `const` desde una instancia `const` o desde una instancia no `const` tratada como `const` , el compilador le dará un error sobre la pérdida de calificadores cv.

Diseño correcto de la clase de Const

En una clase `const` , todas las funciones miembro que no cambian el estado lógico tienen `this` cv calificado como `const` , lo que indica que no modifican el objeto (aparte de `mutable` campos `mutable` , que se pueden modificar libremente incluso en casos `const`); si una función calificada para cv `const` devuelve una referencia, esa referencia también debe ser una `const` . Esto permite que se los llame en instancias constantes y no calificadas para CV, ya que una `const T*` es capaz de vincularse a una `T*` o una `const T*` . Esto, a su vez, permite que las funciones declaren sus parámetros pasados por referencia como `const` cuando no necesitan ser modificados, sin perder ninguna funcionalidad.

Además, en una clase correcta `const` , todos los parámetros de función pasados por referencia serán `const` correctos, como se describe en `Const Correct Function Parameters` , de modo que solo pueden modificarse cuando la función *necesita* modificarlos explícitamente.

Primero, echemos un vistazo a `this` calificadores cv:

```
// Assume class Field, with member function "void insert_value(int);".

class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(Field& f); // Modifies.

    Field& getField();        // Might modify.  Also exposes member as non-const reference,
                              //  allowing indirect modification.
    void setField(Field& f);  // Modifies.

    void doSomething(int i);  // Might modify.
    void doNothing();         // Might modify.
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // Modifies.
Field& ConstIncorrect::getField() { return fld; }    // Doesn't modify.
void ConstIncorrect::setField(Field& f) { fld = f; } // Modifies.
void ConstIncorrect::doSomething(int i) {            // Modifies.
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {}                  // Doesn't modify.


class ConstCorrectCVQ {
    Field fld;

  public:
    ConstCorrectCVQ(Field& f);     // Modifies.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(Field& f);       // Modifies.

    void doSomething(int i);       // Modifies.
    void doNothing() const;        // Doesn't modify.
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const  {}

// This won't work.
// No member functions can be called on const ConstIncorrect instances.
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// But this will.
// getField() and doNothing() can be called on const ConstCorrectCVQ instances.
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
```

```
}
```

Entonces podemos combinar esto con `Const Correct Function Parameters` , causando la clase sea plenamente `const` -correct.

```
class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f);  // Modifies instance.  Doesn't modify parameter.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(const Field& f); // Modifies instance.  Doesn't modify parameter.

    void doSomething(int i);       // Modifies.  Doesn't modify parameter (passed by value).
    void doNothing() const;        // Doesn't modify.
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}
```

Esto también se puede combinar con la sobrecarga basada en `const` , en el caso de que queramos un comportamiento si la instancia es `const` , y un comportamiento diferente si no lo es; un uso común para esto es constainers que proporcionan accesores que solo permiten modificaciones si el contenedor en sí no es `const` .

```
class ConstCorrectContainer {
    int arr[5];

  public:
    // Subscript operator provides read access if instance is const, or read/write access
    // otherwise.
          int& operator[](size_t index)       { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};
```

Esto es comúnmente utilizado en la biblioteca estándar, con la mayoría de los recipientes proporcionando sobrecargas para tomar `const` Ness en cuenta.

Constar los parámetros de función correcta

En una función de corrección `const` , todos los parámetros pasados por referencia se marcan como `const` menos que la función los modifique directa o indirectamente, lo que evita que el programador cambie inadvertidamente algo que no pretendían cambiar. Esto permite que la función de tomar tanto `const` y los casos no-CV-cualificado, ya su vez, hace que la instancia está

`this` a ser de tipo `const T*` cuando un miembro de la función se llama, donde `T` es el tipo de clase.

```
struct Example {
    void func()       { std::cout << 3 << std::endl; }
    void func() const { std::cout << 5 << std::endl; }
};

void const_incorrect_function(Example& one, Example* two) {
    one.func();
    two->func();
}

void const_correct_function(const Example& one, const Example* two) {
    one.func();
    two->func();
}

int main() {
    Example a, b;
    const_incorrect_function(a, &b);
    const_correct_function(a, &b);
}

// Output:
3
3
5
5
```

Si bien los efectos de esto son menos evidentes que los de `const` diseño de la clase correcta (en ese `const` funciones -correct y `const` clases -incorrect causará errores de compilación, mientras que `const` clases -correct y `const` funciones -incorrect compilará correctamente), `const` correcta las funciones detectarán una gran cantidad de errores que las funciones `const` incorrectas dejarán pasar, como la que se muestra a continuación. [Nota, sin embargo, que un `const` función -incorrect *causará* errores de compilación si se aprueba una `const` ejemplo, cuando se espera que un no `const` uno.]

```
// Read value from vector, then compute & return a value.
// Caches return values for speed.
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // Cache values, for future use.
    // Once a return value has been calculated, it's cached & its index is registered.
    static std::vector<T> vals = {};

    int v_ind = h.get_index();               // Current working index for v.
    int vals_ind = h.get_cache_index(v_ind); // Will be -1 if cache index isn't registered.

    if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
        return vals[h.get_cache_index(v_ind)];
    }

    T temp = v[v_ind];

    temp -= h.poll_device();
    temp *= h.obtain_random();
    temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);
```

```
    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Oops.  Should've been accessing vals.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}

// Const correct version.  Is identical to above version, so most of it shall be skipped.
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Error: discards qualifiers.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}
```

Constancia de la corrección como documentación

Una de las cosas más útiles acerca de la corrección `const` es que sirve como una forma de documentar el código, proporcionando ciertas garantías al programador y otros usuarios. Estas garantías son impuestas por el compilador debido a la `const` , con una falta de `const` a su vez indica que el código no las proporciona.

Funciones de miembros calificados para CV `const` :

- Se puede asumir que cualquier función miembro que sea `const` tiene intención de leer la instancia, y:
    - No modificará el estado lógico de la instancia a la que se llama. Por lo tanto, no deben modificar ninguna variable miembro de la instancia a la que se llama, excepto las variables `mutable` .
    - No debe llamar a ninguna *otra* función que pueda modificar ninguna variable miembro de la instancia, excepto las variables `mutable` .
- A la inversa, se puede asumir que cualquier función miembro que no sea `const` tiene la intención de modificar la instancia, y:
    - Puede o no puede modificar el estado lógico.
    - Puede o no llamar a otras funciones que modifican el estado lógico.

Esto se puede usar para hacer suposiciones sobre el estado del objeto después de que se llame a cualquier función miembro dada, incluso sin ver la definición de esa función:

```
// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

  public:
    // Constructor clearly changes logical state.  No assumptions necessary.
    ConstMemberFunctions(int v = 0);

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call squared_calc() or bad_func().
    int calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or bad_func().
    int squared_calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or squared_calc().
    void bad_func() const;

    // We can assume this function changes logical state, and may or may not call
    //  calc(), squared_calc(), or bad_func().
    void set_val(int v);
};
```

Debido a las reglas `const` , estos supuestos serán de hecho aplicados por el compilador.

```
// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0*/)
  : cache(0), val(v), state_changed(true) {}

// Our assumption was correct.
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }

    return cache;
}

// Our assumption was correct.
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers.
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}

// Our assumption was correct.
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
```

```
        state_changed = true;
    }
}
```

Parámetros de la función `const` :

- Se puede suponer que cualquier función con uno o más parámetros que sean `const` tiene intención de leer esos parámetros, y:
    - No modificará esos parámetros ni llamará a ninguna función miembro que los modifique.
    - No debe pasar esos parámetros a ninguna *otra* función que los modifique y / o llame a ninguna función miembro que los modifique.
- A la inversa, se puede suponer que cualquier función con uno o más parámetros que no son `const` tiene la intención de modificar esos parámetros, y:
    - Puede o no modificar esos parámetros, o llamar a cualquier función miembro que los modifique.
    - Pueden o no pasar esos parámetros a otras funciones que los modificarían y / o llamarían a cualquier función miembro que los modificaría.

Esto se puede usar para hacer suposiciones sobre el estado de los parámetros después de pasar a cualquier función dada, incluso sin ver la definición de esa función.

```
// function_parameter.h

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void const_function_parameter(const ConstMemberFunctions& c);

// We can assume that c is modified and/or c.set_val() is called, and may or may not be passed
//  to any of these functions.  If passed to one_const_one_not, it may be either parameter.
void non_qualified_function_parameter(ConstMemberFunctions& c);

// We can assume that:
  // l is not modified, and l.set_val() won't be called.
  // l may or may not be passed to const_function_parameter().
  // r is modified, and/or r.set_val() may be called.
  // r may or may not be passed to either of the preceding functions.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void bad_parameter(const ConstMemberFunctions& c);
```

Debido a las reglas `const` , estos supuestos serán de hecho aplicados por el compilador.

```
// function_parameter.cpp

// Our assumption was correct.
void const_function_parameter(const ConstMemberFunctions& c) {
    std::cout << "With the current value, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
```

```
              << std::endl;
}

// Our assumption was correct.
void non_qualified_function_parameter(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "For the value 42, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct, in the ugliest possible way.
// Note that const correctness doesn't prevent encapsulation from intentionally being broken,
//  it merely prevents code from having write access when it doesn't need it.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // Let's just punch access modifiers and common sense in the face here.
    struct Machiavelli {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<Machiavelli&>(r).val = l.calc();
    reinterpret_cast<Machiavelli&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers in c.set_val().
void bad_parameter(const ConstMemberFunctions& c) {
    c.set_val(18);
}
```

Si bien *es* posible eludir la corrección `const` y, por extensión, romper estas garantías, el programador debe hacer esto intencionalmente (al igual que romper la encapsulación con `Machiavelli` , arriba), y es probable que cause un comportamiento indefinido.

```
class DealBreaker : public ConstMemberFunctions {
  public:
    DealBreaker(int v = 0);

    // A foreboding name, but it's const...
    void no_guarantees() const;
}

DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}

// Our assumption was incorrect.
// const_cast removes const-ness, making the compiler think we know what we're doing.
void DealBreaker::no_guarantees() const {
    const_cast<DealBreaker*>(this)->set_val(823);
}

// ...

const DealBreaker d(50);
d.no_guarantees(); // Undefined behaviour: d really IS const, it may or may not be modified.
```

Sin embargo, debido a que esto requiera el programador de *decir* muy específicamente al compilador que tienen la intención de ignorar `const` Ness, y ser consistentes entre los compiladores, generalmente es seguro asumir que `const` código correcto se abstendrá de hacerlo a menos que se especifique lo contrario.

Lea Const Correccion en línea: https://riptutorial.com/es/cplusplus/topic/7217/const-correccion

Capítulo 26: constexpr

Introducción

`constexpr` es una palabra clave que se puede usar para marcar el valor de una variable como una expresión constante, una función como potencialmente utilizable en expresiones constantes, o (desde C ++ 17) una declaración if que tiene solo una de sus ramas seleccionadas para compilar.

Observaciones

La palabra clave `constexpr` se agregó en C ++ 11, pero desde hace algunos años desde que se publicó el estándar C ++ 11, no todos los compiladores principales lo admitieron. En el momento en que se publicó el estándar C ++ 11. En el momento de la publicación de C ++ 14, todos los compiladores principales admiten `constexpr` .

Examples

variables constexpr

Una variable declarada `constexpr` es implícitamente `const` y su valor puede usarse como una expresión constante.

Comparación con `#define`

Un `constexpr` es un reemplazo de tipo seguro para expresiones de tiempo de compilación basadas en `#define` . Con `constexpr` la expresión evaluada en tiempo de compilación se reemplaza con el resultado. Por ejemplo:

C ++ 11

```
int main()
{
   constexpr int N = 10 + 2;
   cout << N;
}
```

producirá el siguiente código:

```
cout << 12;
```

Una macro en tiempo de compilación basada en un preprocesador sería diferente. Considerar:

```
#define N 10 + 2

int main()
{
    cout << N;
```

```
}
```

Producirá:

```
cout << 10 + 2;
```

que obviamente se convertirá en `cout << 10 + 2;` . Sin embargo, el compilador tendría que hacer más trabajo. Además, crea un problema si no se utiliza correctamente.

Por ejemplo (con `#define`):

```
cout << N * 2;
```

formas:

```
cout << 10 + 2 * 2; // 14
```

Pero un `constexpr` daría correctamente `24` .

Comparacion con `const`

Una variable `const` es una **variable** que necesita memoria para su almacenamiento. Un `constexpr` no lo hace. Un `constexpr` produce una constante de tiempo de compilación, que no se puede cambiar. Puedes argumentar que la `const` puede también ser cambiada. Pero considere:

```
int main()
{
   const int size1 = 10;
   const int size2 = abs(10);

   int arr_one[size1];
   int arr_two[size2];
}
```

Con la mayoría de los compiladores, la segunda instrucción fallará (puede funcionar con GCC, por ejemplo). El tamaño de cualquier matriz, como usted sabe, tiene que ser una expresión constante (es decir, resultados en valor de tiempo de compilación). A la segunda variable `size2` se le asigna algún valor que se decide en tiempo de ejecución (aunque sabe que es `10` , para el compilador no es tiempo de compilación).

Esto significa que una `const` puede o no ser una verdadera constante de compilación. No puede garantizar ni hacer cumplir que un valor `const` particular es absolutamente tiempo de compilación. Puedes usar `#define` pero tiene sus propios escollos.

Por lo tanto simplemente use:

C ++ 11

```
int main()
{
```

```
    constexpr int size = 10;

    int arr[size];
}
```

Una expresión `constexpr` debe evaluar un valor en tiempo de compilación. Por lo tanto, no puede utilizar:

C ++ 11

```
constexpr int size = abs(10);
```

A menos que la función (`abs`) esté devolviendo un `constexpr` .

Todos los tipos básicos se pueden inicializar con `constexpr` .

C ++ 11

```
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

De manera interesante y conveniente, también puede usar `auto` :

C ++ 11

```
constexpr auto domain = ".COM";  // const char * const domain = ".COM"
constexpr auto PI = 3.14;        // constexpr double
```

funciones constexpr

Una función que se declara `constexpr` está implícitamente en línea y las llamadas a dicha función potencialmente producen expresiones constantes. Por ejemplo, la siguiente función, si se llama con argumentos de expresión constante, también produce una expresión constante:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

Por lo tanto, el resultado de la llamada a la función se puede usar como una matriz enlazada o un argumento de plantilla, o para inicializar una variable `constexpr` :

C ++ 11

```
int main()
{
    constexpr int S = Sum(10,20);

    int Array[S];
```

```
    int Array2[Sum(20,30)]; // 50 array size, compile time
}
```

Tenga en cuenta que si elimina `constexpr` de la especificación de tipo de retorno de la función, la asignación a `s` no funcionará, ya que `s` es una variable `constexpr` , y debe asignarse una constante de tiempo de compilación. De manera similar, el tamaño de la matriz tampoco será una expresión constante, si la función `Sum` no es `constexpr` .

Lo interesante de `constexpr` funciones `constexpr` es que también puede usarlo como funciones comunes:

C ++ 11

```
int a = 20;
auto sum = Sum(a, abs(-20));
```

`Sum` no será una función `constexpr` ahora, se compilará como una función ordinaria, tomando argumentos variables (no constantes) y devolviendo un valor no constante. No necesitas escribir dos funciones.

También significa que si intenta asignar dicha llamada a una variable no constante, no se compilará:

C ++ 11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

La razón es simple: a `constexpr` solo se le debe asignar una constante de tiempo de compilación. Sin embargo, la llamada a la función anterior hace que `Sum` no sea `constexpr` (el valor R es no const, pero el valor L se declara a sí mismo como `constexpr`).

La función `constexpr` también **debe** devolver una constante de tiempo de compilación. Lo siguiente no se compilará:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;     // ERROR
    return a + b;
}
```

Porque `a1` es una *variable que* no es constexpr, y prohíbe que la función sea una verdadera función `constexpr` . `constexpr` y asignarle `a` testamento tampoco funcionará, ya que aún no se conoce el valor de `a` (parámetro entrante):

C ++ 11

```
constexpr int Sum(int a, int b)
```

```
{
   constexpr int a1 = a;     // ERROR
   ..
```

Además, lo siguiente tampoco compilará:

C ++ 11

```
constexpr int Sum(int a, int b)
{
   return abs(a) + b; // or abs(a) + abs(b)
}
```

Dado que `abs(a)` no es una expresión constante (incluso `abs(10)` no funcionará, ya que `abs` no devuelve `constexpr int` !

Que hay de esto

C ++ 11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

Creamos nuestra propia función `Abs` , que es un `constexpr` , y el cuerpo de `Abs` tampoco rompe ninguna regla. Además, en el sitio de llamada (dentro de `Sum`), la expresión se evalúa como `constexpr` . Por lo tanto, la llamada a `Sum(-10, 20)` será una expresión constante en tiempo de compilación que resultará en `30` .

Estática si declaración

C ++ 17

La `if constexpr` se puede usar para compilar condicionalmente el código. La condición debe ser una expresión constante. La rama no seleccionada se *descarta.* No se crea una instancia de una declaración descartada dentro de una plantilla. Por ejemplo:

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
  // ... handle p
  if constexpr (sizeof...(rs) > 0)
    g(rs...);  // never instantiated with an empty argument list
}
```

Además, no es necesario definir las variables y funciones que solo se usan dentro de las

sentencias descartadas, y las sentencias de `return` descartadas no se utilizan para la deducción del tipo de devolución de la función.

`if constexpr` es distinto de `#ifdef` . `#ifdef` compila condicionalmente el código, pero solo en función de las condiciones que se pueden evaluar en el momento del preprocesamiento. Por ejemplo, `#ifdef` no podría usarse para compilar condicionalmente el código dependiendo del valor de un parámetro de plantilla. Por otro lado, `if constexpr` no puede usarse para descartar un código sintácticamente no válido, mientras que `#ifdef` puede.

```
if constexpr(false) {
    foobar;  // error; foobar has not been declared
    std::vector<int> v("hello, world");  // error; no matching constructor
}
```

Lea constexpr en línea: https://riptutorial.com/es/cplusplus/topic/3899/constexpr

Capítulo 27: Construir sistemas

Introducción

C ++, al igual que C, tiene un historial largo y variado con respecto a los flujos de trabajo de compilación y los procesos de construcción. Hoy en día, C ++ tiene varios sistemas de compilación populares que se utilizan para compilar programas, a veces para múltiples plataformas dentro de un sistema de compilación. Aquí, algunos sistemas de construcción serán revisados y analizados.

Observaciones

Actualmente, no existe un sistema de compilación universal o dominante para C ++ que sea popular y multiplataforma. Sin embargo, existen varios sistemas de compilación principales que se adjuntan a las plataformas / proyectos principales, el más notable es GNU Make con el sistema operativo GNU / Linux y NMAKE con el sistema de proyectos Visual C ++ / Visual Studio.

Además, algunos entornos de desarrollo integrados (IDE) también incluyen sistemas de compilación especializados para ser utilizados específicamente con el IDE nativo. Ciertos generadores de sistemas de compilación pueden generar estos formatos de proyecto / sistema de compilación nativos IDE, como CMake para Eclipse y Microsoft Visual Studio 2012.

Examples

Generando entorno de construcción con CMake

CMake genera entornos de compilación para casi cualquier compilador o IDE a partir de una sola definición de proyecto. Los siguientes ejemplos demostrarán cómo agregar un archivo CMake al código C ++ multiplataforma "Hello World" .

Los archivos CMake siempre se denominan "CMakeLists.txt" y ya deberían existir en el directorio raíz de cada proyecto (y posiblemente también en los subdirectorios). Un archivo básico de CMakeLists.txt tiene el siguiente aspecto:

```
cmake_minimum_required(VERSION 2.4)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

Véalo en vivo en Coliru .

Este archivo le dice a CMake el nombre del proyecto, qué versión de archivo debe esperar e instrucciones para generar un ejecutable llamado "HelloWorld" que requiere `main.cpp` .

Genere un entorno de compilación para su compilador / IDE instalado desde la línea de

comandos:

```
> cmake .
```

Construye la aplicación con:

```
> cmake --build .
```

Esto genera el entorno de compilación predeterminado para el sistema, según el sistema operativo y las herramientas instaladas. Mantenga el código fuente limpio de cualquier artefacto de construcción con el uso de compilaciones "fuera de la fuente":

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake también puede abstraer los comandos básicos del shell de la plataforma del ejemplo anterior:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake incluye generadores para una serie de herramientas de construcción e IDE comunes. Para generar makefiles para `nmake` Visual Studio :

```
> cmake -G "NMake Makefiles" ..
> nmake
```

Compilando con GNU make

Introducción

GNU Make (styled `make`) es un programa dedicado a la automatización de ejecutar comandos de shell. GNU Make es un programa específico que pertenece a la familia Make. Sigue siendo popular entre los sistemas operativos similares a Unix y POSIX, incluidos los derivados del kernel de Linux, Mac OS X y BSD.

GNU Make es especialmente notable por estar vinculado al Proyecto GNU, que está vinculado al popular sistema operativo GNU / Linux. GNU Make también tiene versiones compatibles que se ejecutan en varias versiones de Windows y Mac OS X. También es una versión muy estable con un significado histórico que sigue siendo popular. Es por estas razones que GNU Make se enseña a menudo junto con C y C ++.

Reglas básicas

Para compilar con make, cree un Makefile en el directorio de su proyecto. Su Makefile podría ser tan simple como:

Makefile

```
# Set some variables to use in our command
# First, we set the compiler to be g++
CXX=g++

# Then, we say that we want to compile with g++'s recommended warnings and some extra ones.
CXXFLAGS=-Wall -Wextra -pedantic

# This will be the output file
EXE=app

SRCS=main.cpp

# When you call `make` at the command line, this "target" is called.
# The $(EXE) at the right says that the `all` target depends on the `$(EXE)` target.
# $(EXE) expands to be the content of the EXE variable
# Note: Because this is the first target, it becomes the default target if `make` is called
without target
all: $(EXE)

# This is equivalent to saying
# app: $(SRCS)
# $(SRCS) can be separated, which means that this target would depend on each file.
# Note that this target has a "method body": the part indented by a tab (not four spaces).
# When we build this target, make will execute the command, which is:
# g++ -Wall -Wextra -pedantic -o app main.cpp
# I.E. Compile main.cpp with warnings, and output to the file ./app
$(EXE): $(SRCS)
    @$(CXX) $(CXXFLAGS) -o $@ $(SRCS)

# This target should reverse the `all` target. If you call
# make with an argument, like `make clean`, the corresponding target
# gets called.
clean:
    @rm -f $(EXE)
```

> **NOTA: asegúrese de que las sangrías estén con una pestaña, no con cuatro espacios. De lo contrario, obtendrá un error de** `Makefile:10: *** missing separator. Stop.`

Para ejecutar esto desde la línea de comandos, haga lo siguiente:

```
$ cd ~/Path/to/project
$ make
$ ls
app  main.cpp  Makefile

$ ./app
Hello World!
```

```
$ make clean
$ ls
main.cpp  Makefile
```

Construcciones incrementales

Cuando empiezas a tener más archivos, make se vuelve más útil. ¿Qué pasa si editas **a.cpp** pero no **b.cpp** ? Recompilar **b.cpp** tomaría más tiempo.

Con la siguiente estructura de directorios:

```
.
+-- src
|   +-- a.cpp
|   +-- a.hpp
|   +-- b.cpp
|   +-- b.hpp
+-- Makefile
```

Esto sería un buen Makefile:

Makefile

```
CXX=g++
CXXFLAGS=-Wall -Wextra -pedantic
EXE=app

SRCS_GLOB=src/*.cpp
SRCS=$(wildcard $(SRCS_GLOB))
OBJS=$(SRCS:.cpp=.o)

all: $(EXE)

$(EXE): $(OBJS)
    @$(CXX) -o $@ $(OBJS)

depend: .depend

.depend: $(SRCS)
    @-rm -f ./.depend
    @$(CXX) $(CXXFLAGS) -MM $^>>./.depend

clean:
    -rm -f $(EXE)
    -rm $(OBJS)
    -rm *~
    -rm .depend

include .depend
```

De nuevo mira las pestañas. Este nuevo Makefile garantiza que solo recompile archivos modificados, minimizando el tiempo de compilación.

Documentación

Para obtener más información sobre make, consulte la documentación oficial de Free Software Foundation , la documentación de stackoverflow y la elaborada respuesta de dmckee sobre stackoverflow .

Construyendo con scons

Puede crear el código C ++ multiplataforma "Hello World" , utilizando Scons , una herramienta de construcción de software en lenguaje Python .

Primero, cree un archivo llamado `SConstruct` (tenga en cuenta que SCons buscará un archivo con este nombre exacto de forma predeterminada). Por ahora, el archivo debe estar en un directorio a lo largo de su `hello.cpp` . Escribe en el nuevo archivo la línea.

```
Program('hello.cpp')
```

Ahora, desde la terminal, ejecute `scons` . Deberías ver algo como

```
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: done building targets.
```

(aunque los detalles variarán dependiendo de su sistema operativo y compilador instalado).

Las clases `Environment` y `Glob` le ayudarán a configurar aún más qué construir. Por ejemplo, el archivo `SConstruct`

```
env=Environment(CPPPATH='/usr/include/boost/',
    CPPDEFINES=[],
    LIBS=[],
    SCONS_CXX_STANDARD="c++11"
    )

env.Program('hello', Glob('src/*.cpp'))
```

construye el ejecutable `hello` , usando todos los archivos `cpp` en `src` . Su `CPPPATH` es `/usr/include/boost` y especifica el estándar C ++ 11.

Ninja

Introducción

El sistema de construcción Ninja se describe en el sitio web del proyecto como "un sistema de construcción pequeña con un enfoque en la velocidad". Ninja está diseñado para generar sus archivos mediante generadores de archivos de sistema de compilación, y adopta un enfoque de bajo nivel para construir sistemas, en contraste con los administradores de sistemas de compilación de nivel superior como CMake o Meson.

Ninja está escrito principalmente en C ++ y Python, y fue creado como una alternativa al sistema de construcción SCons para el proyecto Chromium.

NMAKE (Utilidad de mantenimiento de programas de Microsoft)

Introducción

NMAKE es una utilidad de línea de comandos desarrollada por Microsoft para ser utilizada principalmente junto con Microsoft Visual Studio y / o las herramientas de línea de comandos de Visual C ++.

NMAKE es un sistema de compilación que forma parte de la familia Make de sistemas de compilación, pero tiene ciertas características distintas que divergen de los programas Make de Unix, como la sintaxis de rutas de archivos específicas de Windows (que a su vez difiere de las rutas de archivos de estilo Unix)

Autotools (GNU)

Introducción

Los Autotools son un grupo de programas que crean un sistema de compilación GNU para un paquete de software dado. Es un conjunto de herramientas que trabajan en conjunto para producir varios recursos de compilación, como un Makefile (para usar con GNU Make). Por lo tanto, Autotools puede considerarse un generador de sistema de construcción de facto.

Algunos programas notables de Autotools incluyen:

- Autoconf
- Automake (no debe confundirse con `make`)

En general, Autotools está destinado a generar el script compatible con Unix y Makefile para permitir que el siguiente comando genere (e instale) la mayoría de los paquetes (en el caso simple):

```
./configure && make && make install
```

Como tal, Autotools también tiene una relación con ciertos administradores de paquetes, especialmente aquellos que están conectados a sistemas operativos que cumplen con los Estándares POSIX.

Lea Construir sistemas en línea: https://riptutorial.com/es/cplusplus/topic/8200/construir-sistemas

Capítulo 28: Contenedores C ++

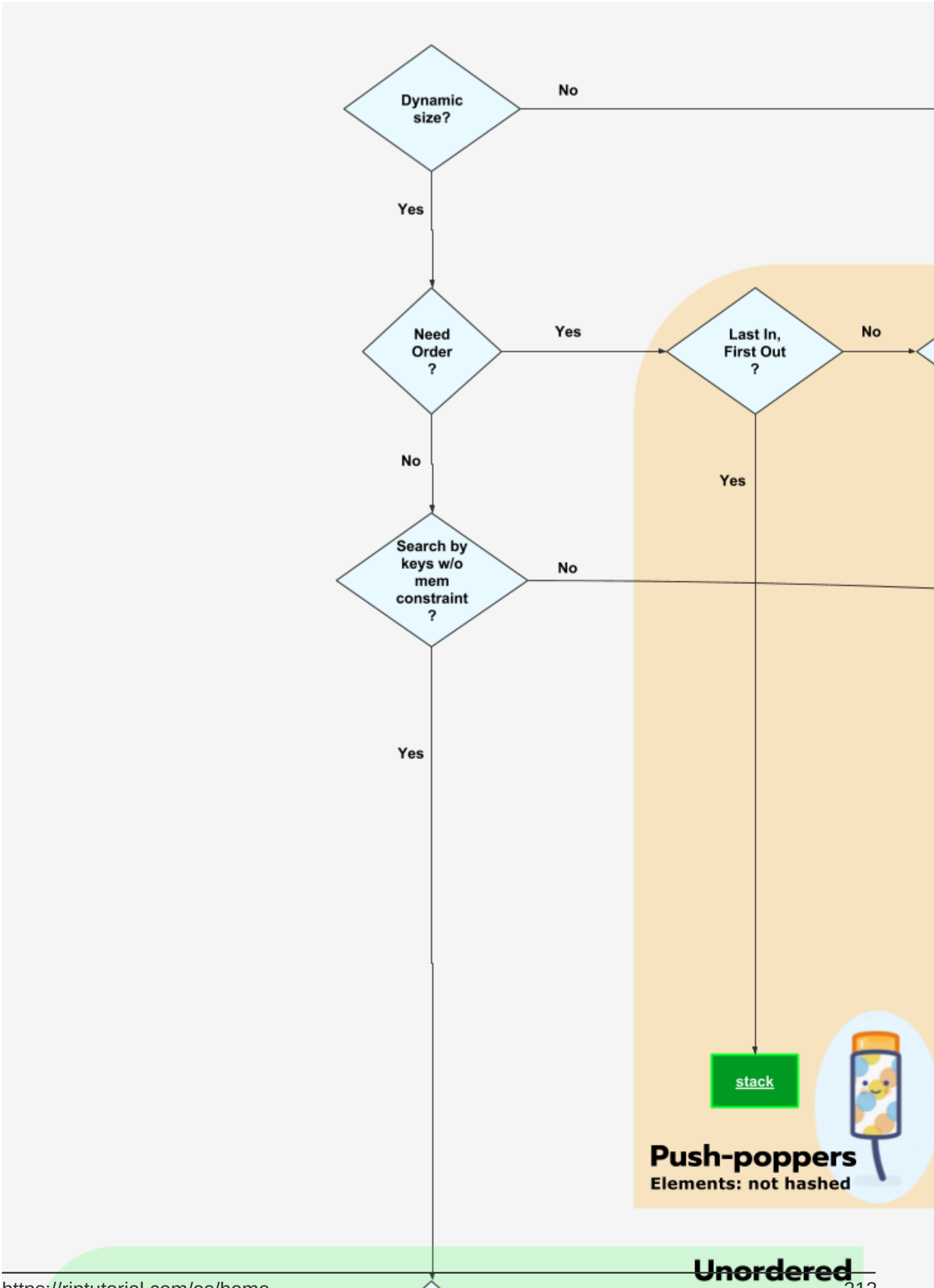
Introducción

Los contenedores C ++ almacenan una colección de elementos. Los contenedores incluyen vectores, listas, mapas, etc. Usando plantillas, los contenedores C ++ contienen colecciones de primitivas (por ejemplo, ints) o clases personalizadas (por ejemplo, MyClass).

Examples

Diagrama de flujo de contenedores C ++

Elegir qué contenedor de C ++ usar puede ser complicado, así que aquí hay un diagrama de flujo simple para ayudarlo a decidir qué contenedor es el adecuado para el trabajo.


Dynamic size?
No
Yes
Need Order ?
Yes
Last In, First Out ?
No
Yes
No
Search by keys w/o mem constraint ?
No
Yes
stack
Push-poppers
Elements: not hashed
Unordered
Elements: hash code

. Este pequeño gráfico en el diagrama de flujo es de Megan Hopkins

Lea Contenedores C ++ en línea: https://riptutorial.com/es/cplusplus/topic/10848/contenedores-c-plusplus

Capítulo 29: Control de flujo

Observaciones

Consulte el tema de los bucles para los diferentes tipos de bucles.

Examples

caso

Introduce una etiqueta de caso de una declaración de cambio. El operando debe ser una expresión constante y coincidir con la condición de cambio en el tipo. Cuando se ejecuta la instrucción de cambio, saltará a la etiqueta del caso con el operando igual a la condición, si la hay.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

cambiar

Según la norma C ++,

> La instrucción de `switch` hace que el control se transfiera a una de varias declaraciones, según el valor de una condición.

El `switch` palabra clave va seguido de una condición entre paréntesis y un bloque, que puede contener etiquetas de `case` y una etiqueta `default` opcional. Cuando se ejecuta la instrucción de cambio, el control se transferirá a una etiqueta de `case` con un valor que coincida con el de la condición, si corresponde, o a la etiqueta `default` , si corresponde.

La condición debe ser una expresión o una declaración, que tenga un tipo de entero o enumeración, o un tipo de clase con una función de conversión a tipo de entero o enumeración.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
```

```
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

captura

La palabra clave `catch` introduce un controlador de excepciones, es decir, un bloque al que se transferirá el control cuando se lance una excepción de tipo compatible. A la palabra clave `catch` le sigue una *declaración de excepción* entre paréntesis, que es similar en forma a una declaración de parámetro de función: el nombre del parámetro se puede omitir, y se permiten los puntos suspensivos `...` que coinciden con cualquier tipo. El controlador de excepciones solo manejará la excepción si su declaración es compatible con el tipo de excepción. Para más detalles, ver excepciones de captura .

```
try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}
```

defecto

En una declaración de cambio, introduce una etiqueta a la que se saltará si el valor de la condición no es igual a ninguno de los valores de las etiquetas de caso.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

C ++ 11

Define un constructor predeterminado, un constructor de copia, un constructor de movimiento, un

destructor, un operador de asignación de copia o un operador de asignación de movimiento para tener su comportamiento predeterminado.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

Si

Introduce una sentencia if. La palabra clave `if` debe ir seguida de una condición entre paréntesis, que puede ser una expresión o una declaración. Si la condición es verdadera, se ejecutará la subestación posterior a la condición.

```
int x;
std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}
```

más

La primera subestación de una sentencia if puede ir seguida de la palabra clave `else` . La subestación después de la palabra clave `else` se ejecutará cuando la condición sea falsey (es decir, cuando no se ejecute la primera subestación).

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

ir

Salta a una declaración etiquetada, que debe estar ubicada en la función actual.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
    result = true;
  end:
```

```
    release_widget(widget);
    return result;
}
```

regreso

Devuelve el control de una función a su interlocutor.

Si el `return` tiene un operando, el operando se convierte al tipo de retorno de la función y el valor convertido se devuelve al llamante.

```
int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3
```

Si `return` no tiene un operando, la función debe tener un tipo de retorno `void` . Como caso especial, una función `void` -returning también puede devolver una expresión si la expresión tiene el tipo `void` .

```
void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}
```

Cuando se devuelve `main` , `std::exit` se llama implícitamente con el valor de retorno, y el valor se devuelve al entorno de ejecución. (Sin embargo, regresar de `main` destruye las variables locales automáticas, mientras que llamar `std::exit` directamente no lo hace).

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

lanzar

1. Cuando se produce el `throw` en una expresión con un operando, su efecto es lanzar una excepción , que es una copia del operando.

   ```
   void print_asterisks(int count) {
       if (count < 0) {
   ```

```
        throw std::invalid_argument("count cannot be negative!");
    }
    while (count--) { putchar('*'); }
}
```

2. Cuando se produce el `throw` en una expresión sin un operando, su efecto es volver a emitir la excepción actual . Si no hay una excepción actual, se llama a `std::terminate` .

```
try {
    // something risky
} catch (const std::bad_alloc&) {
    std::cerr << "out of memory" << std::endl;
} catch (...) {
    std::cerr << "unexpected exception" << std::endl;
    // hope the caller knows how to handle this exception
    throw;
}
```

3. Cuando se produce el `throw` en un declarador de función, introduce una especificación de excepción dinámica, que enumera los tipos de excepciones que la función puede propagar.

```
// this function might propagate a std::runtime_error,
// but not, say, a std::logic_error
void risky() throw(std::runtime_error);
// this function can't propagate any exceptions
void safe() throw();
```

   Las especificaciones de excepciones dinámicas están en desuso a partir de C ++ 11.

Tenga en cuenta que los dos primeros usos de `throw` enumerados anteriormente constituyen expresiones en lugar de declaraciones. (El tipo de una expresión de lanzamiento es `void`). Esto hace posible anidarlos dentro de expresiones, de esta manera:

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

tratar

A la palabra clave `try` le sigue un bloque, o una lista de inicialización de constructor y luego un bloque (ver aquí). Al bloque try le sigue uno o más bloques catch . Si una excepción se propaga fuera del bloque try, cada uno de los bloques catch correspondientes después del bloque try tiene la oportunidad de manejar la excepción, si los tipos coinciden.

```
std::vector<int> v(N);      // if an exception is thrown here,
                            // it will not be caught by the following catch block
try {
    std::vector<int> v(N); // if an exception is thrown here,
                           // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
    // handle bad_alloc exceptions from the try block
```

```
}
```

Estructuras condicionales: if, if..else

si y si no

se usaba para verificar si la expresión dada devuelve verdadero o falso y actúa como tal:

```
if (condition) statement
```

la condición puede ser cualquier expresión válida de C ++ que devuelva algo que se verifique con la verdad / falsedad, por ejemplo:

```
if (true) { /* code here */ }  // evaluate that true is true and execute the code in the
brackets
if (false) { /* code here */ } // always skip the code since false is always false
```

La condición puede ser cualquier cosa, una función, una variable o una comparación, por ejemplo.

```
if(istrue()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evalute the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the experssion (a==b) which will be true if
equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any
non zero value will be true,
```

Si queremos comprobar si hay varias expresiones, podemos hacerlo de dos maneras:

utilizando operadores binarios :

```
if (a && b) { } // will be true only if both a and b are true (binary operators are outside
the scope here
if (a || b ) { } //true if a or b is true
```

usando if / ifelse / else :

para un simple cambio ya sea si o bien

```
if (a== "test") {
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

para opciones múltiples:

```
if (a=='a') {
// if a is a char valued 'a'
} else if (a=='b') {
// if a is a char valued 'b'
```

```
} else if (a=='c') {
// if a is a char valued 'c'
} else {
//if a is none of the above
}
```

sin embargo, debe tenerse en cuenta que debe usar ' **switch** ' en su lugar si su código verifica el valor de la misma variable

Saltar declaraciones: romper, continuar, goto, salir.

La instrucción de descanso:

Usando break podemos dejar un bucle incluso si la condición para su final no se cumple. Puede usarse para terminar un bucle infinito, o para forzarlo a que termine antes de su final natural.

La sintaxis es

```
break;
```

Ejemplo : a menudo utilizamos `break` en los casos de `switch` , es decir, una vez que se cumple un caso, el bloque de código de esa condición se ejecuta.

```
switch(conditon){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

en este caso, si se satisface el caso 1, entonces se ejecuta el bloque 1, lo que realmente queremos es que solo se procese el bloque1, pero una vez que se procesa el bloque1, los bloques restantes, el bloque2, el bloque3 y el bloqueo por defecto también se procesan, aunque solo el caso 1 fue satisfecho Para evitar esto, usamos break al final de cada bloque como:

```
switch(condition){
case 1: block1;
        break;
case 2: block2;
        break;
case 3: block3;
        break;
default: blockdefault;
        break;
}
```

por lo que solo se procesa un bloque y el control sale del bucle de conmutación.

break también se puede usar en otros bucles condicionales y no condicionales como `if` , `while` , `for` etc;

ejemplo:

```
if(condition1){
   ....
   if(condition2){
    .......
    break;
    }
 ...
}
```

La instrucción continua:

La instrucción de continuación hace que el programa omita el resto del bucle en la iteración actual como si se hubiera llegado al final del bloque de instrucciones, lo que provocó que saltara a la siguiente iteración.

La sintaxis es

```
continue;
```

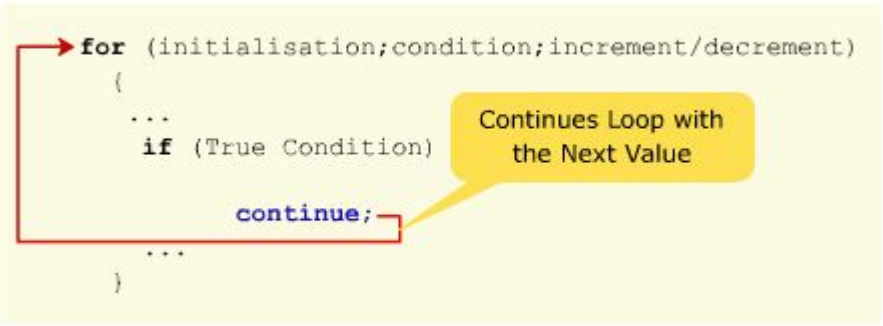
Ejemplo considera lo siguiente:

```
for(int i=0;i<10;i++){
if(i%2==0)
continue;
cout<<"\n @"<<i;
}
```

que produce la salida:

```
 @1
 @3
 @5
 @7
 @9
```

i este código siempre que se cumpla la condición `i%2==0 continue` se procesa, esto hace que el compilador omita todo el código restante (imprimiendo @ ei) y se ejecute la declaración de incremento / decremento del bucle.



La instrucción goto:

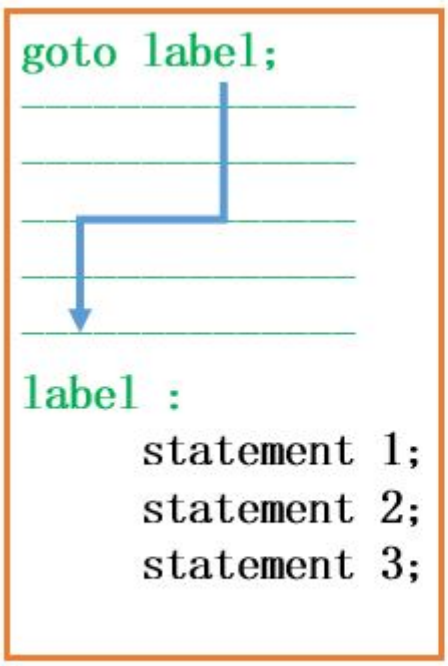
Permite realizar un salto absoluto a otro punto del programa. Debe usar esta función con cuidado,

ya que su ejecución ignora cualquier tipo de limitación de anidamiento. El punto de destino se identifica mediante una etiqueta, que luego se utiliza como un argumento para la instrucción goto. Una etiqueta está hecha de un identificador válido seguido de dos puntos (:)

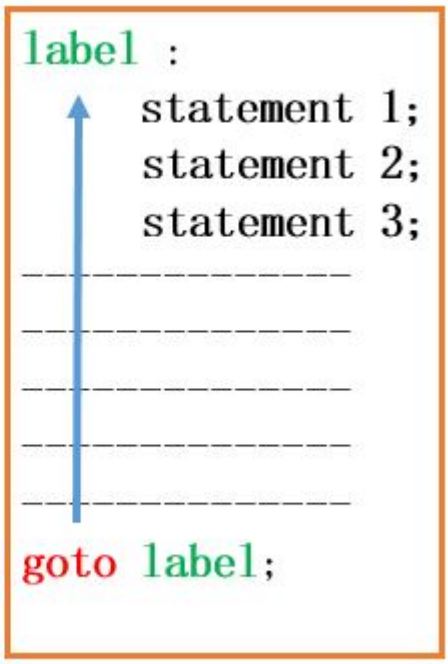
La sintaxis es

```
goto label;
..
.
label: statement;
```

Nota: El *uso de la instrucción goto es altamente desaconsejable porque dificulta el seguimiento del flujo de control de un programa, lo que hace que el programa sea difícil de entender y de modificar.*



Forward Reference



Backward Reference

Ejemplo:

```
int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;
        goto STEP;
     }

   cout << "value of num : " << num << endl;
   num = num + 1;
 }while( num < 10 );
```

salida:

```
value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

siempre que se cumpla la condición `num%2==0` goto envía el control de ejecución al comienzo del bucle `do-while` while.

La función de salida:

`exit` es una función definida en `cstdlib` . El propósito de `exit` es terminar el programa en ejecución con un código de salida específico. Su prototipo es:

```
void exit (int exit code);
```

`cstdlib` define los códigos de salida estándar `EXIT_SUCCESS` y `EXIT_FAILURE` .

Lea Control de flujo en línea: https://riptutorial.com/es/cplusplus/topic/7837/control-de-flujo

Capítulo 30: Conversiones de tipo explícito

Introducción

Una expresión puede ser *convertido* o *fundido* para escribir *explícitamente* `T` usando `dynamic_cast<T>` , `static_cast<T>` , `reinterpret_cast<T>` , o `const_cast<T>` , dependiendo de qué tipo de molde que se pretende.

C ++ también admite la notación de conversión de estilo de función, `T(expr)` y la notación de conversión de estilo de C, `(T)expr` .

Sintaxis

- *especificador de tipo simple* `(` `)`
- *especificador de tipo simple* `(` *expresión-lista* `)`
- *especificador de tipo simple braced-init-list*
- *typename-specifier* `(` `)`
- *typename-specifier* `(` *expresión-lista* `)`
- *nombre de archivo-especificador braced- init-list*
- `dynamic_cast` `<` *type-id* `>` `(` *expresión* `)`
- `static_cast` `<` *type-id* `>` `(` *expresión* `)`
- `reinterpret_cast` `<` *type-id* `>` `(` *expresión* `)`
- `const_cast` `<` *type-id* `>` `(` *expresión* `)`
- `(` *type-id* `)` *expresión-cast*

Observaciones

Las seis notaciones emitidas tienen una cosa en común:

- La `dynamic_cast<Derived&>(base)` a un tipo de referencia `dynamic_cast<Derived&>(base)` , como en `dynamic_cast<Derived&>(base)` , produce un lvalue. Por lo tanto, cuando desea hacer algo con el mismo objeto pero tratarlo como un tipo diferente, se convertiría a un tipo de referencia de valor l.
- La conversión a un tipo de referencia de rvalue, como en `static_cast<string&&>(s)` , produce un rvalue.
- La conversión a un tipo que no es de referencia, como en `(int)x` , produce un prvalue, que puede considerarse como una *copia* del valor que se está emitiendo, pero con un tipo diferente del original.

La palabra clave `reinterpret_cast` es responsable de realizar dos tipos diferentes de conversiones "inseguras":

- Las conversiones de "tipo punning" , que se pueden usar para acceder a la memoria de un tipo como si fuera de un tipo diferente.
- Conversiones entre tipos enteros y tipos de punteros , en cualquier dirección.

La palabra clave `static_cast` puede realizar una variedad de diferentes conversiones:

- Base a conversiones derivadas
- Cualquier conversión que se pueda realizar mediante una inicialización directa, incluidas las conversiones implícitas y las conversiones que llaman a un constructor explícito o una función de conversión. Vea aquí y aquí para más detalles.
- Para `void` , lo que descarta el valor de la expresión.

```
// on some compilers, suppresses warning about x being unused
static_cast<void>(x);
```

- Entre los tipos aritméticos y de enumeración, y entre diferentes tipos de enumeración. Ver las conversiones de enumeración
- Del puntero al miembro de la clase derivada, al puntero del miembro de la clase base. Los tipos apuntados deben coincidir. Ver conversión derivada a base para punteros a miembros
- `void*` a `T*` .

C ++ 11

- Desde un lvalue a un xvalue, como en `std::move` . Ver movimiento semántico .

Examples

Base a conversión derivada

Un puntero a la clase base se puede convertir en un puntero a la clase derivada usando `static_cast` . `static_cast` no realiza ninguna comprobación en tiempo de ejecución y puede provocar un comportamiento indefinido cuando el puntero no apunta al tipo deseado.

```
struct Base {};
struct Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1;                          // error; cast required
Derived* p3 = static_cast<Derived*>(p1); // OK; p2 now points to Derived object
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // undefined behaviour since p4 does not
                                           // point to a Derived object
```

Del mismo modo, una referencia a la clase base se puede convertir en una referencia a la clase derivada usando `static_cast` .

```
struct Base {};
struct Derived : Base {};
Derived d;
Base& r1 = d;
```

```
Derived& r2 = r1;                          // error; cast required
Derived& r3 = static_cast<Derived&>(r1); // OK; r3 now refers to Derived object
```

Si el tipo de fuente es polimórfico, `dynamic_cast` se puede usar para realizar una conversión de base a derivada. Realiza una verificación en tiempo de ejecución y la falla es recuperable en lugar de producir un comportamiento indefinido. En el caso del puntero, se devuelve un puntero nulo en caso de error. En el caso de referencia, se produce una excepción en caso de error de tipo `std::bad_cast` (o una clase derivada de `std::bad_cast`).

```
struct Base { virtual ~Base(); }; // Base is polymorphic
struct Derived : Base {};
Base* b1 = new Derived;
Derived* d1 = dynamic_cast<Derived*>(b1); // OK; d1 points to Derived object
Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 is a null pointer
```

Arrojando constness

Un puntero a un objeto const se puede convertir en un puntero a un objeto no constante utilizando la palabra clave `const_cast` . Aquí usamos `const_cast` para llamar a una función que no es const-correcta. Solo acepta un argumento no const `char*` aunque nunca escribe a través del puntero:

```
void bad_strlen(char*);
const char* s = "hello, world!";
bad_strlen(s);                     // compile error
bad_strlen(const_cast<char*>(s)); // OK, but it's better to make bad_strlen accept const char*
```

`const_cast` puede utilizar `const_cast` al tipo de referencia para convertir un lvalue cualificado por const en un valor no calificado por const.

`const_cast` es peligroso porque hace imposible que el sistema de tipo C ++ impida que intente modificar un objeto const. Si lo hace, se traduce en un comportamiento indefinido.

```
const int x = 123;
int& mutable_x = const_cast<int&>(x);
mutable_x = 456; // may compile, but produces *undefined behavior*
```

Tipo de conversión de punning

Un puntero (referencia de referencia) a un tipo de objeto se puede convertir en un puntero (referencia de referencia) a cualquier otro tipo de objeto utilizando `reinterpret_cast` . Esto no llama a ningún constructor o funciones de conversión.

```
int x = 42;
char* p = static_cast<char*>(&x);      // error: static_cast cannot perform this conversion
char* p = reinterpret_cast<char*>(&x); // OK
*p = 'z';                              // maybe this modifies x (see below)
```

C ++ 11

El resultado de `reinterpret_cast` representa la misma dirección que el operando, siempre que la dirección esté alineada adecuadamente para el tipo de destino. De lo contrario, el resultado no se especifica.

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // should never fire
```

C ++ 11

El resultado de `reinterpret_cast` no está especificado, excepto que un puntero (referencia respectiva) sobrevivirá un viaje de ida y vuelta desde el tipo de origen al tipo de destino y viceversa, siempre que el requisito de alineación del tipo de destino no sea más estricto que el del tipo de origen.

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

En la mayoría de las implementaciones, `reinterpret_cast` no cambia la dirección, pero este requisito no se estandarizó hasta C ++ 11.

`reinterpret_cast` también se puede utilizar para convertir de un tipo de puntero a datos a otro, o un tipo de función de puntero a miembro a otro.

El uso de `reinterpret_cast` se considera peligroso porque la lectura o escritura a través de un puntero o una referencia obtenida mediante `reinterpret_cast` puede desencadenar un comportamiento indefinido cuando los tipos de origen y destino no están relacionados.

Conversión entre puntero y entero

Un puntero de objeto (incluido `void*`) o un puntero de función se puede convertir a un tipo entero usando `reinterpret_cast` . Esto solo se compilará si el tipo de destino es lo suficientemente largo. El resultado está definido por la implementación y generalmente proporciona la dirección numérica del byte en memoria a la que apuntan los punteros.

Por lo general, `long` o `unsigned long` es lo suficientemente largo para contener cualquier valor de puntero, pero esto no está garantizado por la norma.

C ++ 11

Si los tipos `std::intptr_t` y `std::uintptr_t` existen, se garantiza que son lo suficientemente largos para contener un `void*` (y, por lo tanto, cualquier puntero al tipo de objeto). Sin embargo, no se garantiza que sean lo suficientemente largos para contener un puntero de función.

De manera similar, `reinterpret_cast` se puede usar para convertir un tipo entero en un tipo de puntero. Nuevamente, el resultado está definido por la implementación, pero se garantiza que un

valor de puntero no se modificará en un viaje de ida y vuelta a través de un tipo entero. El estándar no garantiza que el valor cero se convierta en un puntero nulo.

```
void register_callback(void (*fp)(void*), void* arg); // probably a C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // will probably compile
}
long x;
std::cin >> x;
register_callback(my_callback,
                  reinterpret_cast<void*>(x)); // hopefully this doesn't lose information...
```

Conversión por constructor explícito o función de conversión explícita

Una conversión que implique llamar a un constructor explícito o una función de conversión no se puede hacer de manera implícita. Podemos solicitar que la conversión se realice explícitamente usando `static_cast` . El significado es el mismo que el de una inicialización directa, excepto que el resultado es temporal.

```
class C {
    std::unique_ptr<int> p;
  public:
    explicit C(int* p) : p(p) {}
};
void f(C c);
void g(int* p) {
    f(p);                  // error: C::C(int*) is explicit
    f(static_cast<C>(p)); // ok
    f(C(p));               // equivalent to previous line
    C c(p); f(c);          // error: C is not copyable
}
```

Conversión implícita

`static_cast` puede realizar cualquier conversión implícita. Este uso de `static_cast` ocasionalmente puede ser útil, como en los siguientes ejemplos:

- Cuando se pasan argumentos a una elipsis, el tipo de argumento "esperado" no se conoce estáticamente, por lo que no se producirá una conversión implícita.

  ```
  const double x = 3.14;
  printf("%d\n", static_cast<int>(x)); // prints 3
  // printf("%d\n", x); // undefined behaviour; printf is expecting an int here
  // alternative:
  // const int y = x; printf("%d\n", y);
  ```

  Sin la conversión explícita de tipos, se pasaría un objeto `double` a los puntos suspensivos y se produciría un comportamiento indefinido.

- Un operador de asignación de clase derivada puede llamar a un operador de asignación de clase base de la siguiente manera:

```
struct Base { /* ... */ };
struct Derived : Base {
    Derived& operator=(const Derived& other) {
        static_cast<Base&>(*this) = other;
        // alternative:
        // Base& this_base_ref = *this; this_base_ref = other;
    }
};
```

Enum las conversiones

`static_cast` puede convertir de un tipo de punto flotante o entero a un tipo de enumeración (ya sea con o sin ámbito), y *viceversa.* También puede convertir entre tipos de enumeración.

- La conversión de un tipo de enumeración sin ámbito a un tipo aritmético es una conversión implícita; es posible, pero no necesario, usar `static_cast` .

C ++ 11

- Cuando un tipo de enumeración de ámbito se convierte en un tipo aritmético:
    - Si el valor de la enumeración se puede representar exactamente en el tipo de destino, el resultado es ese valor.
    - De lo contrario, si el tipo de destino es un tipo entero, el resultado no se especifica.
    - De lo contrario, si el tipo de destino es un tipo de punto flotante, el resultado es el mismo que el de la conversión al tipo subyacente y luego al tipo de punto flotante.

  Ejemplo:

```
enum class Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};
Format f = Format::PDF;
int a = f;                          // error
int b = static_cast<int>(f);        // ok; b is 1000
char c = static_cast<char>(f);      // unspecified, if 1000 doesn't fit into char
double d = static_cast<double>(f); // d is 1000.0... probably
```

- Cuando un entero o tipo de enumeración se convierte en un tipo de enumeración:
    - Si el valor original está dentro del rango de la enumeración de destino, el resultado es ese valor. Tenga en cuenta que este valor puede ser desigual para todos los enumeradores.
    - De lo contrario, el resultado es no especificado (<= C ++ 14) o indefinido (> = C ++ 17).

  Ejemplo:

```
enum Scale {
    SINGLE = 1,
```

```
    DOUBLE = 2,
    QUAD = 4
};
Scale s1 = 1;                       // error
Scale s2 = static_cast<Scale>(2); // s2 is DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 has value 3, and is not equal to any enumerator
Scale s9 = static_cast<Scale>(9); // unspecified value in C++14; UB in C++17
```

C ++ 11

- Cuando un tipo de punto flotante se convierte en un tipo de enumeración, el resultado es el mismo que la conversión al tipo subyacente de la enumeración y luego al tipo de enumeración.

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
Direction d = static_cast<Direction>(3.14); // d is RIGHT
```

Derivado a conversión base para punteros a miembros

Un puntero a miembro de clase derivada se puede convertir en un puntero a miembro de clase base usando `static_cast` . Los tipos apuntados deben coincidir.

Si el operando es un puntero nulo al valor miembro, el resultado también es un puntero nulo al valor miembro.

De lo contrario, la conversión solo es válida si el miembro al que apunta el operando existe realmente en la clase de destino, o si la clase de destino es una clase base o derivada de la clase que contiene el miembro al que apunta el operando. `static_cast` no comprueba la validez. Si la conversión no es válida, el comportamiento no está definido.

```
struct A {};
struct B { int x; };
struct C : A, B { int y; double z; };
int B::*p1 = &B::x;
int C::*p2 = p1;                                // ok; implicit conversion
int B::*p3 = p2;                                // error
int B::*p4 = static_cast<int B::*>(p2);       // ok; p4 is equal to p1
int A::*p5 = static_cast<int A::*>(p2);       // undefined; p2 points to x, which is a member
                                                // of the unrelated class B
double C::*p6 = &C::z;
double A::*p7 = static_cast<double A::*>(p6); // ok, even though A doesn't contain z
int A::*p8 = static_cast<int A::*>(p6);       // error: types don't match
```

nulo * a T *

En C ++, `void*` no se puede convertir implícitamente a `T*` donde `T` es un tipo de objeto. En su lugar, `static_cast` debe usarse para realizar la conversión explícitamente. Si el operando

realmente apunta a un objeto `T` , el resultado apunta a ese objeto. De lo contrario, el resultado no se especifica.

C ++ 11

Incluso si el operando no apunta a un objeto `T` , siempre que el operando apunte a un byte cuya dirección esté correctamente alineada para el tipo `T` , el resultado de la conversión apunta al mismo byte.

```
// allocating an array of 100 ints, the hard way
int* a = malloc(100*sizeof(*a));                    // error; malloc returns void*
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // ok
// int* a = new int[100];                           // no cast needed
// std::vector<int> a(100);                         // better

const char c = '!';
const void* p1 = &c;
const char* p2 = p1;                               // error
const char* p3 = static_cast<const char*>(p1); // ok; p3 points to c
const int* p4 = static_cast<const int*>(p1);   // unspecified in C++03;
                                               // possibly unspecified in C++11 if
                                               // alignof(int) > alignof(char)
char* p5 = static_cast<char*>(p1);             // error: casting away constness
```

Casting estilo c

El casting de estilo C se puede considerar el casting de "Mejor esfuerzo" y se denomina así porque es el único casting que se podría usar en C. La sintaxis de este casting es la `(NewType)variable` .

Cuando se utiliza este lanzamiento, usa uno de los siguientes lanzamientos de c ++ (en orden):

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

La conversión funcional es muy similar, aunque como pocas restricciones como resultado de su sintaxis: `NewType(expression)` . Como resultado, solo se pueden convertir los tipos sin espacios.

Es mejor usar el nuevo c ++ cast, porque es más legible y se puede detectar fácilmente en cualquier lugar dentro de un código fuente de C ++ y los errores se detectarán en tiempo de compilación, en lugar de en tiempo de ejecución.

Como este lanzamiento puede dar lugar a `reinterpret_cast` no deseado, a menudo se considera peligroso.

Lea Conversiones de tipo explícito en línea:
https://riptutorial.com/es/cplusplus/topic/3090/conversiones-de-tipo-explicito

Capítulo 31: Copia elision

Examples

Propósito de la copia elision

Hay lugares en el estándar donde un objeto se copia o mueve para inicializar un objeto. Elección de copia (a veces llamada optimización de valor de retorno) es una optimización por la cual, bajo ciertas circunstancias específicas, se permite que un compilador evite la copia o el movimiento aunque la norma diga que debe suceder.

Considera la siguiente función:

```
std::string get_string()
{
  return std::string("I am a string.");
}
```

De acuerdo con la estricta redacción de la norma, esta función inicializará una `std::string` temporal, luego la copiará / moverá al objeto de valor de retorno y luego destruirá la temporal. El estándar es muy claro que así se interpreta el código.

Copy elision es una regla que permite a un compilador de C ++ *ignorar* la creación del temporal y su posterior copia / destrucción. Es decir, el compilador puede tomar la expresión de inicialización para el temporal e inicializar directamente el valor de retorno de la función. Esto obviamente guarda el rendimiento.

Sin embargo, tiene dos efectos visibles en el usuario:

1. El tipo debe tener el constructor de copiar / mover que se habría llamado. Incluso si el compilador evita la copia / movimiento, el tipo aún debe poder copiarse / moverse.

2. Los efectos secundarios de los constructores de copia / movimiento no están garantizados en circunstancias en las que puede producirse la elección. Considera lo siguiente:

C ++ 11

```
struct my_type
{
  my_type() = default;
  my_type(const my_type &) {std::cout <<"Copying\n";}
  my_type(my_type &&) {std::cout <<"Moving\n";}
};

my_type func()
{
  return my_type();
}
```

¿Qué hará la `func` llamada? Bueno, nunca se imprimirá "Copiando", ya que el temporal es un rvalue y `my_type` es un tipo movible. Entonces, ¿se imprimirá "Moving"?

Sin la regla de elision de copia, esto se requeriría para imprimir siempre "Mover". Pero como la regla de elision de copia existe, el constructor de movimientos puede o no ser llamado; es dependiente de la implementación

Y, por lo tanto, no puede depender de la llamada de los constructores de copiar / mover en contextos donde es posible la elección.

Debido a que elision es una optimización, su compilador puede no ser compatible con elision en todos los casos. Y sin importar si el compilador elude un caso particular o no, el tipo todavía debe soportar la operación que se está elidiando. Por lo tanto, si se elimina una construcción de copia, el tipo aún debe tener un constructor de copia, aunque no se llamará.

Copia garantizada elision

C ++ 17

Normalmente, elision es una optimización. Si bien prácticamente todos los compiladores son compatibles con la copia de la elision en los casos más simples, tenerla aún representa una carga particular para los usuarios. A saber, el tipo de quién se está eliminando la copia / movimiento *debe* seguir teniendo la operación de copia / movimiento que fue eliminada.

Por ejemplo:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  return std::lock_guard<std::mutex>(a_mutex);
}
```

Esto puede ser útil en los casos en que `a_mutex` es un mutex que es privado en algún sistema, pero un usuario externo puede querer tener un bloqueo de ámbito en él.

Esto tampoco es legal, porque `std::lock_guard` no se puede copiar o mover. A pesar de que virtualmente cada compilador de C ++ ocultará la copia / movimiento, el estándar aún *requiere que* el tipo tenga esa operación disponible.

Hasta C ++ 17.

C ++ 17 ordena la elisión al redefinir efectivamente el significado mismo de ciertas expresiones para que no se realicen copias / movimientos. Considere el código anterior.

Bajo la redacción anterior a C ++ 17, ese código dice crear un temporal y luego usar el temporal para copiar / mover al valor de retorno, pero la copia temporal puede eliminarse. Bajo la redacción de C ++ 17, eso no crea un temporal en absoluto.

En C ++ 17, cualquier expresión prvalue , cuando se usa para inicializar un objeto del mismo tipo que la expresión, no genera un temporal. La expresión inicializa directamente ese objeto. Si

devuelve un prvalue del mismo tipo que el valor de retorno, entonces el tipo no necesita tener un constructor de copiar / mover. Y por lo tanto, bajo las reglas de C ++ 17, el código anterior puede funcionar.

La redacción de C ++ 17 funciona en los casos en que el tipo del prvalue coincide con el tipo que se está inicializando. Por lo tanto, dado `get_lock` arriba, esto tampoco requerirá una copia / movimiento:

```
std::lock_guard the_lock = get_lock();
```

Dado que el resultado de `get_lock` es una expresión prvalue que se utiliza para inicializar un objeto del mismo tipo, no se realizará ninguna copia o movimiento. Esa expresión nunca crea un temporal; se utiliza para inicializar directamente `the_lock` . No hay elision porque no hay copia / movimiento para ser elide elide.

El término "elision de copia garantizada" es, por lo tanto, un nombre poco apropiado, pero ese es el nombre de la función tal como se propone para la estandarización de C ++ 17 . No garantiza en absoluto la elisión; *elimina* la copia / movimiento por completo, redefiniendo C ++ para que nunca haya una copia / movimiento que deba ser borrado.

Esta característica solo funciona en casos que involucran una expresión prvalue. Como tal, esto usa las reglas habituales de elision:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  std::lock_guard<std::mutex> my_lock(a_mutex);
  //Do stuff
  return my_lock;
}
```

Si bien este es un caso válido para la elección de copia, las reglas de C ++ 17 no *eliminan* la copia / movimiento en este caso. Como tal, el tipo aún debe tener un constructor de copiar / mover para usar para inicializar el valor de retorno. Y como `lock_guard` no lo hace, esto sigue siendo un error de compilación. Se permite que las implementaciones se nieguen a rechazar copias al pasar o devolver un objeto de tipo de copia trivial. Esto es para permitir el desplazamiento de tales objetos en registros, que algunas ABI podrían imponer en sus convenciones de llamada.

```
struct trivially_copyable {
    int a;
};

void foo (trivially_copyable a) {}

foo(trivially_copyable{}); //copy elision not mandated
```

Valor de retorno elision

Si devuelve una expresión prvalue de una función, y la expresión prvalue tiene el mismo tipo que

el tipo de retorno de la función, entonces se puede eliminar la copia del prvalue temporal:

```
std::string func()
{
  return std::string("foo");
}
```

Casi todos los compiladores eludirán la construcción temporal en este caso.

Parámetro elision

Cuando pasa un argumento a una función, y el argumento es una expresión prvalue del tipo de parámetro de la función, y este tipo no es una referencia, entonces la construcción del prvalue se puede eliminar.

```
void func(std::string str) { ... }

func(std::string("foo"));
```

Esto dice que para crear una `string` temporal, luego muévala en el parámetro de función `str` . Copy elision permite que esta expresión cree directamente el objeto en `str` , en lugar de usar un movimiento + temporal.

Esta es una optimización útil para los casos en que un constructor se declara `explicit` . Por ejemplo, podríamos haber escrito lo anterior como `func("foo")` , pero solo porque la `string` tiene un constructor implícito que convierte de `const char*` a una `string` . Si ese constructor fuera `explicit` , nos veríamos obligados a usar un temporal para llamar al constructor `explicit` . Copiar elision nos evita tener que hacer una copia / movimiento innecesario.

Valor de retorno con nombre elision

Si devuelve una expresión lvalue desde una función, y este valor lvalue:

- representa una variable automática local para esa función, que se destruirá después de la `return`
- La variable automática no es un parámetro de función.
- y el tipo de la variable es el mismo tipo que el tipo de retorno de la función

Si todos estos son el caso, entonces la copia / movimiento desde el lvalue se puede eliminar:

```
std::string func()
{
  std::string str("foo");
  //Do stuff
  return str;
}
```

Los casos más complejos son elegibles para la elección, pero cuanto más complejo sea el caso, menos probable será que el compilador realmente lo evite:

```
std::string func()
{
  std::string ret("foo");
  if(some_condition)
  {
    return "bar";
  }
  return ret;
}
```

El compilador todavía podría elidir a `ret` , pero las posibilidades de que lo hagan disminuyen.

Como se señaló anteriormente, elision no está permitido para los *parámetros de* valor.

```
std::string func(std::string str)
{
  str.assign("foo");
  //Do stuff
  return str; //No elision possible
}
```

Copia inicializacion elision

Si utiliza una expresión prvalue para copiar, inicialice una variable, y esa variable tiene el mismo tipo que la expresión prvalue, entonces la copia puede ser eliminada.

```
std::string str = std::string("foo");
```

La inicialización de la copia efectivamente transforma esto en `std::string str("foo");` (hay diferencias menores).

Esto también funciona con valores de retorno:

```
std::string func()
{
  return std::string("foo");
}

std::string str = func();
```

Sin la elección de copia, esto provocaría 2 llamadas al constructor de movimientos de `std::string` . Copy elision permite que esto llame al constructor de movimientos 1 o cero veces, y la mayoría de los compiladores optarán por este último.

Lea Copia elision en línea: https://riptutorial.com/es/cplusplus/topic/2489/copia-elision

Capítulo 32: Copiando vs Asignación

Sintaxis

- **Copia Constructor**
- MyClass (const MyClass y otros);
- MyClass (MyClass y otros);
- MyClass (const. Volátil MyClass y otros);
- MyClass (MyClass volátil y otros);
- **Constructor de Asignaciones**
- MyClass & operator = (const MyClass & rhs);
- MyClass & operator = (MyClass & rhs);
- MyClass & operator = (MyClass rhs);
- const MyClass & operator = (const MyClass & rhs);
- const MyClass & operator = (MyClass & rhs);
- const MyClass & operator = (MyClass rhs);
- Operador MyClass = (const MyClass & rhs);
- Operador MyClass = (MyClass & rhs);
- Operador MyClass = (myClass rhs);

Parámetros

rs	Lado derecho de la igualdad para los constructores de copia y asignación. Por ejemplo, el constructor de asignación: MyClass operator = (MyClass & rhs);
Marcador de posición	Marcador de posición

Observaciones

Otros buenos recursos para futuras investigaciones:

¿Cuál es la diferencia entre el operador de asignación y el constructor de copia?

operador de asignación vs constructor de copia C ++

GeeksForGeeks

Artículos de C ++

Examples

Operador de Asignación

El Operador de asignación es cuando reemplaza los datos con un objeto ya existente (previamente inicializado) con los datos de algún otro objeto. Tomemos esto como un ejemplo:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo& operator=(const Foo& rhs)
    {
            data = rhs.data;
            return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2(42);
   foo = foo2; // Assignment Operator Called
   cout << foo.data << endl; //Prints 42
}
```

Puede ver que aquí llamo al operador de asignación cuando ya inicialicé el objeto `foo` . Luego más tarde le asigno `foo2` a `foo` . Todos los cambios que aparecen cuando llama a ese operador de signo igual se define en su función `operator=` . Puede ver un resultado ejecutable aquí: http://cpp.sh/3qtbm

Copia Constructor

Copia del constructor, por otro lado, es el opuesto completo del Constructor de asignación. Esta vez, se utiliza para inicializar un objeto ya inexistente (o no inicializado previamente). Esto significa que copia todos los datos del objeto al que se lo está asignando, sin inicializar realmente el objeto en el que se está copiando. Ahora echemos un vistazo al mismo código que antes, pero modifiquemos el constructor de asignaciones para que sea un constructor de copia:

```
// Copy Constructor
#include <iostream>
#include <string>

using std::cout;
```

```
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2 = foo; // Copy Constructor called
   cout << foo2.data << endl;
}
```

Puedes ver aquí `Foo foo2 = foo;` en la función principal, asigno inmediatamente el objeto antes de inicializarlo, lo cual, como se dijo antes, significa que es un constructor de copia. Y note que no necesito pasar el parámetro int para el objeto `foo2` ya que automáticamente extraje los datos anteriores del objeto foo. Aquí hay un ejemplo de salida: http://cpp.sh/5iu7

Copiar constructor vs Asignación de constructor

Ok, hemos revisado brevemente lo que el constructor de copia y el de asignación están arriba y damos ejemplos de cada uno ahora veamos a ambos en el mismo código. Este código será similar al anterior. Tomemos esto:

```
// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
```

```
    {
        data = rhs.data;
        return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) / Normal Constructor called
   Foo foo2 = foo; //Copy Constructor Called
   cout << foo2.data << endl;

   Foo foo3(42);
   foo3=foo; //Assignment Constructor Called
   cout << foo3.data << endl;
}
```

Salida:

```
2
2
```

Aquí puede ver que primero llamamos al constructor de copia ejecutando la línea `Foo foo2 = foo;` . Ya que no lo inicializamos previamente. Y luego, a continuación, llamamos al operador de asignación en foo3 ya que ya estaba inicializado `foo3=foo` ;

Lea Copiando vs Asignación en línea: https://riptutorial.com/es/cplusplus/topic/7158/copiando-vs-asignacion

Capítulo 33: decltype

Introducción

La palabra clave `decltype` se puede usar para obtener el tipo de una variable, función o expresión.

Examples

Ejemplo básico

Este ejemplo solo ilustra cómo se puede utilizar esta palabra clave.

```
int a = 10;

// Assume that type of variable 'a' is not known here, or it may
// be changed by programmer (from int to long long, for example).
// Hence we declare another variable, 'b' of the same type using
// decltype keyword.
decltype(a) b; // 'decltype(a)' evaluates to 'int'
```

Si, por ejemplo, alguien cambia, escriba 'a' a:

```
float a=99.0f;
```

Entonces el tipo de variable `b` ahora se convierte automáticamente en `float` .

Otro ejemplo

Digamos que tenemos vector:

```
std::vector<int> intVector;
```

Y queremos declarar un iterador para este vector. Una idea obvia es usar `auto` . Sin embargo, puede ser necesario simplemente declarar una variable de iterador (y no asignarla a nada). Haríamos:

```
vector<int>::iterator iter;
```

Sin embargo, con `decltype` se vuelve fácil y menos propenso a errores (si cambia el tipo de `intVector`).

```
decltype(intVector)::iterator iter;
```

Alternativamente:

```
decltype(intVector.begin()) iter;
```

En el segundo ejemplo, el tipo de retorno de `begin` se usa para determinar el tipo real, que es `vector<int>::iterator` .

Si necesitamos un const_iterator, solo necesitamos usar `cbegin` :

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

Lea decltype en línea: https://riptutorial.com/es/cplusplus/topic/9930/decltype

Capítulo 34: deducción de tipo

Observaciones

En noviembre de 2014, el Comité de Normalización de C ++ adoptó la propuesta N3922, que elimina la regla de deducción de tipo especial para los inicializadores automáticos y con refuerzo mediante la sintaxis de inicialización directa. Esto no es parte del estándar C ++ pero ha sido implementado por algunos compiladores.

Examples

Deducción de parámetros de plantilla para constructores.

Antes de C ++ 17, la deducción de la plantilla no puede deducir el tipo de clase para usted en un constructor. Debe especificarse explícitamente. A veces, sin embargo, estos tipos pueden ser muy engorrosos o (en el caso de las lambdas) imposibles de nombrar, por lo que tenemos una proliferación de fábricas de tipos (como `make_pair()` , `make_tuple()` , `back_inserter()` , etc.).

C ++ 17

Esto ya no es necesario:

```
std::pair p(2, 4.5);     // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vi1.begin(), 3,
    std::back_insert_iterator(vi2)); // constructs a back_insert_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

Se considera que los constructores deducen los parámetros de la plantilla de clase, pero en algunos casos esto no es suficiente y podemos proporcionar guías de deducción explícitas:

```
template <class Iter>
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>

int array[] = {1, 2, 3};
std::vector v(std::begin(array), std::end(array)); // deduces std::vector<int>
```

Tipo de plantilla Deducción

Sintaxis genérica de plantillas

```
template<typename T>
void f(ParamType param);

f(expr);
```

Caso 1: `ParamType` es una referencia o un puntero, pero no una referencia universal o directa. En

este caso, la deducción de tipos funciona de esta manera. El compilador ignora la parte de referencia si existe en `expr` . El compilador entonces Patronistas los partidos `expr` tipo 's contra `ParamType` a determing `T` .

```
template<typename T>
void f(T& param);      //param is a reference

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a reference to x as a const int

f(x);                  // T is int, param's type is int&
f(cx);                 // T is const int, param's type is const int&
f(rx);                 // T is const int, param's type is const int&
```

Caso 2: `ParamType` es una referencia universal o una referencia directa. En este caso, la deducción de tipo es la misma que en el caso 1 si `expr` es un valor nominal. Si `expr` es un valor límico, tanto `T` como `ParamType` se deducen como referencias de valor l.

```
template<typename T>
void f(T&& param);     // param is a universal reference

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a reference to x as a const int

f(x);                  // x is lvalue, so T is int&, param's type is also int&
f(cx);                 // cx is lvalue, so T is const int&, param's type is also const int&
f(rx);                 // rx is lvalue, so T is const int&, param's type is also const int&
f(27);                 // 27 is rvalue, so T is int, param's type is therefore int&&
```

Caso 3: `ParamType` es un puntero ni una referencia. Si `expr` es una referencia, la parte de referencia se ignora. Si `expr` es const, se ignora también. Si es volátil, eso también se ignora al deducir el tipo de T.

```
template<typename T>
void f(T param);       // param is now passed by value

int x = 27;            // x is an int
const int cx = x;      // cx is a const int
const int& rx = x;     // rx is a reference to x as a const int

f(x);                  // T's and param's types are both int
f(cx);                 // T's and param's types are again both int
f(rx);                 // T's and param's types are still both int
```

Deducción de tipo automático

C ++ 11

La deducción de tipo usando la palabra clave `auto` funciona casi igual que la Deducción de tipo de plantilla. Abajo hay algunos ejemplos:

```
auto x = 27;           // (x is neither a pointer nor a reference), x's type is int
const auto cx = x;     // (cx is neither a pointer nor a reference), cs's type is const int
const auto& rx = x;    // (rx is a non-universal reference), rx's type is a reference to a
const int

auto&& uref1 = x;      // x is int and lvalue, so uref1's type is int&
auto&& uref2 = cx;     // cx is const int and lvalue, so uref2's type is const int &
auto&& uref3 = 27;     // 27 is an int and rvalue, so uref3's type is int&&
```

Las diferencias se detallan a continuación:

```
auto x1 = 27;          // type is int, value is 27
auto x2(27);           // type is int, value is 27
auto x3 = { 27 };      // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 };         // type is std::initializer_list<int>, value is { 27 }
                       // in some compilers type may be deduced as an int with a
                       // value of 27. See remarks for more information.
auto x5 = { 1, 2.0 }   // error! can't deduce T for std::initializer_list<t>
```

Como puede ver si usa inicializadores con refuerzos, se fuerza automáticamente a crear una variable de tipo `std::initializer_list<T>` . Si no puede deducir el valor de `T` , el código es rechazado.

Cuando se utiliza `auto` como el tipo de retorno de una función, especifica que la función tiene un tipo de retorno final .

```
auto f() -> int {
    return 42;
}
```

C ++ 14

C ++ 14 permite, además de los usos de auto permitido en C ++ 11, lo siguiente:

1. Cuando se usa como el tipo de retorno de una función sin un tipo de retorno final, especifica que el tipo de retorno de la función debe deducirse de las declaraciones de retorno en el cuerpo de la función, si corresponde.

   ```
   // f returns int:
   auto f() { return 42; }
   // g returns void:
   auto g() { std::cout << "hello, world!\n"; }
   ```

2. Cuando se usa en el tipo de parámetro de un lambda, define el lambda como un lambda genérico .

   ```
   auto triple = [](auto x) { return 3*x; };
   const auto x = triple(42); // x is a const int with value 126
   ```

La forma especial `decltype(auto)` deduce un tipo utilizando las reglas de deducción de `decltype` de `decltype` lugar de las de `auto` .

```
int* p = new int(42);
auto x = *p;            // x has type int
decltype(auto) y = *p; // y is a reference to *p
```

En C ++ 03 y anteriores, la palabra clave `auto` tenía un significado completamente diferente como un especificador de clase de almacenamiento que se heredó de C.

Lea deducción de tipo en línea: https://riptutorial.com/es/cplusplus/topic/7863/deduccion-de-tipo

Capítulo 35: Devolviendo varios valores de una función

Introducción

Hay muchas situaciones en las que es útil devolver varios valores de una función: por ejemplo, si desea ingresar un artículo y devolver el precio y el número en stock, esta funcionalidad podría ser útil. Hay muchas formas de hacer esto en C ++, y la mayoría involucra a la STL. Sin embargo, si desea evitar el STL por alguna razón, todavía hay varias formas de hacer esto, incluyendo `structs/classes` y `arrays` .

Examples

Uso de parámetros de salida

Los parámetros se pueden usar para devolver uno o más valores; Esos parámetros deben ser punteros o referencias no `const` .

Referencias:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
    e = a * b;
    f = a / b;
}
```

Punteros:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {
    *c = a + b;
    *d = a - b;
    *e = a * b;
    *f = a / b;
}
```

Algunas bibliotecas o marcos de trabajo usan un `#define` 'vacío' para `#define` que los parámetros sean parámetros de salida en la firma de la función. Esto no tiene un impacto funcional y se compilará, pero hace que la firma de la función sea un poco más clara;

```
#define OUT

void calculate(int a, int b, OUT int& c) {
    c = a + b;
}
```

Usando std :: tuple

C ++ 11

El tipo `std::tuple` puede agrupar cualquier número de valores, posiblemente incluyendo valores de diferentes tipos, en un solo objeto de retorno:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
   return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

En C ++ 17, se puede usar una lista de inicializadores de arriostramiento:

C ++ 17

```
std::tuple<int, int, int, int> foo(int a, int b)    {
    return {a + b, a - b, a * b, a / b};
}
```

La recuperación de valores de la `tuple` devuelta puede ser complicada, y requiere el uso de la función `std::get` template:

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

Si los tipos se pueden declarar antes de que se devuelva la función, entonces se puede emplear `std::tie` para descomprimir una `tuple` en variables existentes:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

Si uno de los valores devueltos no es necesario, se puede usar `std::ignore` :

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

C ++ 17

Los enlaces estructurados se pueden utilizar para evitar `std::tie` :

```
auto [add, sub, mul, div] = foo(5,12);
```

Si desea devolver una tupla de referencias de lvalor en lugar de una tupla de valores, use `std::tie` en lugar de `std::make_tuple` .

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
  if (b<a)
    return std::tie(b,a);
```

```
  else
    return std::tie(a,b);
}
```

lo que permite

```
void increase_least(int& a, int& b) {
  std::get<0>(minmax(a,b))++;
}
```

En algunos casos raros `std::forward_as_tuple` lugar de `std::tie` ; tenga cuidado si lo hace, ya que los temporales pueden no durar lo suficiente como para ser consumidos.

Usando std :: array

C ++ 11

El contenedor `std::array` puede agrupar un número fijo de valores de retorno. Este número debe conocerse en tiempo de compilación y todos los valores de retorno deben ser del mismo tipo:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

Esto reemplaza los arrays de estilo c de la `int bar[4]` form `int bar[4]` . La ventaja es que ahora se pueden usar varias funciones std de `c++` . También proporciona funciones útiles como miembros `at` que es una función de acceso miembro de seguro con la comprobación de encuadernado, y `size` que le permite devolver el tamaño de la matriz sin cálculo.

Usando std :: pair

La estructura struct template `std::pair` puede agrupar *exactamente* dos valores de retorno, de cualquiera de los dos tipos:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

Con C ++ 11 o posterior, se puede usar una lista de inicializadores en lugar de `std::make_pair` :

C ++ 11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

Los valores individuales del `std::pair` devuelto se pueden recuperar utilizando los objetos miembros `first` y `second` del par:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Salida:

10

Usando struct

Se puede usar una `struct` para agrupar múltiples valores de retorno:

C ++ 11

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}

auto calc = foo(5, 12);
```

C ++ 11

En lugar de la asignación a campos individuales, se puede usar un constructor para simplificar la construcción de valores devueltos:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
    foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}
};

foo_return_type foo(int a, int b) {
     return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

Los resultados individuales devueltos por la función `foo()` se pueden recuperar accediendo a las variables miembro de la `struct calc` :

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';
```

Salida:

17 -7 60 0

Nota: cuando se utiliza una `struct` , los valores devueltos se agrupan en un solo objeto y se puede acceder a ellos utilizando nombres significativos. Esto también ayuda a reducir el número de variables extrañas creadas en el alcance de los valores devueltos.

C ++ 17

Para desempaquetar una `struct` devuelta desde una función, se pueden usar enlaces estructurados . Esto coloca los parámetros de salida en una posición uniforme con los parámetros de entrada:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';
```

La salida de este código es idéntica a la anterior. La `struct` todavía se utiliza para devolver los valores de la función. Esto le permite hacer frente a los campos de forma individual.

Encuadernaciones Estructuradas

C ++ 17

C ++ 17 introduce enlaces estructurados, lo que hace que sea aún más fácil tratar con múltiples tipos de devolución, ya que no es necesario confiar en `std::tie()` ni realizar ningún desempaquetado manual de tuplas:

```
std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}
```

Los enlaces estructurados se pueden usar de forma predeterminada con `std::pair` , `std::tuple` y cualquier tipo cuyos miembros de datos no estáticos sean todos miembros públicos directos o miembros de una clase base no ambigua:

```
struct A { int x; };
struct B : A { int y; };
B foo();

// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
auto& x = result.x;
```

```
auto& y = result.y;
```

Si hace que su tipo sea "similar a una tupla", también funcionará automáticamente con su tipo. Un tuple-like es un tipo con `tuple_size` , `tuple_element` y `get tuple_element` :

```
namespace my_ns {
    struct my_type {
        int x;
        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}
```

ahora esto funciona:

```
my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}
```

Usando un consumidor de objetos de función

Podemos proporcionar un consumidor al que se llamará con los múltiples valores relevantes:

C ++ 11

```
template <class F>
void foo(int a, int b, F consumer) {
    consumer(a + b, a - b, a * b, a / b);
}

// use is simple... ignoring some results is possible as well
foo(5, 12, [](int sum, int , int , int ){
    std::cout << "sum is " << sum << '\n';
});
```

Esto se conoce como "estilo de paso de continuación" .

Puede adaptar una función devolviendo una tupla a una función de estilo de paso de continuación a través de:

C ++ 17

```
template<class Tuple>
struct continuation {
  Tuple t;
  template<class F>
  decltype(auto) operator->*(F&& f)&&{
    return std::apply( std::forward<F>(f), std::move(t) );
  }
};
std::tuple<int,int,int,int> foo(int a, int b);

continuation(foo(5,12))->*[](int sum, auto&&...) {
  std::cout << "sum is " << sum << '\n';
};
```

con versiones más complejas que se pueden escribir en C ++ 14 o C ++ 11.

Usando std :: vector

Un `std::vector` puede ser útil para devolver un número dinámico de variables del mismo tipo. El siguiente ejemplo usa `int` como tipo de datos, pero un `std::vector` puede contener cualquier tipo que se pueda copiar de forma trivial:

```
#include <vector>
#include <iostream>

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
    std::vector<int> v = fillVectorFrom(1, 10);
```

```
    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Usando el iterador de salida

Se pueden devolver varios valores del mismo tipo pasando un iterador de salida a la función. Esto es particularmente común para funciones genéricas (como los algoritmos de la biblioteca estándar).

Ejemplo:

```
template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
}
```

Ejemplo de uso:

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits now contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Lea Devolviendo varios valores de una función en línea:
https://riptutorial.com/es/cplusplus/topic/487/devolviendo-varios-valores-de-una-funcion

Capítulo 36: Diseño de tipos de objetos

Observaciones

Ver también Tamaño de tipos integrales .

Examples

Tipos de clase

Por "clase", nos referimos a un tipo que se definió usando la palabra clave `class` o `struct` (pero no `enum struct` o `enum class`).

- Incluso una clase vacía aún ocupa al menos un byte de almacenamiento; Por lo tanto, consistirá puramente de relleno. Esto garantiza que si `p` apunta a un objeto de una clase vacía, `p + 1` es una dirección distinta y apunta a un objeto distinto. Sin embargo, es posible que una clase vacía tenga un tamaño de 0 cuando se utiliza como clase base. Ver la optimización de la base vacía .

```
class Empty_1 {};                               // sizeof(Empty_1)       == 1
class Empty_2 {};                               // sizeof(Empty_2)       == 1
class Derived : Empty_1 {};                     // sizeof(Derived)       == 1
class DoubleDerived : Empty_1, Empty_2 {};      // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; };                    // sizeof(Holder)        == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder)  == 2
class DerivedHolder : Empty_1 { Empty_1 e; };   // sizeof(DerivedHolder) == 2
```

- La representación de objeto de un tipo de clase contiene las representaciones de objeto de la clase base y los tipos de miembros no estáticos. Por lo tanto, por ejemplo, en la siguiente clase:

```
struct S {
    int x;
    char* y;
};
```

  hay una secuencia consecutiva de bytes de `sizeof(int)` dentro de un objeto `S` , llamado *subobjeto,* que contiene el valor de `x` , y otro subobjeto con bytes de `sizeof(char*)` que contiene el valor de `y` . Los dos no pueden ser intercalados.

- Si un tipo de clase tiene miembros y / o clases base con los tipos `t1, t2,...tN` , el tamaño debe ser al menos `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)` dados los puntos anteriores . Sin embargo, dependiendo de los requisitos de alineación de los miembros y las clases base, el compilador puede verse obligado a insertar relleno entre subobjetos, o al principio o al final del objeto completo.

```
struct AnInt      { int i; };
```

```
  // sizeof(AnInt)        == sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(AnInt)        == 4 (4).
struct TwoInts    { int i, j; };
  // sizeof(TwoInts)      >= 2 * sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(TwoInts)      == 8 (4 + 4).
struct IntAndChar { int i; char c; };
  // sizeof(IntAndChar)   >= sizeof(int) + sizeof(char)
  // Assuming a typical 32- or 64-bit system, sizeof(IntAndChar)   == 8 (4 + 1 +
padding).
struct AnIntDerived : AnInt { long long l; };
  // sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
  // Assuming a typical 32- or 64-bit system, sizeof(AnIntDerived) == 16 (4 + padding +
8).
```

- Si el relleno se inserta en un objeto debido a los requisitos de alineación, el tamaño será mayor que la suma de los tamaños de los miembros y las clases base. Con la alineación de `n` bytes, el tamaño normalmente será el múltiplo más pequeño de `n` que es más grande que el tamaño de todos los miembros y clases base. Cada miembro `memN` normalmente se coloca en una dirección que es un múltiplo de `alignof(memN)` , y `n` será típicamente la más grande `alignof` de todos los miembros `alignof` s. Debido a esto, si un miembro con una menor `alignof` es seguido por un miembro con una mayor `alignof` , existe la posibilidad de que el último miembro no se alinee correctamente si se coloca inmediatamente después del anterior. En este caso, el relleno (también conocido como *miembro de alineación*) se colocará entre los dos miembros, de modo que el último miembro pueda tener la alineación deseada. Por el contrario, si un miembro con una `alignof` más `alignof` es seguido por un miembro con una `alignof` más `alignof` , generalmente no será necesario `alignof` . Este proceso también se conoce como "embalaje".
  Debido a que las clases normalmente comparten la `alignof` de su miembro con la `alignof` más `alignof` , las clases típicamente se alinearán con la `alignof` del tipo integrado más grande que contengan directa o indirectamente.

```
// Assume sizeof(short) == 2, sizeof(int) == 4, and sizeof(long long) == 8.
// Assume 4-byte alignment is specified to the compiler.
struct Char { char c; };
  // sizeof(Char)                == 1 (sizeof(char))
struct Int  { int i; };
  // sizeof(Int)                 == 4 (sizeof(int))
struct CharInt { char c; int i; };
  // sizeof(CharInt)             == 8 (1 (char) + 3 (padding) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
  // sizeof(ShortIntCharInt)     == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                    3 (padding) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
  // sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                    1 (char) + 2 (padding) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
  // sizeof(ShortCharShortInt)   == 12 (2 (short) + 1 (char) + 1 (padding) + 2 (short) +
  //                                    2 (padding) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
  // sizeof(IntLLInt)            == 16 (4 (int) + 8 (long long) + 4 (int))
  // If packing isn't explicitly specified, most compilers will pack this as
  //   8-byte alignment, such that:
  // sizeof(IntLLInt)            == 24 (4 (int) + 4 (padding) + 8 (long long) +
  //                                    4 (int) + 4 (padding))
```

```
// Assume sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, and sizeof(IntLLInt) == 24.
// Assume default alignment: alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};
  // ShortChar3ArrShortInt has 4-byte alignment: alignof(int) >= alignof(char) &&
  //                                             alignof(int) >= alignof(short)
  // sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (padding) +
  //                                      2 (short) + 4 (int))
  // Note that t is placed at alignment of 2, not 4.  alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};
  // Large_1 has 4-byte alignment.
    // alignof(ShortIntCharInt) == alignof(int) == 4
    // alignof(b) == 1
    // Therefore, alignof(Large_1) == 4.
  // sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (padding) +
  //                        16 (ShortIntCharInt))
struct Large_2 {
    IntLLInt illi;
    float f;
    IntLLInt jmmj;
};
  // Large_2 has 8-byte alignment.
    // alignof(IntLLInt) == alignof(long long) == 8
    // alignof(float) == 4
    // Therefore, alignof(Large_2) == 8.
  // sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (padding) + 24 (IntLLInt))
```

C ++ 11

- Si se fuerza una alineación estricta con las alineaciones, se `alignas` relleno para forzar al tipo a cumplir con la alineación especificada, incluso cuando de lo contrario sería más pequeño. Por ejemplo, con la definición a continuación, los caracteres `Chars<5>` tendrán tres (o posiblemente más) bytes de relleno insertados al final, de modo que su tamaño total sea 8. No es posible que una clase con una alineación de 4 tenga un tamaño. de 5 porque sería imposible hacer una matriz de esa clase, por lo que el tamaño debe "redondearse" a un múltiplo de 4 insertando bytes de relleno.

```
// This type shall always be aligned to a multiple of 4.  Padding shall be inserted as
// needed.
// Chars<1>..Chars<4> are 4 bytes, Chars<5>..Chars<8> are 8 bytes, etc.
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; };

static_assert(sizeof(Chars<1>) == sizeof(Chars<4>), "Alignment is strict.\n");
```

- Si dos miembros no estáticos de una clase tienen el mismo especificador de acceso ,

entonces se garantiza que el que viene más tarde en el orden de declaración vendrá más adelante en la representación del objeto. Pero si dos miembros no estáticos tienen diferentes especificadores de acceso, su orden relativo dentro del objeto no se especifica.
- No se especifica en qué orden aparecen los subobjetos de la clase base dentro de un objeto, ya sea que aparezcan consecutivamente, y si aparecen antes, después o entre subobjetos miembros.

Tipos aritméticos

Tipos de caracteres estrechos

El tipo `unsigned char` utiliza todos los bits para representar un número binario. Por lo tanto, por ejemplo, si el `unsigned char` tiene una longitud de 8 bits, los 256 patrones de bits posibles de un objeto `char` representan los 256 valores diferentes {0, 1, ..., 255}. El número 42 está garantizado para ser representado por el patrón de bits `00101010` .

El tipo `signed char` no tiene bits de relleno, *es decir,* si el `signed char` tiene una longitud de 8 bits, tiene 8 bits de capacidad para representar un número.

Tenga en cuenta que estas garantías no se aplican a tipos que no sean tipos de caracteres estrechos.

Tipos enteros

Los tipos de enteros sin signo utilizan un sistema binario puro, pero pueden contener bits de relleno. Por ejemplo, es posible (aunque improbable) que un `unsigned int` tenga una longitud de 64 bits pero solo sea capaz de almacenar enteros entre 0 y 2^{32} - 1, ambos inclusive. Los otros 32 bits serían bits de relleno, que no deberían escribirse directamente.

Los tipos de enteros con signo utilizan un sistema binario con un bit de signo y posiblemente bits de relleno. Los valores que pertenecen al rango común de un tipo entero con signo y el tipo entero sin signo correspondiente tienen la misma representación. Por ejemplo, si el patrón de bits `0001010010101011` de un objeto `unsigned short` representa el valor `5291` , entonces también representa el valor `5291` cuando se interpreta como un objeto `short` .

Se define por la implementación si se utiliza el complemento de dos, el complemento de uno o la representación de magnitud de signo, ya que los tres sistemas satisfacen el requisito del párrafo anterior.

Tipos de punto flotante

La representación del valor de los tipos de punto flotante está definida por la implementación. Más comúnmente, los tipos `float` y `double` ajustan a IEEE 754 y tienen una longitud de 32 y 64 bits (así, por ejemplo, `float` tendría 23 bits de precisión que seguirían 8 bits de exponente y 1 bit de

signo). Sin embargo, la norma no garantiza nada. Los tipos de punto flotante a menudo tienen "representaciones de trampa", que causan errores cuando se usan en los cálculos.

Arrays

Un tipo de matriz no tiene relleno entre sus elementos. Por lo tanto, una matriz con el tipo de elemento `T` es solo una secuencia de objetos `T` dispuestos en memoria, en orden.

Una matriz multidimensional es una matriz de matrices, y lo anterior se aplica recursivamente. Por ejemplo, si tenemos la declaración.

```
int a[5][3];
```

entonces `a` es una matriz de 5 matrices de 3 `int` s. Por lo tanto, `a[0]` , que consiste en los tres elementos `a[0][0]` , `a[0][1]` , `a[0][2]` , se presenta en la memoria antes de `a[1]` , que consiste en de `a[1][0]` , `a[1][1]` , y `a[1][2]` . Esto se llama *fila* orden *mayor* .

Lea Diseño de tipos de objetos en línea: https://riptutorial.com/es/cplusplus/topic/9329/diseno-de-tipos-de-objetos

Capítulo 37: Ejemplos de servidor cliente

Examples

Hola servidor TCP

Permítanme comenzar diciendo que primero deben visitar la Guía de programación de red de Beej y leerlo rápidamente, lo que explica la mayoría de estas cosas con más detalle. Aquí crearemos un servidor TCP simple que dirá "Hola mundo" a todas las conexiones entrantes y luego las cerraremos. Otra cosa a tener en cuenta es que el servidor se comunicará de forma iterativa con los clientes, lo que significa un cliente a la vez. Asegúrese de revisar las páginas de manual relevantes, ya que pueden contener información valiosa sobre cada llamada de función y estructuras de socket.

Ejecutaremos el servidor con un puerto, así que también tomaremos un argumento para el número de puerto. Empecemos con el código

```
#include <cstring>    // sizeof()
#include <iostream>
#include <string>

// headers for socket(), getaddrinfo() and friends
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h>    // close()


int main(int argc, char *argv[])
{
    // Let's check if port number is supplied or not..
    if (argc != 2) {
        std::cerr << "Run program as 'program <port>'\n";
        return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backLog = 8;  // number of connections allowed on the incoming queue


    addrinfo hints, *res, *p;    // we need 2 pointers, res to hold and p to iterate over
    memset(&hints, 0, sizeof(hints));

    // for more explanation, man socket
    hints.ai_family   = AF_UNSPEC;    // don't specify which IP version to use yet
    hints.ai_socktype = SOCK_STREAM;  // SOCK_STREAM refers to TCP, SOCK_DGRAM will be?
    hints.ai_flags    = AI_PASSIVE;


    // man getaddrinfo
    int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
    if (gAddRes != 0) {
```

```
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
    }

    std::cout << "Detecting addresses" << std::endl;

    unsigned int numOfAddr = 0;
    char ipStr[INET6_ADDRSTRLEN];    // ipv6 length makes sure both ipv4/6 addresses can be
stored in this variable


    // Now since getaddrinfo() has given us a list of addresses
    // we're going to iterate over them and ask user to choose one
    // address for program to bind to
    for (p = res; p != NULL; p = p->ai_next) {
        void *addr;
        std::string ipVer;

        // if address is ipv4 address
        if (p->ai_family == AF_INET) {
            ipVer             = "IPv4";
            sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
            addr              = &(ipv4->sin_addr);
            ++numOfAddr;
        }

        // if address is ipv6 address
        else {
            ipVer              = "IPv6";
            sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
            addr               = &(ipv6->sin6_addr);
            ++numOfAddr;
        }

        // convert IPv4 and IPv6 addresses from binary to text form
        inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
        std::cout << "(" << numOfAddr << ") " << ipVer << " : " << ipStr
                  << std::endl;
    }

    // if no addresses found :(
    if (!numOfAddr) {
        std::cerr << "Found no host address to use\n";
        return -3;
    }

    // ask user to choose an address
    std::cout << "Enter the number of host address to bind with: ";
    unsigned int choice = 0;
    bool madeChoice     = false;
    do {
        std::cin >> choice;
        if (choice > (numOfAddr + 1) || choice < 1) {
            madeChoice = false;
            std::cout << "Wrong choice, try again!" << std::endl;
        } else
            madeChoice = true;
    } while (!madeChoice);
```

```
    p = res;

    // let's create a new socket, socketFD is returned as descriptor
    // man socket for more information
    // these calls usually return -1 as result of some error
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        freeaddrinfo(res);
        return -4;
    }


    // Let's bind address to our socket we've just created
    int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
    if (bindR == -1) {
        std::cerr << "Error while binding socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -5;
    }


    // finally start listening for connections on our socket
    int listenR = listen(sockFD, backLog);
    if (listenR == -1) {
        std::cerr << "Error while Listening on socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -6;
    }


    // structure large enough to hold client's address
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof(client_addr);


    const std::string response = "Hello World";


    // a fresh infinite loop to communicate with incoming connections
    // this will take client connections one at a time
    // in further examples, we're going to use fork() call for each client connection
    while (1) {

        // accept call will give us a new socket descriptor
        int newFD
          = accept(sockFD, (sockaddr *) &client_addr, &client_addr_size);
        if (newFD == -1) {
            std::cerr << "Error while Accepting on socket\n";
            continue;
        }

        // send call sends the data you specify as second param and it's length as 3rd param,
also returns how many bytes were actually sent
        auto bytes_sent = send(newFD, response.data(), response.length(), 0);
```

```
        close(newFD);
    }

    close(sockFD);
    freeaddrinfo(res);

    return 0;
}
```

El siguiente programa se ejecuta como

```
Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::
Enter the number of host address to bind with: 1
```

Hola cliente TCP

Este programa es complementario al programa Hello TCP Server, puede ejecutar cualquiera de ellos para verificar la validez de los demás. El flujo de programas es bastante común con el servidor Hello TCP, así que asegúrese de echarle un vistazo a eso también.

Aquí está el código -

```
#include <cstring>
#include <iostream>
#include <string>

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>


int main(int argc, char *argv[])
{
    // Now we're taking an ipaddress and a port number as arguments to our program
    if (argc != 3) {
        std::cerr << "Run program as 'program <ipaddress> <port>'\n";
        return -1;
    }


    auto &ipAddress = argv[1];
    auto &portNum   = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
```

```
    }

    if (p == NULL) {
        std::cerr << "No addresses found\n";
        return -3;
    }

    // socket() call creates a new socket and returns it's descriptor
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        return -4;
    }

    // Note: there is no bind() call as there was in Hello TCP Server
    // why? well you could call it though it's not necessary
    // because client doesn't necessarily has to have a fixed port number
    // so next call will bind it to a random available port number

    // connect() call tries to establish a TCP connection to the specified server
    int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
    if (connectR == -1) {
        close(sockFD);
        std::cerr << "Error while connecting socket\n";
        return -5;
    }

    std::string reply(15, ' ');

    // recv() call tries to get the response from server
    // BUT there's a catch here, the response might take multiple calls
    // to recv() before it is completely received
    // will be demonstrated in another example to keep this minimal
    auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
    if (bytes_recv == -1) {
        std::cerr << "Error while receiving bytes\n";
        return -6;
    }

    std::cout << "\nClient recieved: " << reply << std::endl;
    close(sockFD);
    freeaddrinfo(p);

    return 0;
}
```

Lea Ejemplos de servidor cliente en línea: https://riptutorial.com/es/cplusplus/topic/7177/ejemplos-de-servidor-cliente

Capítulo 38: El estándar ISO C ++

Introducción

En 1998, hubo una primera publicación de la norma que convierte a C ++ en un lenguaje estandarizado internamente. A partir de ese momento, C ++ ha evolucionado dando como resultado diferentes dialectos de C ++. En esta página, puede encontrar una descripción general de todos los diferentes estándares y sus cambios en comparación con la versión anterior. Los detalles sobre cómo usar estas funciones se describen en páginas más especializadas.

Observaciones

Cuando se menciona C ++, a menudo se hace referencia al "Estándar". Pero, ¿qué es exactamente ese estándar?

C ++ tiene una larga historia. Comenzó como un pequeño proyecto de Bjarne Stroustrup dentro de los Laboratorios Bell, a principios de los 90 se había vuelto bastante popular. Varias compañías creaban compiladores de C ++ para que los usuarios pudieran ejecutar sus compiladores de C ++ en una amplia gama de computadoras. Pero para facilitar esto, todos estos compiladores que compiten deberían compartir una sola definición del lenguaje.

En ese momento, el lenguaje C se había estandarizado con éxito. Esto significa que se escribió una descripción formal del lenguaje. Este se presentó al American National Standards Institute (ANSI), que abrió el documento para su revisión y posteriormente lo publicó en 1989. Un año después, la Organización Internacional de Estándares (ya que tendría diferentes siglas en diferentes idiomas, eligieron un formulario). , ISO, derivado de las islas griegas, que significa igual.) Adoptó la norma estadounidense como norma internacional.

Para C ++, estaba claro desde el principio que había un interés internacional. Se inició un grupo de trabajo dentro de ISO (conocido como WG21, dentro del Subcomité 22). Este grupo de trabajo elaboró un primer estándar alrededor de 1995. Pero como sabemos los programadores, no hay nada más peligroso para una entrega planificada que las funciones de última hora, y eso también le sucedió a C ++. En 1995, surgió una nueva biblioteca llamada STL, y las personas que trabajan en WG21 decidieron agregar una versión reducida al estándar borrador de C ++. Naturalmente, esto hizo que los plazos se perdieran y solo 3 años después el documento se convirtió en definitivo. ISO es una organización muy formal, por lo que el Estándar C ++ fue bautizado con el nombre poco comercial de ISO / IEC 14882. Como los estándares pueden actualizarse, esta versión exacta se conoció como 14882: 1998.

Y de hecho hubo una demanda para actualizar el Estándar. El estándar es un documento muy grueso, cuyo objetivo es describir exactamente cómo deberían funcionar los compiladores de C ++. Incluso vale la pena arreglar una pequeña ambigüedad, por lo que para 2003 se lanzó una actualización como 14882: 2003. Sin embargo, esto no agregó ninguna característica a C ++; Las nuevas características fueron programadas para la segunda actualización.

De manera informal, esta segunda actualización se conoció como C ++ 0x, porque no se sabía si duraría hasta 2008 o 2009. Bueno, esa versión también tuvo un ligero retraso, por lo que se convirtió en 14882: 2011.

Por suerte, WG21 decidió no dejar que eso volviera a suceder. C ++ 11 fue bien recibido y permitió un renovado interés en C ++. Entonces, para mantener el impulso, la tercera actualización pasó de la planificación a la publicación en 3 años, para convertirse en 14882: 2014.

El trabajo no se detuvo allí, tampoco. Se ha propuesto el estándar C ++ 17 y se ha iniciado el trabajo para C ++ 20.

Examples

Borradores de trabajo actuales

Todas las normas ISO publicadas están disponibles para la venta en la tienda ISO (http://www.iso.org). Sin embargo, los borradores de trabajo de los estándares de C ++ están disponibles públicamente de forma gratuita.

Las diferentes versiones de la norma:

- Próximamente (a veces denominado C ++ 20 o C ++ 2a): borrador de trabajo actual (versión HTML)
- Propuesta (a veces denominada C ++ 17 o C ++ 1z): borrador de trabajo de marzo de 2017 N4659 .
- C ++ 14 (a veces denominado C ++ 1y): noviembre de 2014, borrador de trabajo N4296
- C ++ 11 (a veces denominado C ++ 0x): febrero de 2011, borrador de trabajo N3242
- C ++ 03
- C ++ 98

C ++ 11

El estándar C ++ 11 es una extensión importante del estándar C ++. A continuación, puede encontrar una descripción general de los cambios, ya que se han agrupado en las Preguntas frecuentes de isocpp con enlaces a documentación más detallada.

Extensiones de lenguaje

Características generales

- auto
- decltype
- Declaración de rango de
- Listas de inicialización
- Sintaxis de inicialización uniforme y semántica.

- Referencias de valores y semánticas de movimiento.
- Lambdas
- noexcept para prevenir la propagación de excepciones
- constexpr
- nullptr - un puntero nulo literal
- Copiar y volver a emitir excepciones.
- Espacios de nombres en línea
- Literales definidos por el usuario

Las clases

- = predeterminado y = eliminar
- Control de movimiento por defecto y copia.
- Constructores delegantes
- Inicializadores de miembros en clase
- Constructores heredados
- Anular controles: anular
- Controles de anulación: final
- Operadores de conversión explícita

Otros tipos

- clase de enumeración
- long long - un entero más largo
- Tipos enteros extendidos
- Sindicatos generalizados
- POD generalizados

Plantillas

- Plantillas externas
- Alias de plantillas
- Plantillas variables
- Tipos locales como argumentos de plantilla

Concurrencia

- Modelo de memoria concurrente
- Inicialización dinámica y destrucción con concurrencia.
- Almacenamiento de hilo local

Características de varios idiomas

- ¿Cuál es el valor de __cplusplus para C ++ 11?

- Sintaxis de tipo de retorno de sufijo
- Previniendo el estrechamiento
- Corchetes de ángulo recto
- static_assert aserciones en tiempo de compilación
- Literales de cuerda cruda
- Atributos
- Alineación
- Características C99

Extensiones de biblioteca

General

- unique_ptr
- shared_ptr
- débil_ptr
- Recolección de basura abi
- tupla
- Tipo de rasgos
- función y enlace
- Expresiones regulares
- Utilidades de tiempo
- Generación de números aleatorios
- Asignadores de ámbito

Contenedores y algoritmos

- Mejoras de algoritmos.
- Mejoras de contenedores
- desordenados_ * contenedores
- std :: array
- forward_list

Concurrencia

- Trapos
- Exclusión mutua
- Cabellos
- Variables de condicion
- Atomística
- Futuros y promesas
- asíncrono
- Abandonando un proceso

C ++ 14

El estándar C ++ 14 a menudo se conoce como una corrección de errores para C ++ 11. Contiene solo una lista limitada de cambios, de los cuales la mayoría son extensiones de las nuevas funciones en C ++ 11. A continuación, puede encontrar una descripción general de los cambios, ya que se han agrupado en las Preguntas frecuentes de isocpp con enlaces a documentación más detallada.

Extensiones de lenguaje

- Literales binarios
- Deducción del tipo de retorno generalizado.
- decltype (auto)
- Capturas de lambda generalizadas
- Lambdas genericas
- Plantillas variables
- `constexpr` extendido
- El atributo `[[deprecated]]`
- Separadores de dígitos

Extensiones de biblioteca

- Bloqueo compartido
- Literales definidos por el usuario para `std::` types
- `std::make_unique`
- Tipo de transformación `_t` alias
- Abordar las tuplas por tipo (por ejemplo, `get<string>(t)`)
- Funciones de operador transparentes (por ejemplo, `greater<>(x)`)
- `std::quoted`

En desuso / Eliminado

- `std::gets` fue desaprobado en C ++ 11 y eliminado de C ++ 14
- `std::random_shuffle` está en desuso

C ++ 17

El estándar C ++ 17 es una característica completa y se ha propuesto para la estandarización. En los compiladores con soporte experimental para estas características, generalmente se conoce como C ++ 1z.

Extensiones de lenguaje

- Expresiones Fold
- declarando argumentos de plantilla que no son de tipo con `auto`
- Copia garantizada elision
- Deducción de parámetros de plantilla para constructores.
- Enlaces estructurados
- Espacios de nombres anidados compactos
- Nuevos atributos: `[[fallthrough]]` , `[[nodiscard]]` , `[[maybe_unused]]`
- Mensaje predeterminado para `static_assert`
- Inicializadores en `if` y `switch`
- Variables en linea
- `if constexpr`
- Garantías de orden de expresión.
- Asignación de memoria dinámica para datos sobre alineados

Extensiones de biblioteca

- `std::optional`
- `std::variant`
- `std::string_view`
- `merge()` y `extract()` para contenedores asociativos
- Una biblioteca de sistema de archivos con el encabezado `<filesystem>` .
- Versiones paralelas de la mayoría de los algoritmos estándar (en el encabezado `<algorithm>`).
- Adición de funciones matemáticas especiales en el encabezado `<cmath>` .
- Mover nodos entre map <>, unordered_map <>, set <> y unordered_set <>

C ++ 03

El estándar C ++ 03 trata principalmente los informes de defectos del estándar C ++ 98. Aparte de estos defectos, solo agrega una nueva característica.

Extensiones de lenguaje

- Inicialización del valor

C ++ 98

C ++ 98 es la primera versión estandarizada de C ++. Como se desarrolló como una extensión de C, se agregan muchas de las características que distinguen a C ++ de C.

Extensiones de lenguaje (con respecto a C89 / C90)

- Clases, Clases derivadas, funciones miembro virtuales, funciones miembro const.
- Sobrecarga de funciones, Sobrecarga del operador.
- Comentarios de una sola línea (se ha introducido en el C-languague con el estándar C99)
- Referencias
- nuevo y borrar
- tipo booleano (ha sido introducido en el C-languague con el estándar C99)
- plantillas
- espacios de nombres
- excepciones
- moldes específicos

Extensiones de biblioteca

- La biblioteca de plantillas estándar

C ++ 20

C ++ 20 es el próximo estándar de C ++, actualmente en desarrollo, basado en el estándar C ++ 17. Su progreso se puede seguir en el sitio web oficial de ISO cpp .

Las siguientes funciones son simplemente lo que se ha aceptado para la próxima versión del estándar C ++, orientado para 2020.

Extensiones de lenguaje

No se han aceptado extensiones de idioma por ahora.

Extensiones de biblioteca

No se han aceptado extensiones de biblioteca por ahora.

Lea El estándar ISO C ++ en línea: https://riptutorial.com/es/cplusplus/topic/2742/el-estandar-iso-c-plusplus

Capítulo 39: El puntero este

Observaciones

`this` puntero es una palabra clave para C ++, por lo que no se necesita una biblioteca para implementar esto. Y no olvides que `this` es un puntero! Así que no puedes hacer:

```
 this.someMember();
```

A medida que accede a las funciones o variables de los miembros desde los punteros, utilice el símbolo de flecha `->` :

```
this->someMember();
```

Otros enlaces útiles para una mejor comprensión de `this` puntero:

¿Qué es el puntero 'este'?

http://www.geeksforgeeks.org/this-pointer-in-c/

https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm

Examples

este puntero

Todas las funciones miembro no estáticas tienen un parámetro oculto, un puntero a una instancia de la clase, llamado `this` ; este parámetro se inserta silenciosamente al principio de la lista de parámetros, y se maneja por completo por el compilador. Cuando se accede a un miembro de la clase dentro de una función miembro, se accede de forma silenciosa a través de `this` ; esto permite que el compilador utilice una única función miembro no estática para todas las instancias, y permite que una función miembro llame a otras funciones miembro de forma polimórfica.

```
struct ThisPointer {
    int i;

    ThisPointer(int ii);

    virtual void func();

    int  get_i() const;
    void set_i(int ii);
};
ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
// As the constructor is responsible for creating the object, 'this' will not be "fully"
```

```
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int  ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int  ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

En un constructor, `this` se puede usar de manera segura para (implícita o explícitamente) acceder a cualquier campo que ya se haya inicializado, o cualquier campo en una clase principal; a la inversa, el acceso (implícito o explícito) a cualquier campo que aún no se haya inicializado, o cualquier campo en una clase derivada, es inseguro (debido a que la clase derivada aún no se ha construido y, por lo tanto, sus campos no están inicializados ni existen). Tampoco es seguro llamar a las funciones miembro virtuales a través de `this` en el constructor, ya que no se considerarán las funciones de clase derivadas (debido a que la clase derivada aún no se ha construido y, por lo tanto, su constructor aún no actualiza el vtable).

También tenga en cuenta que, mientras se encuentra en un constructor, el tipo de objeto es el tipo que ese constructor construye. Esto es cierto incluso si el objeto se declara como un tipo derivado. Por ejemplo, en el siguiente ejemplo, `ctd_good` y `ctd_bad` son tipo `CtorThisBase` dentro de `CtorThisBase()` , y tipo `CtorThis` dentro de `CtorThis()` , aunque su tipo canónico es `CtorThisDerived` . A medida que las clases más derivadas se construyen alrededor de la clase base, la instancia pasa gradualmente a través de la jerarquía de clases hasta que se trata de una instancia completamente construida de su tipo deseado.

```
class CtorThisBase {
    short s;

  public:
    CtorThisBase() : s(516) {}
};

class CtorThis : public CtorThisBase {
    int i, j, k;

  public:
```

```
    // Good constructor.
    CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
    CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
        virt_func();
    }

    virtual void virt_func() { i += 2; }
};

class CtorThisDerived : public CtorThis {
    bool b;

  public:
    CtorThisDerived()       : b(true) {}
    CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }
};

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

Con estas clases y funciones miembro:

- En el buen constructor, para `ctd_good` :
  - `CtorThisBase` se construye completamente cuando se `CtorThis` constructor `CtorThis` . Por lo tanto, `s` encuentra en un estado válido al inicializar `i` y, por lo tanto, se puede acceder a él.
  - `i` se inicializa antes de que se alcance `j(this->i)` . Por lo tanto, `i` está en un estado válido al inicializar `j` , y por lo tanto se puede acceder a él.
  - `j` se inicializa antes de alcanzar `k(j)` . Por lo tanto, `j` encuentra en un estado válido al inicializar `k` , y por lo tanto se puede acceder a él.
- En el constructor malo, para `ctd_bad` :
  - `k` se inicializa después de alcanzar `j(this->k)` . Por lo tanto, `k` está en un estado no válido al inicializar `j` , y acceder a él causa un comportamiento indefinido.
  - `CtorThisDerived` no se construye hasta después de que `CtorThis` se construya. Por lo tanto, `b` encuentra en un estado no válido al inicializar `k` , y acceder a él provoca un comportamiento indefinido.
  - El objeto `ctd_bad` sigue siendo un `CtorThis` hasta que sale `CtorThis()` , y no se actualiza para utilizar `CtorThisDerived` vtable 's hasta `CtorThisDerived()` . Por lo tanto, `virt_func()` llamará a `CtorThis::virt_func()` , independientemente de si está destinado a llamar a eso o `CtorThisDerived::virt_func()` .

Uso de este puntero para acceder a datos de miembros

En este contexto, el uso de `this` puntero no es completamente necesario, pero hará que su código sea más claro para el lector, al indicar que una función o variable determinada es un miembro de la clase. Un ejemplo en esta situación:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
  public:
    Class();
    ~Class();
    int getPrivateNumber () const;
  private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

Véalo en acción aquí .

Uso de este puntero para diferenciar entre datos de miembros y parámetros

Esta es una estrategia realmente útil para diferenciar los datos de los miembros de los parámetros ... Tomemos este ejemplo:

```
// Dog Class Example
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
* @class Dog
*   @member name
*       Dog's name
*   @function bark
*       Dog Barks!
*   @function getName
*       To Get Private
*       Name Variable
*/
class Dog
{
 public:
```

```
    Dog(std::string name);
    ~Dog();
    void  bark() const;
    std::string  getName() const;
 private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
    *  this->name is the
    *  name variable from
    *  the class dog . and
    *  name is from the
    *  parameter of the function
    */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
  cout << "BARK" << endl;
}

std::string  Dog::getName() const
{
    return this->name;
}


int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}
```

Puedes ver aquí en el constructor ejecutamos lo siguiente:

```
this->name = name;
```

Aquí puede ver que estamos asignando el nombre del parámetro al nombre de la variable privada de la clase Dog (this-> name).

Para ver la salida del código anterior: http://cpp.sh/75r7

este puntero CV-calificadores

`this` también puede ser calificado como CV, al igual que cualquier otro puntero. Sin embargo, debido a que `this` parámetro no aparece en la lista de parámetros, se requiere una sintaxis especial para esto; los calificadores cv se enumeran después de la lista de parámetros, pero antes del cuerpo de la función.

```
struct ThisCVQ {
    void no_qualifier()                {} // "this" is: ThisCVQ*
    void  c_qualifier() const          {} // "this" is: const ThisCVQ*
    void  v_qualifier() volatile       {} // "this" is: volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is: const volatile ThisCVQ*
};
```

Como `this` es un parámetro, una función puede ser sobrecargado basado en su `this` cv-calificador (s) .

```
struct CVOverload {
    int func()                { return    3; }
    int func() const          { return   33; }
    int func() volatile       { return  333; }
    int func() const volatile { return 3333; }
};
```

Cuando `this` es `const` (incluyendo `const volatile`), la función no puede escribir en las variables miembro a través de él, ya sea de forma implícita o explícita. La única excepción a esto son las variables miembro `mutable` , que pueden escribirse independientemente de la constancia. Debido a esto, `const` se utiliza para indicar que la función miembro no cambia el estado lógico del objeto (la forma en que el objeto aparece en el mundo exterior), incluso si modifica el estado físico (la forma en que el objeto se ve debajo del capó).

> El estado lógico es la forma en que el objeto aparece ante los observadores externos. No está directamente relacionado con el estado físico y, de hecho, puede que ni siquiera se almacene como estado físico. Mientras los observadores externos no puedan ver ningún cambio, el estado lógico es constante, incluso si volteas cada bit en el objeto.
>
> El estado físico, también conocido como estado a nivel de bits, es la forma en que el objeto se almacena en la memoria. Este es el meollo del objeto, los 1s y 0s en bruto que componen sus datos. Un objeto solo es físicamente constante si su representación en la memoria nunca cambia.

Tenga en cuenta que C ++ basa la `const` en el estado lógico, no en el estado físico.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somehow(const Param& p);

    // ...

  public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somehow(p);
    }

    void change_state(Param p) {
```

```
        modify_somehow(p);
        state_changed = true;
    }

    // Return some complex and/or expensive-to-calculate result.
    // As this has no reason to modify logical state, it is marked as "const".
    ResultType get_result() const;
};
ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    //  by caching the result, so it doesn't need to be recalculated every time the function
    //  is called.  This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}
```

Tenga en cuenta que, si bien técnicamente *podría* usar `const_cast` en `this` para que no esté calificado como `const_cast` , realmente, ***REALMENTE*** no debería, y debería usar `mutable` lugar. Un `const_cast` puede invocar un comportamiento indefinido cuando se usa en un objeto que en realidad *es* `const` , mientras que `mutable` está diseñado para ser seguro de usar. Sin embargo, es posible que se encuentre con este código extremadamente antiguo.

Una excepción a esta regla es la definición de accesores no calificados para CV en términos de `const` acceso; como se garantiza que el objeto no será `const` si se llama a la versión no calificada de CV, no hay riesgo de UB.

```
class CVAccessor {
    int arr[5];

  public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};
```

Esto evita la duplicación innecesaria de código.

Al igual que con los punteros regulares, si `this` es `volatile` (incluyendo `const volatile`), se carga desde la memoria cada vez que se accede, en lugar de ser almacenado en caché. Esto tiene los mismos efectos en la optimización que declarar cualquier otro puntero `volatile` , por lo que se debe tener cuidado.

Tenga en cuenta que si una instancia es cv-cualificado, las únicas funciones miembro que se permite el acceso a funciones son miembros cuyas `this` puntero es al menos tan cv-calificada

como la propia instancia:

- Las instancias no cv pueden acceder a cualquier función miembro.
- `const` instancias `const` pueden acceder a `const volatile` funciones `const` y `const volatile` .
- `volatile` instancias `volatile` pueden acceder a funciones `volatile` y `const volatile` .
- `const volatile` instancias `const volatile` pueden acceder a funciones `const volatile` .

Este es uno de los principios clave de la corrección `const` .

```
struct CVAccess {
    void    func()                {}
    void  func_c() const          {}
    void  func_v() volatile       {}
    void func_cv() const volatile {}
};

CVAccess cva;
cva.func();    // Good.
cva.func_c();  // Good.
cva.func_v();  // Good.
cva.func_cv(); // Good.

const CVAccess c_cva;
c_cva.func();    // Error.
c_cva.func_c();  // Good.
c_cva.func_v();  // Error.
c_cva.func_cv(); // Good.

volatile CVAccess v_cva;
v_cva.func();    // Error.
v_cva.func_c();  // Error.
v_cva.func_v();  // Good.
v_cva.func_cv(); // Good.

const volatile CVAccess cv_cva;
cv_cva.func();    // Error.
cv_cva.func_c();  // Error.
cv_cva.func_v();  // Error.
cv_cva.func_cv(); // Good.
```

este puntero ref-calificadores

C ++ 11

De manera similar a `this` calificadores de CV, también podemos aplicar *ref-calificadores* a `*this` . Los ref-calificadores se utilizan para elegir entre semántica de referencia normal y rvalor, lo que permite al compilador usar la semántica de copiar o mover según sea más apropiado, y se aplican a `*this` lugar de `this` .

Tenga en cuenta que a pesar de los ref-calificadores que usan la sintaxis de referencia, `this` sigue siendo un puntero. También tenga en cuenta que los ref-calificadores no cambian realmente el tipo de `*this` ; es más fácil describir y entender sus efectos mirándolos como si lo hicieran.

```
struct RefQualifiers {
```

```
    std::string s;

    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}

    // Normal version.
    void func() &  { std::cout << "Accessed on normal instance "    << s << std::endl; }
    // Rvalue version.
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }

    const std::string& still_a_pointer() &  { return this->s; }
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }
};

// ...

RefQualifiers rf("Fred");
rf.func();              // Output:  Accessed on normal instance Fred
RefQualifiers{}.func(); // Output:  Accessed on temporary instance The nameless one
```

Una función miembro no puede tener sobrecargas con y sin ref-calificadores; El programador tiene que elegir entre uno u otro. Afortunadamente, cv-qualifiers se puede utilizar junto con ref-qualifiers, lo que permite seguir las reglas de corrección de `const` .

```
struct RefCV {
    void func() &                {}
    void func() &&               {}
    void func() const&           {}
    void func() const&&          {}
    void func() volatile&        {}
    void func() volatile&&       {}
    void func() const volatile&  {}
    void func() const volatile&& {}
};
```

Lea El puntero este en línea: https://riptutorial.com/es/cplusplus/topic/7146/el-puntero-este

Capítulo 40: Enhebrado

Sintaxis

- hilo()
- hilo (hilo y & otros)
- Hilo explícito (Function && func, Args && ... args)

Parámetros

Parámetro	Detalles
`other`	Toma posesión de `other` , `other` ya no son dueños del hilo.
`func`	Función para llamar en un hilo separado
`args`	Argumentos para `func`

Observaciones

Algunas notas:

- Dos objetos `std::thread` **nunca** pueden representar el mismo hilo.
- Un objeto `std::thread` puede estar en un estado en el que no representa **ningún** hilo (es decir, después de un movimiento, después de llamar a `join` , etc.).

Examples

Operaciones de hilo

Cuando comienzas un hilo, se ejecutará hasta que termine.

A menudo, en algún momento, debe (posiblemente, el subproceso ya esté terminado) esperar a que el subproceso finalice, porque desea utilizar el resultado, por ejemplo.

```
int n;
std::thread thread{ calculateSomething, std::ref(n) };

//Doing some other stuff

//We need 'n' now!
//Wait for the thread to finish - if it is not already done
thread.join();

//Now 'n' has the result of the calculation done in the seperate thread
std::cout << n << '\n';
```

También puede `detach` el hilo, permitiendo que se ejecute libremente:

```
std::thread thread{ doSomething };

//Detaching the thread, we don't need it anymore (for whatever reason)
thread.detach();

//The thread will terminate when it is done, or when the main thread returns
```

Pasando una referencia a un hilo

No puede pasar una referencia (o una referencia `const`) directamente a un hilo porque `std::thread` los copiará / moverá. En su lugar, use `std::reference_wrapper` :

```
void foo(int& b)
{
    b = 10;
}

int a = 1;
std::thread thread{ foo, std::ref(a) }; //'a' is now really passed as reference

thread.join();
std::cout << a << '\n'; //Outputs 10
```

```
void bar(const ComplexObject& co)
{
    co.doCalculations();
}

ComplexObject object;
std::thread thread{ bar, std::cref(object) }; //'object' is passed as const&

thread.join();
std::cout << object.getResult() << '\n'; //Outputs the result
```

Creando un std :: thread

En C ++, los subprocesos se crean utilizando la clase std :: thread. Un hilo es un flujo de ejecución separado; es análogo a que un ayudante realice una tarea mientras usted realiza otra. Cuando se ejecuta todo el código en el hilo, *termina* . Cuando creas un hilo, necesitas pasar algo para ser ejecutado en él. Algunas cosas que puedes pasar a un hilo:

- Funciones libres
- Funciones miembro
- Objetos funcionales
- Expresiones lambda

Ejemplo de función libre: ejecuta una función en un subproceso separado (ejemplo en vivo):

```
#include <iostream>
```

```
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    // Create and execute the thread
    std::thread thread(foo, 10); // foo is the function to execute, 10 is the
                                 // argument to pass to it

    // Keep going; the thread is executed separately

    // Wait for the thread to finish; we stay here until it is done
    thread.join();

    return 0;
}
```

Ejemplo de función miembro: ejecuta una función miembro en un subproceso independiente (ejemplo en vivo):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(&Bar::foo, &bar, 10); // Pass 10 to member function

    // The member function will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Ejemplo de objeto funcional (ejemplo en vivo):

```
#include <iostream>
#include <thread>

class Bar
```

```
{
public:
    void operator()(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(bar, 10); // Pass 10 to functor object

    // The functor object will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Ejemplo de expresión lambda (ejemplo vivo):

```
#include <iostream>
#include <thread>

int main()
{
    auto lambda = [](int a) { std::cout << a << '\n'; };

    // Create and execute the thread
    std::thread thread(lambda, 10); // Pass 10 to the lambda expression

    // The lambda expression will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Operaciones en el hilo actual

`std::this_thread` es un `namespace` que tiene funciones para hacer cosas interesantes en el hilo actual desde la función desde la que se llama.

Función	Descripción
`get_id`	Devuelve el id del hilo
`sleep_for`	Duerme durante un tiempo determinado
`sleep_until`	Duerme hasta una hora determinada.

Función	Descripción
`yield`	Reprogramar subprocesos en ejecución, dando prioridad a otros subprocesos

---

Obteniendo el id de los hilos actuales usando `std::this_thread::get_id` :

```
void foo()
{
    //Print this threads id
    std::cout << std::this_thread::get_id() << '\n';
}

std::thread thread{ foo };
thread.join(); //'threads' id has now been printed, should be something like 12556

foo(); //The id of the main thread is printed, should be something like 2420
```

Durmiendo por 3 segundos usando `std::this_thread::sleep_for` :

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

std::thread thread{ foo };
foo.join();

std::cout << "Waited for 3 seconds!\n";
```

Dormir hasta 3 horas en el futuro usando `std::this_thread::sleep_until` :

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread thread{ foo };
thread.join();

std::cout << "We are now located 3 hours after the thread has been called\n";
```

Permitir que otros subprocesos tengan prioridad utilizando `std::this_thread::yield` :

```
void foo(int a)
{
    for (int i = 0; i < al ++i)
        std::this_thread::yield(); //Now other threads take priority, because this thread
                                   //isn't doing anything important

    std::cout << "Hello World!\n";
}

std::thread thread{ foo, 10 };
```

```
thread.join();
```

Usando std :: async en lugar de std :: thread

`std::async` también es capaz de hacer hilos. En comparación con `std::thread` , se considera menos potente pero más fácil de usar cuando solo desea ejecutar una función de forma asíncrona.

Asincrónicamente llamando a una función

```
#include <future>
#include <iostream>

unsigned int square(unsigned int i){
    return i*i;
}

int main() {
    auto f = std::async(std::launch::async, square, 8);
    std::cout << "square currently running\n"; //do something while square is running
    std::cout << "result is " << f.get() << '\n'; //getting the result from square
}
```

Errores comunes

- `std::async` devuelve un `std::future` que contiene el valor de retorno que la función calculará. Cuando ese `future` se destruye, espera hasta que se complete el subproceso, haciendo que su código sea efectivamente único. Esto se pasa por alto fácilmente cuando no necesita el valor de retorno:

  ```
  std::async(std::launch::async, square, 5);
  //thread already completed at this point, because the returning future got destroyed
  ```

- `std::async` funciona sin una política de lanzamiento, así que `std::async(square, 5);` compila Cuando haces eso, el sistema decide si quiere crear un hilo o no. La idea era que el sistema elige crear un subproceso a menos que ya esté ejecutando más subprocesos de los que puede ejecutar de manera eficiente. Desafortunadamente, las implementaciones comúnmente solo eligen no crear un subproceso en esa situación, por lo que es necesario anular ese comportamiento con `std::launch::async` que obliga al sistema a crear un subproceso.

- Cuidado con las condiciones de la carrera.

Más sobre asíncrono sobre futuros y promesas.

Asegurando un hilo siempre está unido

Cuando se invoca el destructor para `std::thread` se **debe** haber realizado una llamada a `join()` o

`detach()` . Si un subproceso no se ha unido o separado, se llamará de forma predeterminada a `std::terminate` . Usando RAII , esto es generalmente bastante simple de lograr:

```
class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    { }

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
    }

private:

    std::thread t_;
}
```

Esto se usa entonces como tal:

```
 void perform_work()
 {
     // Perform some work
 }

 void t()
 {
     thread_joiner j{std::thread(perform_work)};
     // Do some other calculations while thread is running
 } // Thread is automatically joined here
```

Esto también proporciona una excepción de seguridad; si hubiéramos creado nuestro hilo normalmente y el trabajo realizado en `t()` realizando otros cálculos hubiera generado una excepción, nunca se habría llamado `join()` en nuestro hilo y nuestro proceso habría terminado.

Reasignando objetos de hilo

Podemos crear objetos de hilo vacíos y asignarles trabajo más tarde.

Si asignamos un objeto de subproceso a otro subproceso activo, que se puede `joinable` , se llamará automáticamente a `std::terminate` antes de que se reemplace el subproceso.

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
//create 100 thread objects that do nothing
std::thread executors[100];
```

```
// Some code

// I want to create some threads now

for (int i = 0;i < 100;i++)
{
    // If this object doesn't have a thread assigned
    if (!executors[i].joinable())
         executors[i] = std::thread(foo);
}
```

Sincronizacion basica

La sincronización de subprocesos se puede realizar utilizando mutexes, entre otras primitivas de sincronización. La biblioteca estándar proporciona varios tipos de exclusión mutua, pero el más simple es `std::mutex` . Para bloquear un mutex, construyes un bloqueo en él. El tipo de bloqueo más simple es `std::lock_guard` :

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // Acquires a lock on the mutex
    // Synchronized code here
} // the mutex is automatically released when guard goes out of scope
```

Con `std::lock_guard` el mutex está bloqueado durante toda la vida útil del objeto de bloqueo. En los casos en que necesite controlar manualmente las regiones para el bloqueo, use `std::unique_lock` lugar:

```
std::mutex m;
void worker() {
    // by default, constructing a unique_lock from a mutex will lock the mutex
    // by passing the std::defer_lock as a second argument, we
    // can construct the guard in an unlocked state instead and
    // manually lock later.
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // the mutex is not locked yet!
    guard.lock();
    // critical section
    guard.unlock();
    // mutex is again released
}
```

Más estructuras de sincronización de hilos

Uso de variables de condición

Una variable de condición es una primitiva que se usa junto con un mutex para orquestar la comunicación entre hilos. Si bien no es la forma exclusiva ni más eficiente de lograr esto, puede ser una de las más sencillas para aquellos familiarizados con el patrón.

Uno espera en un `std::condition_variable` con un `std::unique_lock<std::mutex>` . Esto permite que el código examine de forma segura el estado compartido antes de decidir si proceder o no con la

adquisición.

A continuación se muestra un boceto productor-consumidor que usa `std::thread` , `std::condition_variable` , `std::mutex` , y algunos otros para hacer las cosas interesantes.

```
#include <condition_variable>
#include <cstddef>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>


int main()
{
    std::condition_variable cond;
    std::mutex mtx;
    std::queue<int> intq;
    bool stopped = false;

    std::thread producer{[&]()
    {
        // Prepare a random number generator.
        // Our producer will simply push random numbers to intq.
        //
        std::default_random_engine gen{};
        std::uniform_int_distribution<int> dist{};

        std::size_t count = 4006;
        while(count--)
        {
            // Always lock before changing
            // state guarded by a mutex and
            // condition_variable (a.k.a. "condvar").
            std::lock_guard<std::mutex> L{mtx};

            // Push a random int into the queue
            intq.push(dist(gen));

            // Tell the consumer it has an int
            cond.notify_one();
        }

        // All done.
        // Acquire the lock, set the stopped flag,
        // then inform the consumer.
        std::lock_guard<std::mutex> L{mtx};

        std::cout << "Producer is done!" << std::endl;

        stopped = true;
        cond.notify_one();
    }};

    std::thread consumer{[&]()
    {
        do{
            std::unique_lock<std::mutex> L{mtx};
            cond.wait(L,[&]()
```

```
            {
                // Acquire the lock only if
                // we've stopped or the queue
                // isn't empty
                return stopped || ! intq.empty();
            });

            // We own the mutex here; pop the queue
            // until it empties out.

            while( ! intq.empty())
            {
                const auto val = intq.front();
                intq.pop();

                std::cout << "Consumer popped: " << val << std::endl;
            }

            if(stopped){
                // producer has signaled a stop
                std::cout << "Consumer is done!" << std::endl;
                break;
            }

        }while(true);
    }};

    consumer.join();
    producer.join();

    std::cout << "Example Completed!" << std::endl;

    return 0;
}
```

Crear un grupo de subprocesos simple

Las primitivas de subprocesos de C ++ 11 son todavía un nivel relativamente bajo. Se pueden usar para escribir una construcción de nivel superior, como un grupo de hilos:

C ++ 14

```
struct tasks {
  // the mutex, condition variable and deque form a single
  // thread-safe triggered queue of tasks:
  std::mutex m;
  std::condition_variable v;
  // note that a packaged_task<void> can store a packaged_task<R>:
  std::deque<std::packaged_task<void()>> work;

  // this holds futures representing the worker threads being done:
  std::vector<std::future<void>> finished;

  // queue( lambda ) will enqueue the lambda into the tasks for the threads
  // to use.  A future of the type the lambda returns is given to let you get
  // the result out.
  template<class F, class R=std::result_of_t<F&()>>
  std::future<R> queue(F&& f) {
    // wrap the function object into a packaged task, splitting
```

```
    // execution from the return value:
    std::packaged_task<R()> p(std::forward<F>(f));

    auto r=p.get_future(); // get the return value before we hand off the task
    {
      std::unique_lock<std::mutex> l(m);
      work.emplace_back(std::move(p)); // store the task<R()> as a task<void()>
    }
    v.notify_one(); // wake a thread to work on the task

    return r; // return the future result of the task
  }

  // start N threads in the thread pool.
  void start(std::size_t N=1){
    for (std::size_t i = 0; i < N; ++i)
    {
      // each thread is a std::async running this->thread_task():
      finished.push_back(
        std::async(
          std::launch::async,
          [this]{ thread_task(); }
        )
      );
    }
  }
  // abort() cancels all non-started tasks, and tells every working thread
  // stop running, and waits for them to finish up.
  void abort() {
    cancel_pending();
    finish();
  }
  // cancel_pending() merely cancels all non-started tasks:
  void cancel_pending() {
    std::unique_lock<std::mutex> l(m);
    work.clear();
  }
  // finish enques a "stop the thread" message for every thread, then waits for them:
  void finish() {
    {
      std::unique_lock<std::mutex> l(m);
      for(auto&&unused:finished){
        work.push_back({});
      }
    }
    v.notify_all();
    finished.clear();
  }
  ~tasks() {
    finish();
  }
private:
  // the work that a worker thread does:
  void thread_task() {
    while(true){
      // pop a task off the queue:
      std::packaged_task<void()> f;
      {
        // usual thread-safe queue code:
        std::unique_lock<std::mutex> l(m);
        if (work.empty()){
```

```
          v.wait(l,[&]{return !work.empty();});
        }
        f = std::move(work.front());
        work.pop_front();
      }
      // if the task is invalid, it means we are asked to abort:
      if (!f.valid()) return;
      // otherwise, run the task:
      f();
    }
  }
};
```

`tasks.queue( []{ return "hello world"s; } )` devuelve `std::future<std::string>` , que cuando el objeto de tareas se ejecuta, se rellena con `hello world` .

Puede crear subprocesos ejecutando `tasks.start(10)` (que inicia 10 subprocesos).

El uso de `packaged_task<void()>` se debe simplemente a que no hay una `std::function` borrado de tipo que almacene tipos de solo movimiento. Escribir uno personalizado de esos probablemente sería más rápido que usar `packaged_task<void()>` .

Ejemplo vivo .

C ++ 11

En C ++ 11, reemplace `result_of_t<blah>` con `typename result_of<blah>::type` .

Más sobre Mutexes .

Almacenamiento de hilo local

El almacenamiento local de subprocesos se puede crear utilizando la palabra clave `thread_local` . Una variable declarada con el especificador `thread_local` se dice que tiene **duración de almacenamiento de subprocesos.**

- Cada subproceso en un programa tiene su propia copia de cada variable de subproceso local.
- Una variable de subproceso local con función (local) se inicializará la primera vez que el control pase por su definición. Dicha variable es implícitamente estática, a menos que se declare `extern` .
- Una variable de subproceso local con espacio de nombres o ámbito de clase (no local) se inicializará como parte del inicio de subproceso.
- Las variables locales del hilo se destruyen al terminar el hilo.
- Un miembro de una clase solo puede ser subproceso local si es estático. Por lo tanto, habrá una copia de esa variable por subproceso, en lugar de una copia por par (hilo, instancia).

Ejemplo:

```
void debug_counter() {
    thread_local int count = 0;
```

```
    Logger::log("This function has been called %d times by this thread", ++count);
}
```

Lea Enhebrado en línea: https://riptutorial.com/es/cplusplus/topic/699/enhebrado

Capítulo 41: Entrada / salida básica en c ++

Observaciones

La biblioteca estándar `<iostream>` define algunas secuencias para entrada y salida:

```
|stream | description                       |
|-------|-----------------------------------|
|cin    | standard input stream             |
|cout   | standard output stream            |
|cerr   | standard error (output) stream    |
|clog   | standard logging (output) stream  |
```

De las cuatro secuencias mencionadas anteriormente, `cin` se usa básicamente para la entrada del usuario y otras tres se usan para generar los datos. En general o en la mayoría de los entornos de codificación, `cin` (*entrada de consola* o *entrada* estándar) es teclado y `cout` (*salida de consola* o salida estándar) es monitor.

```
cin >> value

cin   - input stream
'>>'  - extraction operator
value - variable (destination)
```

`cin` aquí extrae la entrada introducida por el usuario y alimenta en valor variable. El valor se extrae solo después de que el usuario presiona la tecla ENTER.

```
cout << "Enter a value: "

cout              - output stream
'<<'              - insertion operator
"Enter a value: " - string to be displayed
```

`cout` here toma la cadena que se muestra y la inserta en la salida estándar o monitor

Todos los cuatro corrientes se encuentran en espacio de nombres estándar `std` por lo que necesitamos para imprimir `std::stream` de corriente de `stream` usarlo.

También hay un manipulador `std::endl` en el código. Se puede usar solo con flujos de salida. Inserta el carácter de final de línea `'\n'` en el flujo y lo vacía. Provoca producción inmediata.

Examples

entrada de usuario y salida estándar

```
#include <iostream>

int main()
```

```
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

Lea Entrada / salida básica en c ++ en línea:
https://riptutorial.com/es/cplusplus/topic/10683/entrada---salida-basica-en-c-plusplus

Capítulo 42: Enumeración

Examples

Declaración de enumeración básica

Las enumeraciones estándar permiten a los usuarios declarar un nombre útil para un conjunto de enteros. Los nombres se conocen colectivamente como enumeradores. Una enumeración y sus enumeradores asociados se definen de la siguiente manera:

```
enum myEnum
{
    enumName1,
    enumName2,
};
```

Una enumeración es un *tipo* , uno que es distinto de todos los otros tipos. En este caso, el nombre de este tipo es `myEnum` . Se espera que los objetos de este tipo asuman el valor de un enumerador dentro de la enumeración.

Los enumeradores declarados dentro de la enumeración son valores constantes del tipo de enumeración. Aunque los enumeradores están declarados dentro del tipo, el operador de alcance `::` no es necesario para acceder al nombre. Entonces el nombre del primer enumerador es `enumName1` .

C ++ 11

El operador de alcance se puede usar opcionalmente para acceder a un enumerador dentro de una enumeración. Así que `enumName1` también se puede deletrear `myEnum::enumName1` .

A los enumeradores se les asignan valores enteros que comienzan desde 0 y aumentan en 1 para cada enumerador en una enumeración. Entonces, en el caso anterior, `enumName1` tiene el valor 0, mientras que `enumName2` tiene el valor 1.

Los usuarios también pueden asignar un valor específico al usuario; este valor debe ser una expresión constante constante. Los enumeradores cuyos valores no se proporcionan explícitamente tendrán su valor establecido en el valor del enumerador anterior + 1.

```
enum myEnum
{
    enumName1 = 1, // value will be 1
    enumName2 = 2, // value will be 2
    enumName3,     // value will be 3, previous value + 1
    enumName4 = 7, // value will be 7
    enumName5,     // value will be 8
    enumName6 = 5, // value will be 5, legal to go backwards
    enumName7 = 3, // value will be 3, legal to reuse numbers
    enumName8 = enumName4 + 2, // value will be 9, legal to take prior enums and adjust them
};
```

Enumeración en declaraciones de cambio

Un uso común para los enumeradores es para las declaraciones de cambio y, por lo tanto, suelen aparecer en las máquinas de estado. De hecho, una característica útil de las declaraciones de cambio con enumeraciones es que si no se incluye una declaración predeterminada para el cambio, y no se han utilizado todos los valores de la enumeración, el compilador emitirá una advertencia.

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
       ...
    case middle:
       ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]
```

Iteración sobre una enumeración

No hay una función para iterar sobre la enumeración.

Pero hay varias formas

- para la `enum` con solo valores consecutivos:

  ```
  enum E {
      Begin,
      E1 = Begin,
      E2,
      // ..
      En,
      End
  };

  for (E e = E::Begin; e != E::End; ++e) {
      // Do job with e
  }
  ```

C ++ 11

con la `enum class` , el `operator ++` tiene que ser implementado:

```
E& operator ++ (E& e)
{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
```

```
    return e;
}
```

- usando un contenedor como `std::vector`

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

std::vector<E> build_all_E()
{
    const E all[] = {E1, E2, /*..*/ En};

    return std::vector<E>(all, all + sizeof(all) / sizeof(E));
}

std::vector<E> all_E = build_all_E();
```

y entonces

```
for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
    E e = *it;
    // Do job with e;
}
```

C ++ 11

- o `std::initializer_list` y una sintaxis más simple:

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*..*/ En};
```

y entonces

```
for (auto e : all_E) {
    // Do job with e
}
```

Enumerados con alcance

C ++ 11 introduce lo que se conoce como *enumeraciones de ámbito* . Estas son enumeraciones cuyos miembros deben ser calificados con `enumname::membername` . Las enums con alcance se declaran usando la sintaxis de `enum class` . Por ejemplo, para almacenar los colores en un arco iris:

```
enum class rainbow {
    RED,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
};
```

Para acceder a un color específico:

```
rainbow r = rainbow::INDIGO;
```

`enum class` no se pueden convertir implícitamente a `int` s sin una conversión. Así que `int x = rainbow::RED` no es válido.

Los enums con ámbito también le permiten especificar el *tipo subyacente* , que es el tipo utilizado para representar a un miembro. Por defecto es `int` . En un juego de Tic-Tac-Toe, puedes guardar la pieza como

```
enum class piece : char {
    EMPTY = '\0',
    X = 'X',
    O = 'O',
};
```

Como puede observar, las `enum` pueden tener una coma final después del último miembro.

Enumerar la declaración hacia adelante en C ++ 11

Enumeración de ámbito:

```
...
enum class Status; // Forward declaration
Status doWork(); // Use the forward declaration
...
enum class Status { Invalid, Success, Fail };
Status doWork() // Full declaration required for implementation
{
    return Status::Success;
}
```

Enumeración sin ámbito:

```
...
enum Status: int; // Forward declaration, explicit type required
Status doWork(); // Use the forward declaration
...
enum Status: int{ Invalid=0, Success, Fail }; // Must match forward declare type
static_assert( Success == 1 );
```

Puede encontrar un ejemplo detallado de archivos múltiples aquí: Ejemplo de comerciante de

frutas ciegas

Lea Enumeración en línea: https://riptutorial.com/es/cplusplus/topic/2796/enumeracion

Capítulo 43: Errores comunes de compilación / enlazador (GCC)

Examples

error: '***' no fue declarado en este alcance

Este error ocurre si se utiliza un objeto desconocido.

Variables

No compilar

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i is not in the scope of the main function

    return 0;
}
```

Fijar:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
    }

    return 0;
}
```

Funciones

> La mayoría de las veces, este error se produce si no se incluye el encabezado necesario (por ejemplo, utilizando `std::cout` sin `#include <iostream>`)

No compilar

```
#include <iostream>
```

```
int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Fijar:

```
#include <iostream>

void doCompile(); // forward declare the function

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

O:

```
#include <iostream>

void doCompile() // define the function before using it
{
    std::cout << "No!" << std::endl;
}

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}
```

Nota: El compilador interpreta el código de arriba a abajo (simplificación). Todo debe ser al menos declarado (o definido) antes de su uso.

referencia indefinida a `*** '

Este error del enlazador ocurre, si el enlazador no puede encontrar un símbolo usado. La mayoría de las veces, esto sucede si una biblioteca usada no está vinculada.

qmake:

```
LIBS += nameOfLib
```

cmake:

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

llamada g ++:

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

También se podría olvidar compilar y vincular todos los archivos `.cpp` utilizados (functionsModule.cpp define la función necesaria):

```
g++  -o binName main.o functionsModule.o
```

error fatal: ***: No existe tal archivo o directorio

El compilador no puede encontrar un archivo (un archivo de origen usa `#include "someFile.hpp"`).

qmake:

```
INCLUDEPATH += dir/Of/File
```

cmake:

```
include_directories(dir/Of/File)
```

llamada g ++:

```
g++ -o main main.cpp -Idir/Of/File
```

Lea Errores comunes de compilación / enlazador (GCC) en línea: https://riptutorial.com/es/cplusplus/topic/4256/errores-comunes-de-compilacion---enlazador--gcc-

Capítulo 44: Escriba palabras clave

Examples

clase

1. Introduce la definición de un tipo de clase .

```
class foo {
    int x;
  public:
    int get_x();
    void set_x(int new_x);
};
```

2. Introduce un *especificador de tipo elaborado,* que especifica que el siguiente nombre es el nombre de un tipo de clase. Si el nombre de la clase ya se ha declarado, se puede encontrar incluso si está oculto por otro nombre. Si el nombre de la clase no se ha declarado ya, se declara hacia adelante.

```
class foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
class baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
class foo {
    bar b;
    friend class baz; // elaborated type specifier refers to the class,
                      // not the function of the same name
  public:
    foo();
};
```

3. Introduce un parámetro de tipo en la declaración de una plantilla .

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. En la declaración de un parámetro de plantilla de plantilla , la `class` palabra clave precede al nombre del parámetro. Dado que el argumento para un parámetro de plantilla de plantilla solo puede ser una plantilla de clase, el uso de `class` aquí es redundante. Sin embargo, la gramática de C ++ lo requiere.

```
template <template <class T> class U>
//                          ^^^^^ "class" used in this sense here;
```

```
//                                  U is a template template parameter
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

5. Tenga en cuenta que sense 2 y sense 3 pueden combinarse en la misma declaración. Por ejemplo:

```
template <class T>
class foo {
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

C ++ 11

6. En la declaración o definición de una enumeración, declara que la enumeración es una enumeración de ámbito .

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

estructura

Intercambiable con `class` , excepto por las siguientes diferencias:

- Si un tipo de clase se define utilizando la `struct` palabras clave, la accesibilidad predeterminada de las bases y los miembros es `public` lugar de `private` .
- `struct` no se puede utilizar para declarar un parámetro de tipo de plantilla o un parámetro de plantilla de plantilla; Sólo la `class` puede.

enumerar

1. Introduce la definición de un tipo de enumeración .

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

C ++ 11

En C ++ 11, `enum` puede ser seguido opcionalmente por `class` o `struct` para definir una

enumeración de ámbito . Además, las enumeraciones con y sin ámbito pueden tener su tipo subyacente explícitamente especificado por `: T` después del nombre de enumeración, donde `T` refiere a un tipo entero.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Los enumeradores en `enum` normales también pueden estar precedidos por el operador del alcance, aunque todavía se considera que están en el alcance en que se definió la `enum` .

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. Introduce un *especificador de tipo elaborado,* que especifica que el siguiente nombre es el nombre de un tipo de enumeración previamente declarado. (Un especificador de tipo elaborado no se puede usar para declarar hacia delante un tipo de enumeración). Una enumeración se puede nombrar de esta manera incluso si está oculta por otro nombre.

   ```
   enum Foo { FOO };
   void Foo() {}
   Foo foo = FOO;      // ill-formed; Foo refers to the function
   enum Foo foo = FOO; // ok; Foo refers to the enum type
   ```

C ++ 11

3. Introduce una *declaración de enumeración opaca,* que declara una enumeración sin definirla. Puede volver a declarar una enumeración previamente declarada o declarar hacia adelante una enumeración que no se haya declarado previamente.

   Una enumeración que se declara primero como ámbito no se puede declarar más tarde como sin ámbito, o *viceversa.* Todas las declaraciones de una enumeración deben coincidir en el tipo subyacente.

   Al declarar hacia adelante una enumeración sin ámbito, el tipo subyacente debe especificarse explícitamente, ya que no se puede inferir hasta que se conozcan los valores de los enumeradores.

   ```
   enum class Format; // underlying type is implicitly int
   void f(Format f);
   ```

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction;    // ill-formed; must specify underlying type
```

Unión

1. Introduce la definición de un tipo de unión .

```
// Example is from POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

2. Introduce un *especificador de tipo elaborado,* que especifica que el siguiente nombre es el nombre de un tipo de unión. Si el nombre de la unión ya se ha declarado, se puede encontrar incluso si está oculto por otro nombre. Si el nombre del sindicato no se ha declarado ya, se declara hacia adelante.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
union baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
union foo {
    long l;
    union baz* b; // elaborated type specifier refers to the class,
                  // not the function of the same name
};
```

Lea Escriba palabras clave en línea: https://riptutorial.com/es/cplusplus/topic/7838/escriba-palabras-clave

Capítulo 45: Espacios de nombres

Introducción

Utilizado para evitar colisiones de nombres cuando se usan bibliotecas múltiples, un espacio de nombres es un prefijo declarativo para funciones, clases, tipos, etc.

Sintaxis

- *identificador de* espacio de nombres (*opt*) { *declaración-seq* }
- *identificador de* espacio de nombres en línea (*opt*) { *declaraciones-seq* } / * desde C ++ 11 * /
- inline (*opt*) namespace *atributo-specifier-seq identifier* (*opt*) { *declaraciones-seq* } / * desde C ++ 17 * /
- espacio de nombres *adjuntando-espacio de nombres-especificador* :: *identificador* { *declaración-seq* } / * desde C ++ 17 * /
- *identificador de* espacio de nombres = *calificador de espacio de nombres calificado* ;
- usando el espacio de nombres *nombre-anidado-especificador* (*opt*) *nombre-espacio-nombres* ;
- *atributo-especificador-seq* usando el espacio de nombres *nombre-anidado-especificador* (*opt*) *nombre-espacio-nombres* ; / * desde C ++ 11 * /

Observaciones

El `namespace` palabras clave tiene tres significados diferentes según el contexto:

1. Cuando le sigue un nombre opcional y una secuencia de declaraciones encerrada en corchetes, define un nuevo espacio de nombres o amplía un espacio de nombres existente con esas declaraciones. Si se omite el nombre, el espacio de nombres es un espacio de nombres sin nombre .

2. Cuando le sigue un nombre y un signo igual, declara un alias de espacio de nombres .

3. Cuando está precedido por el `using` y seguido de un nombre de espacio de nombres, forma una *directiva de uso* , que permite encontrar los nombres en el espacio de nombres dado por búsqueda de nombre no calificado (pero no vuelve a declarar esos nombres en el ámbito actual). Una *directiva de uso* no puede ocurrir en el alcance de la clase.

`using namespace std;` se desanima ¿Por qué? Porque `namespace std` es enorme! Esto significa que hay una alta probabilidad de que los nombres colisionen:

```
//Really bad!
using namespace std;

//Calculates p^e and outputs it to std::cout
void pow(double p, double e) { /*...*/ }
```

```
//Calls pow
pow(5.0, 2.0); //Error! There is already a pow function in namespace std with the same
signature,
               //so the call is ambiguous
```

Examples

¿Qué son los espacios de nombres?

Un espacio de nombres de C ++ es una colección de entidades de C ++ (funciones, clases, variables), cuyos nombres tienen el prefijo del nombre del espacio de nombres. Al escribir código dentro de un espacio de nombres, las entidades con nombre que pertenecen a ese espacio de nombres no tienen que tener un prefijo con el nombre del espacio de nombres, pero las entidades externas deben usar el nombre completo. El nombre completo tiene el formato `<namespace>::<entity>` . Ejemplo:

```
namespace Example
{
  const int test = 5;

  const int test2 = test + 12; //Works within `Example` namespace
}

const int test3 = test + 3; //Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; //Works; fully qualified name used.
```

Los espacios de nombres son útiles para agrupar definiciones relacionadas. Tomemos la analogía de un centro comercial. En general, un centro comercial se divide en varias tiendas, cada una de las cuales vende artículos de una categoría específica. Una tienda podría vender productos electrónicos, mientras que otra tienda podría vender zapatos. Estas separaciones lógicas en los tipos de tiendas ayudan a los compradores a encontrar los artículos que están buscando. Los espacios de nombres ayudan a los programadores de c ++, como los compradores, a encontrar las funciones, clases y variables que buscan mediante su organización de una manera lógica. Ejemplo:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
```

```
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
    class Slipper
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}
```

Hay un espacio de nombres único predefinido, que es el espacio de nombres global que no tiene nombre, pero puede ser denotado por `::` . Ejemplo:

```
void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar();   // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}
```

Haciendo espacios de nombres

Crear un espacio de nombres es realmente fácil:

```
//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}
```

Para llamar a la `bar` , primero debe especificar el espacio de nombres, seguido del operador de resolución de alcance `::`

```
Foo::bar();
```

Se permite crear un espacio de nombres en otro, por ejemplo:

```
namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
```

```
}
```

C ++ 17

El código anterior podría simplificarse a lo siguiente:

```
namespace A::B::C
{
    void bar() {}
}
```

Extendiendo espacios de nombres

Una característica útil de los `namespace` de `namespace` es que puede expandirlos (agregar miembros a ellos).

```
namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}
```

Usando directiva

La palabra clave 'usar' tiene tres sabores. Combinado con la palabra clave 'espacio de nombres' se escribe una 'directiva de uso':

Si no quieres escribir `Foo::` delante de todas las cosas en el espacio de nombres `Foo` , puedes usar `using namespace Foo;` para importar cada cosa de `Foo` .

```
namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK
```

También es posible importar entidades seleccionadas en un espacio de nombres en lugar de todo el espacio de nombres:

```
using Foo::bar;
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

Una advertencia: el `using namespace` en los archivos de encabezado se considera un estilo incorrecto en la mayoría de los casos. Si se hace esto, el espacio de nombres se importa en *cada* archivo que incluye el encabezado. Dado que no hay manera de "des- `using` " un espacio de nombres, esto puede conducir a la contaminación del espacio de nombres (más o símbolos inesperados en el espacio de nombres global) o, peor aún, conflictos. Vea este ejemplo para una ilustración del problema:

```
/***** foo.h *****/
namespace Foo
{
    class C;
}

/***** bar.h *****/
namespace Bar
{
    class C;
}

/***** baz.h *****/
#include "foo.h"
using namespace Foo;

/***** main.cpp *****/
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C
```

Una *directiva de uso* no puede ocurrir en el alcance de la clase.

Búsqueda dependiente del argumento

Cuando se llama a una función sin un calificador de espacio de nombres explícito, el compilador puede elegir llamar a una función dentro de un espacio de nombres si uno de los tipos de parámetros para esa función también se encuentra en ese espacio de nombres. Esto se denomina "búsqueda dependiente de argumentos" o ADL:

```
namespace Test
{
  int call(int i);

  class SomeClass {...};

  int call_too(const SomeClass &data);
}

call(5); //Fails. Not a qualified function name.

Test::SomeClass data;
```

```
call_too(data); //Succeeds
```

`call` falla porque ninguno de sus tipos de parámetros proviene del espacio de nombres de `Test` . `call_too` funciona porque `SomeClass` es miembro de `Test` y, por lo tanto, califica para las reglas ADL.

¿Cuándo no se produce ADL?

ADL no se produce si la búsqueda normal no cualificada encuentra un miembro de la clase, una función que se ha declarado en el ámbito del bloque o algo que no es del tipo de función. Por ejemplo:

```
void foo();
namespace N {
    struct X {};
    void foo(X ) { std::cout << '1'; }
    void qux(X ) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {
        foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
    }
};

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{});       // error: ADL is disabled and ::foo() doesn't take any arguments
}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}
```

Espacio de nombres en línea

C ++ 11

`inline namespace` incluye el contenido del espacio de nombres en línea en el espacio de nombres adjunto, por lo que

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

es mayormente equivalente a

```
namespace Outer
{

    namespace Inner
    {
        void foo();
    }

    using Inner::foo;
}
```

pero el elemento de `Outer::Inner::` y los asociados a `Outer::` son idénticos.

Así que lo siguiente es equivalente.

```
Outer::foo();
Outer::Inner::foo();
```

La alternativa `using namespace Inner;` No sería equivalente para algunas partes difíciles como la especialización de plantillas:

por

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
    template <>
    void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- El espacio de nombres en línea permite la especialización de `Outer::foo`

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      inline namespace Inner
      {
          template <typename T>
          void foo() { std::cout << "Generic"; }
      }
  }
  ```

- Mientras que el `using namespace` no permite la especialización de `Outer::foo`

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      namespace Inner
      {
  ```

```
        template <typename T>
        void foo() { std::cout << "Generic"; }
    }
    using namespace Inner;
    // Specialization of `Outer::foo` is not possible
    // it should be `Outer::Inner::foo`.
}
```

El espacio de nombres en línea es una forma de permitir que varias versiones cohabiten y por defecto a la en `inline`

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }

}
```

Y con uso

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();           // default version : MyNamespace::Version1::foo();
```

Sin nombre / espacios de nombres anónimos

Se puede usar un espacio de nombres sin nombre para garantizar que los nombres tengan un enlace interno (solo puede hacer referencia la unidad de traducción actual). Dicho espacio de nombres se define de la misma manera que cualquier otro espacio de nombres, pero sin el nombre:

```
namespace {
    int foo = 42;
}
```

`foo` solo es visible en la unidad de traducción en la que aparece.

Se recomienda nunca utilizar espacios de nombres sin nombre en los archivos de encabezado, ya que esto proporciona una versión del contenido para cada unidad de traducción en la que se incluye. Esto es especialmente importante si define globales no constantes.

```
// foo.h
namespace {
```

```
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";

// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< Will always print the empty string
```

Espacios de nombres anidados compactos

C ++ 17

```
namespace a {
  namespace b {
    template<class T>
    struct qualifies : std::false_type {};
  }
}

namespace other {
  struct bob {};
}

namespace a::b {
  template<>
  struct qualifies<::other::bob> : std::true_type {};
}
```

Puede introducir tanto la `a` y `b` espacios de nombres en un solo paso con el `namespace a::b` comenzando en C ++ 17.

Aliasing un espacio de nombres largo

Por lo general, se usa para cambiar el nombre o acortar referencias largas de nombres de nombres, como los componentes de una biblioteca.

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;


//    Both Type declarations are equivalent
boost::multiprecision::Number X   //    Writing the full namespace path, longer
Name1::Number Y                   //    using the name alias, shorter
```

Alcance de la Declaración de Alias

Declaración de alias se ven afectadas por las declaraciones de *uso* anteriores

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}


using namespace boost;

//   Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

Sin embargo, es más fácil confundirse sobre qué espacio de nombres está creando alias cuando tiene algo como esto:

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;

//    Not recommended as
//    its not explicitly clear whether Name1 refers to
//    numeric::multiprecision or boost::multiprecision
namespace Name1 = multiprecision;

//    For clarity, its recommended to use absolute paths
//    instead
namespace Name2 = numeric::multiprecision;
namespace Name3 = boost::multiprecision;
```

Alias del espacio de nombres

A un espacio de nombres se le puede dar un alias (*es decir,* otro nombre para el mismo espacio de nombres) usando el *identificador de* `namespace =` sintaxis. Se puede acceder a los miembros del espacio de nombres con alias calificándolos con el nombre del alias. En el siguiente ejemplo, el

espacio de nombres anidado `AReallyLongName::AnotherReallyLongName` es inconveniente de escribir, por lo que la función `qux` declara localmente un alias `N` Se puede acceder a los miembros de ese espacio de nombres simplemente usando `N::` .

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int foo();
        int bar();
        void baz(int x, int y);
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

Lea Espacios de nombres en línea: https://riptutorial.com/es/cplusplus/topic/495/espacios-de-nombres

Capítulo 46: Especificaciones de vinculación

Introducción

Una especificación de vinculación le dice al compilador que compile las declaraciones de una manera que les permita vincularse con declaraciones escritas en otro idioma, como C.

Sintaxis

- *cadena-literal* externa { *declaración-seq* (*opt*)}
- *declaración cadena-literal* externa

Observaciones

El estándar requiere que todos los compiladores admitan la `extern "C"` para permitir que C ++ sea compatible con C, y la `extern "C++"` , que se puede usar para anular una especificación de enlace adjunto y restaurar la predeterminada. Otras especificaciones de enlace soportadas están definidas por la implementación .

Examples

Controlador de señal para sistema operativo similar a Unix

Como el kernel llamará a un manejador de señales utilizando la convención de llamada C, debemos decirle al compilador que use la convención de llamada C al compilar la función.

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
    bind(...);
    listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {
            printf("Caught signal %d; shutting down\n", death_signal);
            break;
        }
        // ...
    }
}
```

Hacer un encabezado de biblioteca C compatible con C ++

El encabezado de la biblioteca de CA generalmente se puede incluir en un programa de C ++, ya que la mayoría de las declaraciones son válidas tanto en C como en C ++. Por ejemplo, considere

el siguiente `foo.h` :

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

La definición de `make_foo` se compila y distribuye por separado con el encabezado en forma de objeto.

Un programa de C ++ puede `#include <foo.h>` , pero el compilador no sabrá que la función `make_foo` está definida como un símbolo C, y probablemente intentará buscarlo con un nombre mutilado y no podrá localizarlo. Incluso si puede encontrar la definición de `make_foo` en la biblioteca, no todas las plataformas usan las mismas convenciones de llamada para C y C ++, y el compilador de C ++ usará la convención de llamada de C ++ al llamar a `make_foo` , lo que probablemente cause un error de segmentación si `make_foo` está esperando ser llamado con la convención de llamadas C.

La forma de solucionar este problema es envolver casi todas las declaraciones en el encabezado en un bloque `extern "C"` .

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);

#ifdef __cplusplus
}  /* end of "extern C" block */
#endif
```

Ahora, cuando se incluye `foo.h` de un programa en C, simplemente aparecerá como declaraciones ordinarias, pero cuando se incluye `foo.h` de un programa en C ++, `make_foo` estará dentro de un bloque `extern "C"` y el compilador sabrá si debe buscar un nombre no enredado y utilizar la convención de llamadas C.

Lea Especificaciones de vinculación en línea:
https://riptutorial.com/es/cplusplus/topic/9268/especificaciones-de-vinculacion

Capítulo 47: Especificadores de clase de almacenamiento

Introducción

Los especificadores de clase de almacenamiento son palabras clave que se pueden usar en declaraciones. No afectan el tipo de la declaración, pero generalmente modifican la forma en que se almacena la entidad.

Observaciones

Hay seis especificadores de clase de almacenamiento, aunque no todos en la misma versión del idioma: `auto` (hasta C ++ 11), `register` (hasta C ++ 17), `static` , `thread_local` (desde C ++ 11), `extern` y `mutable`

Según la norma,

> A lo sumo, un *especificador de clase de almacenamiento* aparecerá en un *decl-specifier-seq* dado , excepto que `thread_local` puede aparecer con `static` o `extern` .

Una declaración no puede contener ningún especificador de clase de almacenamiento. En ese caso, el idioma especifica un comportamiento predeterminado. Por ejemplo, de forma predeterminada, una variable declarada en el ámbito del bloque tiene implícitamente una duración de almacenamiento automático.

Examples

mudable

Un especificador que se puede aplicar a la declaración de un miembro de datos no estático y no de referencia de una clase. Un miembro mutable de una clase no es `const` incluso cuando el objeto es `const` .

```
class C {
    int x;
    mutable int times_accessed;
  public:
    C(): x(0), times_accessed(0) {
    }
    int get_x() const {
        ++times_accessed; // ok: const member function can modify mutable data member
        return x;
    }
    void set_x(int x) {
        ++times_accessed;
        this->x = x;
    }
```

```
};
```

C ++ 11

Un segundo significado para `mutable` se añadió en C ++ 11. Cuando sigue la lista de parámetros de un lambda, suprime la `const` implícita en el operador de llamada de función del lambda. Por lo tanto, un lambda mutable puede modificar los valores de las entidades capturadas por copia. Ver lambdas mutable para más detalles.

```
std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
                  [start]() mutable { return start++; });
    return result;
}
```

Tenga en cuenta que `mutable` *no* es un especificador de clase de almacenamiento cuando se utiliza de esta manera para formar un lambda mutable.

registro

C ++ 17

Un especificador de clase de almacenamiento que insinúa al compilador que una variable será muy utilizada. La palabra "registro" está relacionada con el hecho de que un compilador puede elegir almacenar una variable de este tipo en un registro de CPU para que se pueda acceder a ella en menos ciclos de reloj. Fue desaprobado a partir de C ++ 11.

```
register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}
```

Tanto las variables locales como los parámetros de función pueden ser declarados de `register` . A diferencia de C, C ++ no impone restricciones a lo que se puede hacer con una variable de `register` . Por ejemplo, es válido tomar la dirección de una variable de `register` , pero esto puede impedir que el compilador realmente almacene dicha variable en un registro.

C ++ 17

El `register` palabras clave está sin uso y reservado. Un programa que utiliza el `register` palabras clave está mal formado.

estático

El especificador de clase de almacenamiento `static` tiene tres significados diferentes.

1. Da enlace interno a una variable o función declarada en el ámbito del espacio de nombres.

```
// internal function; can't be linked to
static double semiperimeter(double a, double b, double c) {
    return (a + b + c)/2.0;
}
// exported to client
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. Declara que una variable tiene una duración de almacenamiento estática (a menos que sea `thread_local`). Las variables del ámbito del espacio de nombres son implícitamente estáticas. Una variable local estática se inicializa solo una vez, el primer control de tiempo pasa por su definición y no se destruye cada vez que se sale de su alcance.

```
void f() {
    static int count = 0;
    std::cout << "f has been called " << ++count << " times so far\n";
}
```

3. Cuando se aplica a la declaración de un miembro de la clase, declara que ese miembro es un miembro estático .

```
struct S {
    static S* create() {
        return new S;
    }
};
int main() {
    S* s = S::create();
}
```

Tenga en cuenta que en el caso de un miembro de datos estáticos de una clase, tanto 2 como 3 se aplican simultáneamente: la palabra clave `static` convierte al miembro en un miembro de datos estáticos y lo convierte en una variable con una duración de almacenamiento estática.

auto

C ++ 03

Declara una variable que tiene duración de almacenamiento automático. Es redundante, ya que la duración del almacenamiento automático ya es la predeterminada en el ámbito del bloque, y el especificador automático no está permitido en el ámbito del espacio de nombres.

```
void f() {
    auto int x; // equivalent to: int x;
    auto y;     // illegal in C++; legal in C89
}
auto int z;     // illegal: namespace-scope variable cannot be automatic
```

En C ++ 11, el significado de `auto` cambió completamente, y ya no es un especificador de clase de almacenamiento, sino que se usa para la deducción de tipos .

externo

El especificador de clase de almacenamiento `extern` puede modificar una declaración de una de las tres formas siguientes, según el contexto:

1. Se puede utilizar para declarar una variable sin definirla. Por lo general, esto se usa en un archivo de encabezado para una variable que se definirá en un archivo de implementación separado.

```
// global scope
int x;             // definition; x will be default-initialized
extern int y;      // declaration; y is defined elsewhere, most likely another TU
extern int z = 42; // definition; "extern" has no effect here (compiler may warn)
```

2. `constexpr` un enlace externo a una variable en el ámbito del espacio de nombres, incluso si `const` o `constexpr` hubieran provocado que tuviera un enlace interno.

```
// global scope
const int w = 42;            // internal linkage in C++; external linkage in C
static const int x = 42;     // internal linkage in both C++ and C
extern const int y = 42;     // external linkage in both C++ and C
namespace {
    extern const int z = 42; // however, this has internal linkage since
                             // it's in an unnamed namespace
}
```

3. Redecierra una variable en el ámbito del bloque si se declaró previamente con un enlace. De lo contrario, declara una nueva variable con vinculación, que es un miembro del espacio de nombres envolvente más cercano.

```
// global scope
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x;           // redeclares namespace-scope x
            std::cout << x << '\n'; // therefore, this prints 1, not 2
        }
    };
}
void g() {
    extern int y; // y has external linkage; refers to global y defined elsewhere
}
```

Una función también puede ser declarada `extern` , pero esto no tiene efecto. Generalmente se usa como una sugerencia para el lector de que una función declarada aquí se define en otra unidad de traducción. Por ejemplo:

```
void f();        // typically a forward declaration; f defined later in this TU
extern void g(); // typically not a forward declaration; g defined in another TU
```

En el código anterior, si `f` se cambiara a `extern` y `g` a no `extern` , no afectaría en absoluto la corrección o la semántica del programa, pero probablemente confundiría al lector del código.

Lea Especificadores de clase de almacenamiento en línea:
https://riptutorial.com/es/cplusplus/topic/9225/especificadores-de-clase-de-almacenamiento

Capítulo 48: Estructuras de datos en C ++

Examples

Implementación de listas enlazadas en C ++

Creación de un nodo de lista

```
class listNode
{
    public:
    int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

Creando clase de lista

```
class List
{
    public:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
    ~List();
};
```

Insertar un nuevo nodo al principio de la lista

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

Insertar un nuevo nodo al final de la lista

```
void List::insertAtEnd(int val) //inserting at end of list
{
    if(head==NULL)
    {
        insertAtBegin(val);
        return;
    }
    listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
    while(ptr->next!=NULL)
```

```
    {
        ptr=ptr->next;
    }
    ptr->next=newnode;
}
```

Insertar en una posición particular en la lista

```
void List::insertAtPos(int pos,int val)
{
    listNode *newnode=new listNode(val);
    if(pos==1)
    {
        //as head
        newnode->next=this->head;
        this->head=newnode;
        return;
    }
    pos--;
    listNode *ptr=this->head;
    while(ptr!=NULL && --pos)
    {
        ptr=ptr->next;
    }
    if(ptr==NULL)
    return;//not enough elements
    newnode->next=ptr->next;
    ptr->next=newnode;
}
```

Eliminar un nodo de la lista

```
void List::remove(int toBeRemoved)//removing an element
{
    if(this->head==NULL)
    return; //empty
    if(this->head->data==toBeRemoved)
    {
        //first node to be removed
        listNode *temp=this->head;
        this->head=this->head->next;
        delete(temp);
        return;
    }
    listNode *ptr=this->head;
    while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
    ptr=ptr->next;
    if(ptr->next==NULL)
    return;//not found
    listNode *temp=ptr->next;
    ptr->next=ptr->next->next;
    delete(temp);
}
```

Imprimir la lista

```
void List::print()//printing the list
```

```
{
    listNode *ptr=this->head;
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" " ;
        ptr=ptr->next;
    }
    cout<<endl;
}
```

Destructor para la lista

```
 List::~List()
{
    listNode *ptr=this->head,*next=NULL;
    while(ptr!=NULL)
    {
        next=ptr->next;
        delete(ptr);
        ptr=next;
    }
}
```

Lea Estructuras de datos en C ++ en línea:
https://riptutorial.com/es/cplusplus/topic/7485/estructuras-de-datos-en-c-plusplus

Capítulo 49: Estructuras de sincronización de hilos.

Introducción

Trabajar con hilos puede necesitar algunas técnicas de sincronización si los hilos interactúan. En este tema, puede encontrar las diferentes estructuras que proporciona la biblioteca estándar para resolver estos problemas.

Examples

std :: shared_lock

Un shared_lock puede usarse junto con un bloqueo único para permitir múltiples lectores y escritores exclusivos.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
    public:
        string getPhoneNo( const std::string & name )
        {
            shared_lock<shared_timed_mutex> r(_protect);
            auto it =  _phonebook.find( name );
            if ( it == _phonebook.end() )
                return (*it).second;
            return "";
        }
        void addPhoneNo ( const std::string & name, const std::string & phone )
        {
            unique_lock<shared_timed_mutex> w(_protect);
            _phonebook[name] = phone;
        }

        shared_timed_mutex _protect;
        unordered_map<string,string>  _phonebook;
    };
```

std :: call_once, std :: once_flag

`std::call_once` garantiza la ejecución de una función exactamente una vez por subprocesos de la competencia. `std::system_error` en caso de que no pueda completar su tarea.

Se utiliza junto con s `td::once_flag` .

```
#include <mutex>
#include <iostream>

std::once_flag flag;
void do_something(){
      std::call_once(flag, [](){std::cout << "Happens once" << std::endl;});

      std::cout << "Happens every time" << std::endl;
}
```

Bloqueo de objetos para un acceso eficiente.

A menudo desea bloquear todo el objeto mientras realiza varias operaciones en él. Por ejemplo, si necesita examinar o modificar el objeto utilizando *iteradores* . Siempre que necesite llamar a múltiples funciones miembro, generalmente es más eficiente bloquear todo el objeto en lugar de funciones individuales.

Por ejemplo:

```
class text_buffer
{
    // for readability/maintainability
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

public:
    // This returns a scoped lock that can be shared by multiple
    // readers at the same time while excluding any writers
    [[nodiscard]]
    reading_lock lock_for_reading() const { return reading_lock(mtx); }

    // This returns a scoped lock that is exclusing to one
    // writer preventing any readers
    [[nodiscard]]
    updates_lock lock_for_updates() { return updates_lock(mtx); }

    char* data() { return buf; }
    char const* data() const { return buf; }

    char* begin() { return buf; }
    char const* begin() const { return buf; }

    char* end() { return buf + sizeof(buf); }
    char const* end() const { return buf + sizeof(buf); }

    std::size_t size() const { return sizeof(buf); }

private:
    char buf[1024];
    mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

Al calcular una suma de comprobación, el objeto está bloqueado para la lectura, lo que permite que otros subprocesos que desean leer del objeto al mismo tiempo lo hagan.

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
    auto lock = buf.lock_for_reading();

    for(auto c: buf)
        sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

    return sum;
}
```

Al borrar el objeto, se actualizan sus datos internos, por lo que se debe hacer utilizando un bloqueo exclusivo.

```
void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}
```

Al obtener más de un bloqueo, se debe tener cuidado de adquirir siempre los bloqueos en el mismo orden para todos los subprocesos.

```
void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

    std::copy(std::begin(input), std::end(input), std::begin(output));
}
```

nota: esto se hace mejor usando std :: deferred :: lock y llamando a std :: lock

std :: condition_variable_any, std :: cv_status

Una generalización de `std::condition_variable` , `std::condition_variable_any` funciona con cualquier tipo de estructura BasicLockable.

`std::cv_status` como estado de retorno para una variable de condición tiene dos códigos de retorno posibles:

- std :: cv_status :: no_timeout: No hubo tiempo de espera, se notificó la variable de condición
- std :: cv_status :: no_timeout: variable de condición agotada por el tiempo de espera

Lea Estructuras de sincronización de hilos. en línea:
https://riptutorial.com/es/cplusplus/topic/9794/estructuras-de-sincronizacion-de-hilos-

Capítulo 50: Excepciones

Examples

Atrapando excepciones

Se utiliza un bloque `try/catch` para capturar excepciones. El código en la sección de `try` es el código que puede generar una excepción, y el código en la (s) cláusula (s) de `catch` maneja la excepción.

```
#include <iostream>
#include <string>
#include <stdexcept>

int main() {
  std::string str("foo");

  try {
      str.at(10); // access element, may throw std::out_of_range
  } catch (const std::out_of_range& e) {
      // what() is inherited from std::exception and contains an explanatory message
      std::cout << e.what();
  }
}
```

Se pueden usar múltiples cláusulas `catch` para manejar múltiples tipos de excepciones. Si hay varias cláusulas `catch` , el mecanismo de manejo de excepciones intenta hacerlas coincidir **en el orden** en que aparecen en el código:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

Las clases de excepción que se derivan de una clase base común se pueden capturar con una única cláusula `catch` para la clase base común. El ejemplo anterior puede reemplazar las dos cláusulas `catch` para `std::length_error` y `std::out_of_range` con una sola cláusula para `std:exception` :

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::exception& e) {
```

```
    std::cout << e.what();
}
```

Debido a que las cláusulas de `catch` se prueban en orden, asegúrese de escribir primero más cláusulas de captura específicas, de lo contrario, es posible que nunca se llame a su código de control de excepciones:

```
try {
    /* Code throwing exceptions omitted. */
} catch (const std::exception& e) {
    /* Handle all exceptions of type std::exception. */
} catch (const std::runtime_error& e) {
    /* This block of code will never execute, because std::runtime_error inherits
       from std::exception, and all exceptions of type std::exception were already
       caught by the previous catch clause. */
}
```

Otra posibilidad es el controlador catch-all, que capturará cualquier objeto lanzado:

```
try {
    throw 10;
} catch (...) {
    std::cout << "caught an exception";
}
```

Excepción de recirculación (propagación)

A veces, desea hacer algo con la excepción que captura (como escribir en el registro o imprimir una advertencia) y dejar que suba hasta el alcance superior para ser manejado. Para hacerlo, puede volver a lanzar cualquier excepción que detecte:

```
try {
    ... // some code here
} catch (const SomeException& e) {
    std::cout << "caught an exception";
    throw;
}
```

Utilizando el `throw;` sin argumentos volverá a lanzar la excepción capturada actualmente.

C ++ 11

Para volver a emitir un `std::exception_ptr` administrado, la biblioteca estándar de C ++ tiene la función `rethrow_exception` que se puede usar al incluir el encabezado `<exception>` en su programa.

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
    try {
```

```
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```

Función Try Blocks En constructor

La única forma de capturar una excepción en la lista de inicializadores:

```
struct A : public B
{
    A() try : B(), foo(1), bar(2)
    {
        // constructor body
    }
    catch (...)
    {
        // exceptions from the initializer list and constructor are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }

private:
    Foo foo;
    Bar bar;
};
```

Función Try Block para la función regular

```
void function_with_try_block()
try
{
    // try block body
}
catch (...)
{
    // catch block body
}
```

Que es equivalente a

```
void function_with_try_block()
```

```
{
    try
    {
        // try block body
    }
    catch (...)
    {
        // catch block body
    }
}
```

Tenga en cuenta que para los constructores y destructores, el comportamiento es diferente, ya que el bloque catch vuelve a lanzar una excepción de todos modos (el que se atrapa si no hay otro lanzamiento en el cuerpo del bloque catch).

Se permite que la función `main` tenga un bloque de prueba de función como cualquier otra función, pero el bloque de prueba de la función `main` no detectará las excepciones que se producen durante la construcción de una variable estática no local o la destrucción de cualquier variable estática. En su lugar, se llama `std::terminate` .

Función Try Blocks En Destructor

```
struct A
{
    ~A() noexcept(false) try
    {
        // destructor body
    }
    catch (...)
    {
        // exceptions of destructor body are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }
};
```

Tenga en cuenta que, aunque esto es posible, hay que tener mucho cuidado al lanzar desde el destructor, ya que si un destructor llamado durante el desenrollado de la pila lanza una excepción, se llama a `std::terminate` .

Mejores prácticas: tirar por valor, atrapar por referencia constante

En general, se considera una buena práctica lanzar por valor (en lugar de por puntero), pero capturar por (const) referencia.

```
try {
    // throw new std::runtime_error("Error!");   // Don't do this!
    // This creates an exception object
    // on the heap and would require you to catch the
    // pointer and manage the memory yourself. This can
    // cause memory leaks!

    throw std::runtime_error("Error!");
} catch (const std::runtime_error& e) {
```

```
    std::cout << e.what() << std::endl;
}
```

Una razón por la cual la captura por referencia es una buena práctica es que elimina la necesidad de reconstruir el objeto cuando se pasa al bloque catch (o cuando se propaga a otros bloques catch). La captura por referencia también permite que las excepciones se manejen de forma polimórfica y evita el corte de objetos. Sin embargo, si está volviendo a generar una excepción (como `throw e;` vea el ejemplo a continuación), todavía puede obtener el corte de objetos porque la `throw e;` La declaración hace una copia de la excepción a medida que se declara el tipo:

```
#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // "virtual" keyword is optional here
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "First catch block: " << e.what() << std::endl;
            // Output ==> First catch block: DerivedException

            throw e; // This changes the exception to BaseException
                     // instead of the original DerivedException!
        }
    } catch (const BaseException& e) {
        std::cout << "Second catch block: " << e.what() << std::endl;
        // Output ==> Second catch block: BaseException
    }
    return 0;
}
```

Si está seguro de que no va a hacer nada para cambiar la excepción (como agregar información o modificar el mensaje), la captura por referencia constante permite al compilador realizar optimizaciones y mejorar el rendimiento. Pero esto todavía puede causar el empalme de objetos (como se ve en el ejemplo anterior).

Advertencia: tenga cuidado de lanzar excepciones no intencionadas en `catch` bloques de `catch` , especialmente relacionados con la asignación de memoria o recursos adicionales. Por ejemplo, la construcción de `logic_error` , `runtime_error` o sus subclases puede `bad_alloc` debido a que la memoria se está agotando al copiar la cadena de excepción, las secuencias de E / S pueden lanzarse durante el registro con el conjunto de máscaras de excepción respectivas, etc.

La excepción jerarquizada

C ++ 11

Durante el manejo de excepciones, hay un caso de uso común cuando detecta una excepción genérica de una función de bajo nivel (como un error del sistema de archivos o un error de transferencia de datos) y lanza una excepción de alto nivel más específica que indica que alguna operación de alto nivel podría no se realiza (como no poder publicar una foto en la Web). Esto permite que el manejo de excepciones reaccione a problemas específicos con operaciones de alto nivel y también permite, al tener solo un mensaje de error, al programador encontrar un lugar en la aplicación donde se produjo una excepción. La desventaja de esta solución es que la excepción callstack se trunca y la excepción original se pierde. Esto obliga a los desarrolladores a incluir manualmente el texto de la excepción original en uno recién creado.

Las excepciones anidadas intentan resolver el problema adjuntando la excepción de bajo nivel, que describe la causa, a una excepción de alto nivel, que describe lo que significa en este caso particular.

`std::nested_exception` permite anidar excepciones gracias a `std::throw_with_nested` :

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } catch (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << '\n';
    } catch (...) {
        std::cerr << "Unkown exception\n";
    }
}

void print_current_exception_with_nested(int level =  0)
{
    try {
        throw;
    } catch (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // recursion
        }
    } catch (...) {
```

```
        //Empty // End recursion
    }
}

// sample function that catches an exception and wraps it in a nested exception
void open_file(const std::string& s)
{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch(...) {
        std::throw_with_nested(MyException{"Couldn't open " + s});
    }
}

// sample function that catches an exception and wraps it in a nested exception
void run()
{
    try {
        open_file("nonexistent.file");
    } catch(...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

// runs the sample function above and prints the caught exception
int main()
{
    try {
        run();
    } catch(...) {
        print_current_exception_with_nested();
    }
}
```

Salida posible:

```
exception: run() failed
 MyException: Couldn't open nonexistent.file
  exception: basic_ios::clear
```

Si trabaja solo con excepciones heredadas de `std::exception` , el código puede incluso simplificarse.

std :: uncaught_exceptions

c ++ 17

C ++ 17 introduce `int std::uncaught_exceptions()` (para reemplazar la limitada `bool std::uncaught_exception()`) para saber cuántas excepciones actualmente no se detectan. Eso permite que una clase determine si se destruye durante el desenrollado de una pila o no.

```
#include <exception>
#include <string>
#include <iostream>
```

```
// Apply change on destruction:
// Rollback in case of exception (failure)
// Else Commit (success)
class Transaction
{
public:
    Transaction(const std::string& s) : message(s) {}
    Transaction(const Transaction&) = delete;
    Transaction& operator =(const Transaction&) = delete;
    void Commit() { std::cout << message << ": Commit\n"; }
    void RollBack() noexcept(true) { std::cout << message << ": Rollback\n"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // May throw.
        } else { // current stack unwinding
            RollBack();
        }
    }

private:
    std::string message;
    int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
public:
    ~Foo() {
        try {
            Transaction transaction("In ~Foo"); // Commit,
                                                // even if there is an uncaught exception
            //...
        } catch (const std::exception& e) {
            std::cerr << "exception/~Foo:" << e.what() << std::endl;
        }
    }
};

int main()
{
    try {
        Transaction transaction("In main"); // RollBack
        Foo foo; // ~Foo commit its transaction.
        //...
        throw std::runtime_error("Error");
    } catch (const std::exception& e) {
        std::cerr << "exception/main:" << e.what() << std::endl;
    }
}
```

Salida:

```
In ~Foo: Commit
In main: Rollback
exception/main:Error
```

Excepción personalizada

No debe lanzar valores brutos como excepciones, en su lugar, utilice una de las clases de excepción estándar o cree las suyas propias.

Tener su propia clase de excepción heredada de `std::exception` es una buena manera de hacerlo. Aquí hay una clase de excepción personalizada que hereda directamente de `std::exception` :

```
#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset
    std::string error_message;      ///< Error message

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
        {}

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns a pointer to the (constant) error description.
     *  @return A pointer to a const char*. The underlying memory
     *  is in possession of the Except object. Callers must
     *  not attempt to free the memory.
     */
    virtual const char* what() const throw () {
       return error_message.c_str();
    }

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }
```

```
};
```

Un ejemplo de lanzar catch:

```
try {
    throw(Except("Couldn't do what you were expecting", -12, -34));
} catch (const Except& e) {
    std::cout<<e.what()
             <<"\nError number: "<<e.getErrorNumber()
             <<"\nError offset: "<<e.getErrorOffset();
}
```

Como no solo está lanzando un mensaje de error simple, sino también algunos otros valores que representan el error exactamente, su manejo de errores se vuelve mucho más eficiente y significativo.

Hay una clase de excepción que te permite manejar bien los mensajes de error:
`std::runtime_error`

También puedes heredar de esta clase:

```
#include <stdexcept>

class Except: virtual public std::runtime_error {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        std::runtime_error(msg)
        {
            error_number = err_num;
            error_offset = err_off;

        }

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
```

```
    }

    /**Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Tenga en cuenta que no he anulado la función `what()` de la clase base (`std::runtime_error`), es decir, `std::runtime_error` la versión de la clase base de `what()` . Puedes anularlo si tienes otra agenda.

Lea Excepciones en línea: https://riptutorial.com/es/cplusplus/topic/1354/excepciones

Capítulo 51: Expresiones Fold

Observaciones

Fold Expressions son compatibles con los siguientes operadores

+	-	*	/	%	\ ^	Y	|	<<	>>		
+ =	- =	* =	/ =	% =	\ ^ =	& =	| =	<< =	>> =	=	
==	! =	<	>	<=	> =	&&	||	,	. *	-> *	

Al plegar una secuencia vacía, una expresión de plegado está mal formada, excepto por los siguientes tres operadores:

Operador	Valor cuando el paquete de parámetros está vacío
&&	cierto
||	falso
,	vacío()

Examples

Pliegues Unarios

Los pliegues únicos se utilizan para *plegar* paquetes de parámetros sobre un operador específico. Hay 2 tipos de pliegues únicos:

- Unary **Left** Fold `(... op pack)` que se expande de la siguiente manera:

```
((Pack1 op Pack2) op ...) op PackN
```

- Unary **Right** Fold `(pack op ...)` que se expande de la siguiente manera:

```
Pack1 op (... (Pack(N-1) op PackN))
```

Aquí hay un ejemplo

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //Unary left fold
    //return (args + ...); //Unary right fold
```

```
    // The two are equivalent if the operator is associative.
    // For +, ((1+2)+3) (left fold) == (1+(2+3)) (right fold)
    // For -, ((1-2)-3) (left fold) != (1-(2-3)) (right fold)
}

int result = sum(1, 2, 3); //  6
```

Pliegues binarios

Los pliegues binarios son básicamente pliegues únicos , con un argumento adicional.

Hay 2 tipos de pliegues binarios:

- Doblado **izquierdo** binario - `(value op ... op pack)` - Expande de la siguiente manera:

  ```
  (((Value op Pack1) op Pack2) op ...) op PackN
  ```

- Plegado **derecho** binario `(pack op ... op value)` - Se expande de la siguiente manera:

  ```
  Pack1 op (... op (Pack(N-1) op (PackN op Value)))
  ```

Aquí hay un ejemplo:

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //Binary left fold
    // Note that a binary right fold cannot be used
    // due to the lack of associativity of operator-
}

int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

Doblando sobre una coma

Es una operación común la necesidad de realizar una función particular sobre cada elemento en un paquete de parámetros. Con C ++ 11, lo mejor que podemos hacer es:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Pero con una expresión de doblez, lo anterior se simplifica muy bien para:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
```

```
}
```

No requiere caldera críptica.

Lea Expresiones Fold en línea: https://riptutorial.com/es/cplusplus/topic/2676/expresiones-fold

Capítulo 52: Expresiones regulares

Introducción

Las expresiones regulares (a veces llamadas expresiones regulares o expresiones regulares) son una sintaxis textual que representa los patrones que se pueden hacer coincidir en las cadenas operadas.

Las expresiones regulares, introducidas en c ++ 11 , pueden opcionalmente admitir una matriz de retorno de cadenas coincidentes u otra sintaxis textual que defina cómo reemplazar patrones coincidentes en cadenas operadas.

Sintaxis

- regex_match // Devuelve si la secuencia de caracteres completa coincidió con la expresión regular, opcionalmente capturando en un objeto coincidente
- regex_search // Devuelve si una parte de la secuencia de caracteres coincidió con la expresión regular, opcionalmente capturando en un objeto coincidente
- regex_replace // Devuelve la secuencia de caracteres de entrada modificada por una expresión regular a través de una cadena de formato de reemplazo
- regex_token_iterator // Inicializado con una secuencia de caracteres definida por iteradores, una lista de índices de captura para iterar, y una expresión regular. La desreferenciación devuelve la coincidencia actualmente indexada de la expresión regular. El incremento se mueve al siguiente índice de captura o, si está actualmente en el último índice, restablece el índice y dificulta la próxima aparición de una coincidencia de expresiones regulares en la secuencia de caracteres
- regex_iterator // Inicializado con una secuencia de caracteres definida por iteradores y una expresión regular. La anulación de referencia devuelve la parte de la secuencia de caracteres con la que coinciden actualmente todas las expresiones regulares. Incrementando encuentra la siguiente aparición de una coincidencia de expresiones regulares en la secuencia de caracteres

Parámetros

Firma	Descripción
`bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)`	**`BidirectionalIterator`** es cualquier iterador personaje que proporciona a los operadores de incremento y decremento **`smatch`** pueden ser `cmatch` o cualquier otra otra variante de `match_results` que acepte el tipo de `BidirectionalIterator` la `smatch` argumento puede omitirse si no son necesarios los resultados de la expresión regular **Devuelve** si

Firma	Descripción
	`re` coincide con todo el carácter secuencia definida por `first` y `last`
`bool regex_match(const string& str, smatch& sm, const regex re&, regex_constraints::match_flag_type flags)`	**`string`** puede ser una `const char*` o una `string` valor L, *las funciones que aceptan una `string` valor R se eliminan de forma explícita.* **`smatch`** puede ser `cmatch` o cualquier otra variante de `match_results` que acepte el tipo de `str` `smatch` argumento `smatch` puede omitirse si los resultados de la expresión regular no son necesarios **Devuelve** si `re` coincidir con la secuencia de caracteres completa definida por `str`

Examples

Ejemplos básicos de regex_match y regex_search

```
const auto input = "Some people, when confronted with a problem, think \"I know, I'll use
regular expressions.\""s;
smatch sm;

cout << input << endl;

// If input ends in a quotation that contains a word that begins with "reg" and another word
begining with "ex" then capture the preceeding portion of input
if (regex_match(input, sm, regex("(.*)\".*\\breg.*\\bex.*\"\\s*$"))) {
    const auto capture = sm[1].str();

    cout << '\t' << capture << endl; // Outputs: "\tSome people, when confronted with a
problem, think\n"

    // Search our capture for "a problem" or "# problems"
    if(regex_search(capture, sm, regex("(a|d+)\\s+problems?"))) {
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);

        cout << '\t' << count << (count > 1 ? " problems\n" : " problem\n"); // Outputs: "\t1
problem\n"
        cout << "Now they have " << count + 1 << " problems.\n"; // Ouputs: "Now they have 2
problems\n"
    }
}
```

Ejemplo vivo

Ejemplo de regex_replace

Este código toma varios estilos de llaves y los convierte en One True Brace Style :

```
const auto input = "if (KnR)\n\tfoo();\nif (spaces) {\n    foo();\n}\nif
```

```
(allman)\n{\n\tfoo();\n}\nif (horstmann)\n{\tfoo();\n}\nif (pico)\n{\tfoo(); }\nif
(whitesmiths)\n\t{\n\tfoo();\n\t}\n"s;

cout << input << regex_replace(input, regex("(.+?)\\s*\\{?\\s*(.+?;)\\s*\\}?\\s*"), "$1
{\n\t$2\n}\n") << endl;
```

Ejemplo vivo

regex_token_iterator Ejemplo

Un `std::regex_token_iterator` proporciona una herramienta tremenda para extraer elementos de un archivo de valores separados por comas . Aparte de las ventajas de la iteración, este iterador también es capaz de capturar comas escapadas donde otros métodos luchan:

```
const auto input = "please split,this,csv, ,line,\\,\n"s;
const regex re{ "((?:[^\\\\,]|\\\\.)+)(?:,|$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, "\n"));
```

Ejemplo vivo

Un problema notable con los iteradores de expresiones regulares es que el argumento de `regex` debe ser un valor de l. Un valor R no funcionará .

Ejemplo de regex_iterator

Cuando el procesamiento de las capturas debe realizarse de forma iterativa, un `regex_iterator` es una buena opción. La desreferenciación de un `regex_iterator` devuelve un `match_result` . Esto es ideal para capturas condicionales o capturas que tienen interdependencia. Digamos que queremos tokenizar algún código C ++. Dado:

```
enum TOKENS {
    NUMBER,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    EQUALITY,
    OPEN_PARENTHESIS,
    CLOSE_PARENTHESIS
};
```

Podemos tokenizar esta cadena: `const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 -3"s` con un `regex_iterator` como este:

```
vector<TOKENS> tokens;
const regex re{ "\\s*(\\(?)\\s*(-?\\s*\\d+)\\s*(\\)?)\\s*(?:(\\+)|(-)|(\\*)|(/)|(=))" };
```

```
for_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto&
i) {
    if(i[1].length() > 0) {
        tokens.push_back(OPEN_PARENTHESIS);
    }

    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);

    if(i[3].length() > 0) {
        tokens.push_back(CLOSE_PARENTHESIS);
    }

    auto it = next(cbegin(i), 4);

    for(int result = ADDITION; it != cend(i); ++result, ++it) {
        if (it->length() > 0U) {
            tokens.push_back(static_cast<TOKENS>(result));
            break;
        }
    }
});

match_results<string::const_reverse_iterator> sm;

if(regex_search(crbegin(input), crend(input), sm, regex{ tokens.back() == SUBTRACTION ?
"^\\s*\\d+\\s*-\\s*(-?)" : "^\\s*\\d+\\s*(-?)" })) {
    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);
}
```

Ejemplo vivo

Un problema notable con los iteradores de expresiones regulares es que el argumento de `regex` debe ser un valor L, un valor R no funcionará: Visual Studio regex_iterator Bug?

Dividiendo una cuerda

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}
```

```
split("Some  string\t with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

Cuantificadores

Digamos que nos dan `const string input` como un número de teléfono para validar. Podríamos comenzar requiriendo una entrada numérica con **cero o más cuantificadores** : `regex_match(input, regex("\\d*"))` o **uno o más cuantificadores** : `regex_match(input, regex("\\d+"))` Pero ambos se quedan cortos si la `input` contiene una cadena numérica no válida como: "123" Usemos uno **o más cuantificadores** para asegurarnos de que obtengamos al menos 7 dígitos:

```
regex_match(input, regex("\\d{7,}"))
```

Esto garantizará que obtendremos al menos un número de teléfono de dígitos, pero la `input` también podría contener una cadena numérica que sea demasiado larga como: "123456789012". Así que vamos con un **cuantificador entre n y m** para que la `input` tenga al menos 7 dígitos pero no más de 11:

```
regex_match(input, regex("\\d{7,11}"));
```

Esto nos acerca, pero todavía se aceptan cadenas numéricas ilegales que están en el rango de [7, 11], como: "123456789" Así que hagamos que el código de país sea opcional con un **cuantificador perezoso** :

```
regex_match(input, regex("\\d?\\d{7,10}"))
```

Es importante tener en cuenta que el **cuantificador perezoso hace** coincidir la *menor cantidad de caracteres posible* , por lo que la única manera de que este carácter coincida es si ya hay 10 caracteres que se han comparado con `\d{7,10}` . (Para hacer coincidir el primer carácter con avidez, habríamos tenido que hacer: `\d{0,1}` .) El **cuantificador perezoso** se puede agregar a cualquier otro cuantificador.

Ahora, ¿cómo haríamos el código de área opcional *y* solo aceptaríamos un código de país si el código de área estaba presente?

```
regex_match(input, regex("(?:\\d{3,4})?\\d{7}"))
```

En esta expresión regular, el `\d{7}` *requiere* 7 dígitos. Estos 7 dígitos están precedidos opcionalmente por 3 o 4 dígitos.

Tenga en cuenta que no agregamos el **cuantificador perezoso** : ~~`\d{3,4}?\d{7}`~~ , el `\d{3,4}?` hubiera coincidido con 3 o 4 caracteres, prefiriendo 3. En su lugar, hacemos que el grupo que no captura coincida como máximo una vez, prefiriendo no coincidir. Causando una discrepancia si la `input` no incluyó el código de área como: "1234567".

En conclusión del tema del cuantificador, me gustaría mencionar el otro cuantificador adjunto que puede usar, el **cuantificador posesivo** . *El* **cuantificador perezoso** o el **cuantificador posesivo** se pueden agregar a cualquier cuantificador. La única función del **cuantificador posesivo** es ayudar al motor de expresiones regulares diciéndole, tome estos caracteres con avidez *y nunca los abandone, incluso si causa que la expresión regular falle* . Esto, por ejemplo, no tiene mucho sentido: `regex_match(input, regex("\\d{3,4}+\\d{7}))` porque una `input` como:" 1234567890 "no coincidiría con `\d{3,4}+` siempre coincidirá con 4 caracteres, incluso si la coincidencia con 3 hubiera permitido que la expresión regular sea exitosa.
El **cuantificador posesivo** se utiliza mejor *cuando el token cuantificado limita el número de caracteres comparables* . Por ejemplo:

```
regex_match(input, regex("(?:.*\\d{3,4}+){3}"))
```

Se puede usar para hacer coincidir si la `input` contenía alguno de los siguientes:

123 456 7890
123-456-7890
(123) 456-7890
(123) 456 - 7890

Pero cuando esta expresión regular realmente brilla es cuando la `input` contiene una entrada *ilegal* :

12345 - 67890

Sin el **cuantificador posesivo,** el motor de expresiones regulares debe regresar y probar *cada combinación de* `.*` *Y 3 o 4 caracteres* para ver si puede encontrar una combinación compatible. Con el **cuantificador posesiva** las expresiones regulares comienza donde el **cuantificador** 2^o **posesivo** fue apagado, el carácter '0', y el motor de expresiones regulares trata de ajustar el `.*` Para permitir `\d{3,4}` para que coincida; cuando no se puede, la expresión regular simplemente falla, no se realiza un seguimiento de retroceso para ver si anteriormente `.*` ajuste podría haber permitido una coincidencia.

Anclas

C ++ proporciona solo 4 anclas:

- `^` que afirma el inicio de la cadena
- `$` que afirma el final de la cadena
- `\b` que afirma un carácter `\W` o el principio o el final de la cadena
- `\B` que afirma un carácter `\w`

Digamos, por ejemplo, que queremos capturar un número *con* su signo:

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W)([+-]?\\d+)" })) {

    do {
        cout << sm[1] << endl;
        input = sm.suffix().str();
    } while(regex_search(input, sm, regex{ "(?:^\\W|\\b\\W)([+-]?\\d+)" }));
}
```

Ejemplo vivo

Una nota importante aquí es que el ancla no consume ningún carácter.

Lea Expresiones regulares en línea: https://riptutorial.com/es/cplusplus/topic/1681/expresiones-regulares

Capítulo 53: Fecha y hora usando encabezamiento

Examples

Tiempo de medición utilizando

El `system_clock` se puede usar para medir el tiempo transcurrido durante alguna parte de la ejecución de un programa.

c ++ 11

```
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is
std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

En este ejemplo, se usó `sleep_for` para hacer que el subproceso activo esté inactivo durante un período de tiempo medido en `std::chrono::seconds` , pero el código entre llaves puede ser cualquier llamada de función que tarde algún tiempo en ejecutarse.

Encuentra el número de días entre dos fechas

Este ejemplo muestra cómo encontrar el número de días entre dos fechas. La fecha se especifica por año / mes / día del mes y, además, hora / minuto / segundo.

El programa calcula el número de días en años desde 2000.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/***
 * Creates a std::tm structure from raw date.
 *
 * \param year (must be 1900 or greater)
 * \param month months since January - [1, 12]
 * \param day day of the month - [1, 31]
```

```
 * \param minutes minutes after the hour - [0, 59]
 * \param seconds seconds after the minute - [0, 61](until C++11) / [0, 60] (since C++11)
 *
 * Based on http://en.cppreference.com/w/cpp/chrono/c/tm
 */
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
    tm_ret.tm_mon = month - 1;
    tm_ret.tm_year = year - 1900;

    return tm_ret;
}

int get_days_in_year(int year) {

    using namespace std;
    using namespace std::chrono;

    // We want results to be in days
    typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

    // Create start time span
    std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
    auto tms = system_clock::from_time_t(std::mktime(&tm_start));

    // Create end time span
        std::tm tm_end =   CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
    auto tme = system_clock::from_time_t(std::mktime(&tm_end));

    // Calculate time duration between those two dates
    auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

    return diff_in_days.count();
}

int main()
{
    for ( int year = 2000; year <= 2016; ++year )
        std::cout << "There are " << get_days_in_year(year) << " days in " << year << "\n";
}
```

Lea Fecha y hora usando encabezamiento en línea:
https://riptutorial.com/es/cplusplus/topic/3936/fecha-y-hora-usando--chrono--encabezamiento

Capítulo 54: Función de C ++ "llamada por valor" vs. "llamada por referencia"

Introducción

El alcance de esta sección es explicar las diferencias en la teoría y la implementación de lo que sucede con los parámetros de una función al llamar.

En detalle, los parámetros se pueden ver como variables antes de la llamada a la función y dentro de la función, donde el comportamiento visible y la accesibilidad a estas variables difiere con el método utilizado para entregarlas.

Además, este tema también explica la reutilización de las variables y sus valores respectivos después de la llamada a la función.

Examples

Llamar por valor

Al llamar a una función hay nuevos elementos creados en la pila de programas. Estos incluyen alguna información sobre la función y también el espacio (ubicaciones de memoria) para los parámetros y el valor de retorno.

Cuando se entrega un parámetro a una función, el valor de la variable utilizada (o literal) se copia en la ubicación de memoria del parámetro de la función. Esto implica que ahora hay dos ubicaciones de memoria con el mismo valor. Dentro de la función solo trabajamos en la ubicación de memoria de parámetros.

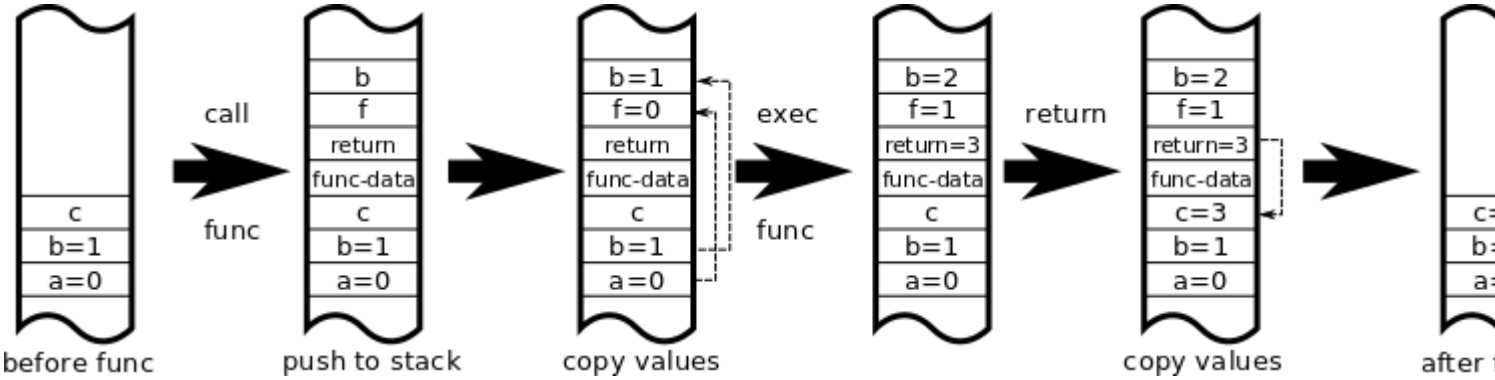
Después de abandonar la función, la memoria en la pila de programas se abre (elimina), lo que borra todos los datos de la llamada a la función, incluida la ubicación de la memoria de los parámetros que utilizamos en el interior. Por lo tanto, los valores cambiados dentro de la función no afectan los valores de las variables externas.

```
int func(int f, int b) {
  //new variables are created and values from the outside copied
  //f has a value of 0
  //inner_b has a value of 1
  f = 1;
  //f has a value of 1
  b = 2;
  //inner_b has a value of 2
  return f+b;
}

int main(void) {
  int a = 0;
  int b = 1; //outer_b
```

```
  int c;

  c = func(a,b);
  //the return value is copied to c

  //a has a value of 0
  //outer_b has a value of 1   <--- outer_b and inner_b are different variables
  //c has a value of 3
}
```

En este código creamos variables dentro de la función principal. Estos obtienen valores asignados. Al llamar a las funciones, se crean dos nuevas variables: `f` e `inner_b` donde `b` comparte el nombre con la variable externa y no comparte la ubicación de la memoria. El comportamiento de `a<->f` y `b<->b` es idéntico.

El siguiente gráfico simboliza lo que está sucediendo en la pila y por qué no hay cambios en varibale `b` . El gráfico no es completamente exacto pero enfatiza el ejemplo.



Se llama "llamada por valor" porque no entregamos las variables, sino solo los valores de estas variables.

Lea Función de C ++ "llamada por valor" vs. "llamada por referencia" en línea: https://riptutorial.com/es/cplusplus/topic/10669/funcion-de-c-plusplus--llamada-por-valor--vs---llamada-por-referencia-

Capítulo 55: Función de sobrecarga de plantillas

Observaciones

- Una función normal nunca está relacionada con una plantilla de función, a pesar del mismo nombre, del mismo tipo.
- Una llamada de función normal y una llamada de plantilla de función generada son diferentes incluso si comparten el mismo nombre, el mismo tipo de retorno y la misma lista de argumentos

Examples

¿Qué es una sobrecarga de plantilla de función válida?

Una plantilla de función se puede sobrecargar bajo las reglas para la sobrecarga de funciones que no son de plantilla (mismo nombre, pero diferentes tipos de parámetros) y además, la sobrecarga es válida si

- El tipo de retorno es diferente, o
- La lista de parámetros de la plantilla es diferente, excepto por la denominación de los parámetros y la presencia de argumentos predeterminados (no forman parte de la firma)

Para una función normal, la comparación de dos tipos de parámetros es fácil para el compilador, ya que tiene toda la información. Pero un tipo dentro de una plantilla puede no estar determinado todavía. Por lo tanto, la regla para cuando dos tipos de parámetros son iguales es aproximativa aquí y dice que los tipos y valores no dependientes deben coincidir y que la ortografía de los tipos y expresiones dependientes debe ser la misma (más precisamente, deben ajustarse a la denominadas reglas de ODR), excepto que los parámetros de la plantilla pueden ser renombrados. Sin embargo, si bajo tales ortografías diferentes, dos valores dentro de los tipos se consideran diferentes, pero siempre se crea una instancia de los mismos valores, la sobrecarga no es válida, pero no se requiere ningún diagnóstico del compilador.

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

Esta es una sobrecarga válida, ya que "T" y "T *" son ortografías diferentes. Pero lo siguiente no es válido, sin diagnóstico requerido

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }
```

```
template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

Lea Función de sobrecarga de plantillas en línea:
https://riptutorial.com/es/cplusplus/topic/4164/funcion-de-sobrecarga-de-plantillas

Capítulo 56: Funciones de miembro de clase constante

Observaciones

Lo que realmente significa "funciones miembro const" de una clase significa. La definición simple parece ser que una función miembro const no puede cambiar el objeto. Pero lo que significa 'no puede cambiar' realmente significa aquí. Simplemente significa que no puede hacer una asignación para miembros de datos de clase.

Sin embargo, puede realizar otras operaciones indirectas como insertar una entrada en un mapa como se muestra en el ejemplo. Permitir que esto parezca que esta función const es modificar el objeto (sí, lo hace en un sentido), pero está permitido.

Entonces, el significado real es que una función miembro const no puede hacer una asignación para las variables de datos de clase. Pero puede hacer otras cosas como las explicadas en el ejemplo.

Examples

función miembro constante

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;             // This works? Yes it does.
        delete mapOfStrings;                      // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
```

```
};

int main(int argc, char* argv[]) {

    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

Lea Funciones de miembro de clase constante en línea:
https://riptutorial.com/es/cplusplus/topic/7120/funciones-de-miembro-de-clase-constante

Capítulo 57: Funciones de miembro virtual

Sintaxis

- vacío virtual f ();
- vacío virtual g () = 0;
- // C ++ 11 o posterior:
    - anulación virtual h (h);
    - anulación i () anulación;
    - vacío virtual j () final;
    - void k () final;

Observaciones

- Solo las funciones miembro no estáticas, sin plantilla pueden ser `virtual` .
- Si está utilizando C ++ 11 o posterior, se recomienda usar la función de `override` al anular una función de miembro virtual de una clase base.
- Las clases base polimórficas a menudo tienen destructores virtuales para permitir que un objeto derivado se elimine a través de un puntero a la clase base . Si el destructor no fuera virtual, tal operación conduce a un comportamiento indefinido [expr.delete] §5.3.5 / 3 .

Examples

Usando override con virtual en C ++ 11 y versiones posteriores

La `override` especificador tiene un significado especial en C ++ 11 en adelante, si se adjunta al final de la firma de la función. Esto significa que una función es

- Anulando la función presente en la clase base y
- La función de clase base es `virtual`

No hay `run time` significado de `run time` de este especificador, ya que está pensado principalmente como una indicación para compiladores

El siguiente ejemplo demostrará el cambio en el comportamiento con nuestro uso sin anular.

Sin `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};
```

```
struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; }
};
```

Con `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; }
};
```

Tenga en cuenta que la `override` no es una palabra clave, sino un identificador especial que solo puede aparecer en las firmas de funciones. En todos los demás contextos, la `override` todavía se puede usar como un identificador:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2;  // Compilation error: keywords can't be used as identifiers.
}
```

Funciones de miembro virtual vs no virtual

Con funciones de miembro virtual:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
}
```

Sin funciones de miembro virtual:

```
#include <iostream>

struct X {
   void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
   void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

Funciones virtuales finales

C ++ 11 introdujo el especificador `final` que prohíbe la invalidación de métodos si aparece en la firma del método:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }
};

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

El especificador `final` solo se puede utilizar con la función miembro "virtual" y no se puede aplicar a funciones miembro no virtuales

Como `final` , también hay un especificador de llamada 'reemplazo' que evita el reemplazo de funciones `virtual` en la clase derivada.

Los especificadores `override` y `final` pueden combinarse para tener el efecto deseado:

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo\n";
    }
};
```

Comportamiento de funciones virtuales en constructores y destructores.

El comportamiento de las funciones virtuales en constructores y destructores a menudo es confuso cuando se encuentran por primera vez.

```
#include <iostream>
using namespace std;

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "When called from " << caller << ", "  << v() << " gets called.\n";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }

};

int main() {
     derived d;
}
```

Salida:

Cuando se llama desde el constructor base, se llama base :: v ().
Cuando se llama desde un constructor derivado, se deriva :: v ().
Cuando se llama desde un destructor derivado, se deriva :: v ().
Cuando se llama desde el destructor base, se llama base :: v ().

El razonamiento detrás de esto es que la clase derivada puede definir miembros adicionales que aún no están inicializados (en el caso del constructor) o que ya están destruidos (en el caso del destructor), y llamar a sus funciones miembro no sería seguro. Por lo tanto, durante la construcción y destrucción de objetos C ++, el tipo *dinámico* de `*this` se considera que es la clase del constructor o destructor y no una clase más derivada.

Ejemplo:

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
    base()
    {
        std::cout << "foo is " << foo() << std::endl;
    }
    virtual int foo() { return 42; }
};

class derived : public base {
    unique_ptr<int> ptr_;
public:
    derived(int i) : ptr_(new int(i*i)) { }
    // The following cannot be called before derived::derived due to how C++ behaves,
    // if it was possible... Kaboom!
    int foo() override   { return *ptr_; }
};

int main() {
    derived d(4);
}
```

Funciones virtuales puras

También podemos especificar que una función `virtual` es *simplemente virtual* (abstracta), agregando `= 0` a la declaración. Las clases con una o más funciones virtuales puras se consideran abstractas y no se pueden crear instancias; solo las clases derivadas que definen o heredan definiciones para todas las funciones virtuales puras pueden ser instanciadas.

```
struct Abstract {
    virtual void f() = 0;
};

struct Concrete {
    void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.
```

Incluso si una función se especifica como virtual pura, se le puede dar una implementación predeterminada. A pesar de esto, la función todavía se considerará abstracta, y las clases derivadas tendrán que definirla antes de que puedan ser instanciadas. En este caso, la versión de la clase derivada de la función puede incluso llamar a la versión de la clase base.

```
struct DefaultAbstract {
    virtual void f() = 0;
};
void DefaultAbstract::f() {}
```

```
struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

Hay un par de razones por las que podríamos querer hacer esto:

- Si queremos crear una clase que no pueda ser instanciada, pero que no evite que sus clases derivadas sean instanciadas, podemos declarar el destructor como simplemente virtual. Al ser el destructor, debe definirse de todos modos, si queremos poder desasignar la instancia. Y como es muy probable que el destructor ya sea virtual para evitar fugas de memoria durante el uso polimórfico , no incurriremos en un impacto de rendimiento innecesario al declarar otra función `virtual` . Esto puede ser útil al hacer interfaces.

  ```
  struct Interface {
      virtual ~Interface() = 0;
  };
  Interface::~Interface() = default;

  struct Implementation : Interface {};
  // ~Implementation() is automatically defined by the compiler if not explicitly
  //  specified, meeting the "must be defined before instantiation" requirement.
  ```

- Si la mayoría o todas las implementaciones de la función virtual pura contendrán código duplicado, ese código se puede mover a la versión de clase base, haciendo que el código sea más fácil de mantener.

  ```
  class SharedBase {
      State my_state;
      std::unique_ptr<Helper> my_helper;
      // ...

    public:
      virtual void config(const Context& cont) = 0;
      // ...
  };
  /* virtual */ void SharedBase::config(const Context& cont) {
      my_helper = new Helper(my_state, cont.relevant_field);
      do_this();
      and_that();
  }

  class OneImplementation : public SharedBase {
      int i;
      // ...

    public:
      void config(const Context& cont) override;
      // ...
  };
  void OneImplementation::config(const Context& cont) /* override */ {
      my_state = { cont.some_field, cont.another_field, i };
      SharedBase::config(cont);
      my_unique_setup();
  };
  ```

```
// And so on, for other classes derived from SharedBase.
```

Lea Funciones de miembro virtual en línea:
https://riptutorial.com/es/cplusplus/topic/1752/funciones-de-miembro-virtual

Capítulo 58: Funciones en linea

Introducción

Una función definida con el especificador en `inline` es una función en línea. Una función en línea se puede definir de forma múltiple sin violar la Regla de una definición y, por lo tanto, se puede definir en un encabezado con enlace externo. Declarar una función en línea sugiere al compilador que la función debe estar en línea durante la generación del código, pero no proporciona una garantía.

Sintaxis

- **`inline`** *function_declaration*
- **`inline`** *function_definition*
- *class {function_definition};*

Observaciones

Generalmente, si el código generado para una función es lo *suficientemente* pequeño, entonces es un buen candidato para estar en línea. ¿Porque? Si una función es grande y está en línea en un bucle, para todas las llamadas realizadas, el código de la función grande se duplicará, lo que provocará la inflexión del tamaño binario generado. Pero, ¿qué tan pequeño es suficiente?

Si bien las funciones en línea parecen ser una excelente manera de evitar la sobrecarga de las llamadas de funciones, se debe tener en cuenta que no todas las funciones que están marcadas en `inline` están en línea. En otras palabras, cuando se dice en `inline` , es solo una sugerencia para el compilador, no un pedido: el compilador no está obligado a incluir la función en línea, es libre de ignorarla, la mayoría lo hace. Los compiladores modernos son mejores para hacer tales optimizaciones que esta palabra clave es ahora un vestigio del pasado, cuando esta sugerencia de función incorporada por el programador fue tomada en serio por los compiladores. Incluso las funciones no marcadas en `inline` están integradas por el compilador cuando ve beneficio al hacerlo.

Inline como directiva de vinculación

El uso más práctico de `inline` en C ++ moderno viene de usarlo como una directiva de vinculación. Cuando se *define* , no se declara, una función en un encabezado que se incluirá en varias fuentes, cada unidad de traducción tendrá su propia copia de esta función que dará lugar a una violación de la ODR (Regla de una definición); esta regla aproximadamente dice que solo puede haber una definición de una función, variable, etc. Para evitar esta violación, marcar la definición de la función en `inline` hace que la vinculación de la función sea interna.

Preguntas frecuentes

¿Cuándo debo escribir la palabra clave 'en línea' para una función / método en C ++?

Solo cuando desea que la función se defina en un encabezado. Más exactamente solo cuando la definición de la función puede aparecer en varias unidades de compilación. Es una buena idea definir funciones pequeñas (como en un trazador de líneas) en el archivo de encabezado, ya que le da al compilador más información para trabajar mientras optimiza su código. También aumenta el tiempo de compilación.

¿Cuándo no debo escribir la palabra clave 'en línea' para una función / método en C ++?

No agregue en `inline` cuando crea que su código se ejecutará más rápido si el compilador lo alinea.

¿Cuándo no sabrá el compilador cuándo hacer una función / método en línea?

Generalmente, el compilador podrá hacer esto mejor que tú. Sin embargo, el compilador no tiene la opción de código en línea si no tiene la definición de la función. En el código optimizado al máximo, generalmente todos los métodos privados están en línea, ya sea que lo solicite o no.

Ver también

- ¿Cuándo debo escribir la palabra clave 'en línea' para una función / método?
- ¿Todavía hay un uso para en línea?

Examples

Declaración de función en línea no miembro

```
inline int add(int x, int y);
```

Definición de función en línea no miembro

```
inline int add(int x, int y)
{
    return x + y;
}
```

Funciones en línea miembro

```
// header (.hpp)
struct A
{
```

```
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

¿Qué es la función en línea?

```
inline int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

En el código anterior, cuando la `add` está en línea, el código resultante se convertirá en algo como esto

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

La función en línea no se ve por ninguna parte, su cuerpo se *inserta* en el cuerpo de la persona que llama. Si el `add` no estuviera en línea, se llamaría una función. Se debe incurrir en la sobrecarga de llamar a una función, como crear un nuevo marco de pila , copiar argumentos, hacer variables locales, saltar (perder la localidad de referencia y no por falta de caché), etc.

Lea Funciones en linea en línea: https://riptutorial.com/es/cplusplus/topic/7150/funciones-en-linea

Capítulo 59: Funciones especiales para miembros

Examples

Destructores virtuales y protegidos.

Una clase diseñada para ser heredada se llama clase base. Se debe tener cuidado con las funciones especiales para miembros de dicha clase.

Una clase diseñada para ser utilizada polimórficamente en tiempo de ejecución (a través de un puntero a la clase base) debe declarar el destructor `virtual` . Esto permite que las partes derivadas del objeto se destruyan adecuadamente, incluso cuando el objeto se destruye a través de un puntero a la clase base.

```
class Base {
public:
    virtual ~Base() = default;

private:
    //    data members etc.
};

class Derived : public Base { //  models Is-A relationship
public:
    //    some methods

private:
    //    more data members
};

//    virtual destructor in Base ensures that derived destructors
//    are also called when the object is destroyed
std::unique_ptr<Base> base = std::make_unique<Derived>();
base = nullptr;  //    safe, doesn't leak Derived's members
```

Si la clase no necesita ser polimórfica, pero aún necesita permitir que su interfaz sea heredada, use un destructor `protected` no virtual.

```
class NonPolymorphicBase {
public:
    //    some methods

protected:
    ~NonPolymorphicBase() = default; //    note: non-virtual

private:
    //    etc.
};
```

Tal clase nunca puede ser destruida a través de un puntero, evitando fugas silenciosas debido al

corte en rodajas.

Esta técnica se aplica especialmente a las clases diseñadas para ser clases base `private` . Una clase de este tipo podría usarse para encapsular algunos detalles de implementación comunes, al tiempo que proporciona métodos `virtual` como puntos de personalización. Este tipo de clase nunca debe utilizarse polimórficamente, y un destructor `protected` ayuda a documentar este requisito directamente en el código.

Finalmente, algunas clases pueden requerir que *nunca se* usen como clase base. En este caso, la clase se puede marcar como `final` . Un destructor público no virtual normal está bien en este caso.

```
class FinalClass final {  //    marked final here
public:
    ~FinalClass() = default;

private:
    //    etc.
};
```

Movimiento implícito y copia

Tenga en cuenta que declarar un destructor impide que el compilador genere constructores de movimiento implícito y operadores de asignación de movimiento. Si declara un destructor, recuerde agregar también las definiciones apropiadas para las operaciones de movimiento.

Además, la declaración de operaciones de movimiento suprimirá la generación de operaciones de copia, por lo que también deben agregarse (si se requiere que los objetos de esta clase tengan copia semántica).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    //    compiler won't generate these unless we tell it to
    //    because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    //    declaring move operations will suppress generation
    //    of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Copiar e intercambiar

Si está escribiendo una clase que administra recursos, debe implementar todas las funciones especiales para miembros (consulte la Regla de tres / cinco / cero). El enfoque más directo para escribir el constructor de copia y el operador de asignación sería:

```
person(const person &other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
        name = new char[std::strlen(other.name) + 1];
        std::strcpy(name, other.name);
        age = other.age;
    }

    return *this;
}
```

Pero este enfoque tiene algunos problemas. Falla la fuerte garantía excepción - si es `new[]` tiros, que ya hemos aclarado los recursos propiedad de `this` y no se puede recuperar. Estamos duplicando gran parte de la lógica de la construcción de copias en la asignación de copias. Y tenemos que recordar la verificación de autoasignación, que generalmente solo agrega gastos generales a la operación de copia, pero aún es crítica.

Para satisfacer la fuerte garantía de excepción y evitar la duplicación de código (doble con el operador de asignación de movimiento subsiguiente), podemos utilizar el lenguaje de copia e intercambio:

```
class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};
```

¿Por qué funciona esto? Considera lo que pasa cuando tenemos

```
person p1 = ...;
person p2 = ...;
p1 = p2;
```

Primero, copiamos y construimos `rhs` desde `p2` (que no tuvimos que duplicar aquí). Si esa operación se produce, no hacemos nada en el `operator=` y `p1` permanece intacto. A continuación,

intercambiamos los miembros entre `*this` y `rhs` , y luego `rhs` queda fuera del alcance. Cuando `operator=` , eso limpia de forma implícita los recursos originales de `this` (a través del destructor, que no tuvimos que duplicar). La autoasignación también funciona: es menos eficiente con la copia e intercambio (implica una asignación adicional y una desasignación), pero si ese es el escenario improbable, no ralentizamos el caso de uso típico para tenerlo en cuenta.

C ++ 11

La formulación anterior funciona tal como está para la asignación de movimiento.

```
p1 = std::move(p2);
```

Aquí, movemos-construimos `rhs` desde `p2` , y todo lo demás es igual de válido. Si una clase es movible pero no puede copiarse, no es necesario eliminar la asignación de copia, ya que este operador de asignación simplemente tendrá una forma incorrecta debido al constructor de copia eliminado.

Constructor predeterminado

Un *constructor predeterminado* es un tipo de constructor que no requiere parámetros cuando se le llama. Se nombra después del tipo que construye y es una función miembro de él (como lo son todos los constructores).

```
class C{
    int i;
public:
    // the default constructor definition
    C()
    : i(0){ // member initializer list -- initialize i to 0
        // constructor function body -- can do more complex things here
    }
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Otra forma de satisfacer el requisito de "sin parámetros" es que el desarrollador proporcione valores predeterminados para todos los parámetros:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
    D( int i = 0, int j = 42 )
    : i(i), j(j){
    }
};
```

```
D d; // calls constructor of D with the provided default values for the parameters
```

En algunas circunstancias (es decir, el desarrollador no proporciona constructores y no hay otras condiciones de descalificación), el compilador proporciona implícitamente un constructor predeterminado vacío:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Tener algún otro tipo de constructor es una de las condiciones de descalificación mencionadas anteriormente:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};

C c1; // Compile ERROR: C has no (implicitly defined) default constructor
```

c ++ 11

Para evitar la creación implícita del constructor predeterminado, una técnica común es declararla como `private` (sin definición). La intención es provocar un error de compilación cuando alguien intenta usar el constructor (esto puede resultar en un error de *Acceso a privado* o un error de vinculador, según el compilador).

Para asegurarse de que se defina un constructor predeterminado (funcionalmente similar al implícito), un desarrollador podría escribir uno vacío explícitamente.

c ++ 11

En C ++ 11, un desarrollador también puede usar la palabra clave `delete` para evitar que el compilador proporcione un constructor predeterminado.

```
class C{
    int i;
public:
    // default constructor is explicitly deleted
    C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted
```

Además, un desarrollador también puede ser explícito acerca de querer que el compilador proporcione un constructor predeterminado.

```
class C{
    int i;
public:
    // does have automatically generated default constructor (same as implicit one)
    C() = default;

    C( int i ) : i(i){}
};

C c1; // default constructed
C c2( 1 ); // constructed with the int taking constructor
```

c ++ 14

Puede determinar si un tipo tiene un constructor predeterminado (o es un tipo primitivo) usando `std::is_default_constructible` de `<type_traits>` :

```
class C1{ };
class C2{ public: C2(){} };
class C3{ public: C3(int){} };

using std::cout; using std::boolalpha; using std::endl;
using std::is_default_constructible;
cout << boolalpha << is_default_constructible<int>() << endl; // prints true
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

c ++ 11

En C ++ 11, todavía es posible usar la versión no funcional de `std::is_default_constructible` :

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

Incinerador de basuras

Un *destructor* es una función sin argumentos que se llama cuando un objeto definido por el usuario está a punto de ser destruido. Se nombra después del tipo que destruye con un prefijo `~` .

```
class C{
    int* is;
    string s;
public:
    C()
    : is( new int[10] ){
    }

    ~C(){  // destructor definition
        delete[] is;
    }
};

class C_child : public C{
    string s_ch;
public:
    C_child(){}
```

```
    ~C_child(){} // child destructor
};

void f(){
    C c1; // calls default constructor
    C c2[2]; // calls default constructor for both elements
    C* c3 = new C[2]; // calls default constructor for both array elements

    C_child c_ch;  // when destructed calls destructor of s_ch and of C base (and in turn s)

    delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch
```

En la mayoría de las circunstancias (es decir, un usuario no proporciona un destructor y no hay otras condiciones de descalificación), el compilador proporciona un destructor predeterminado de manera implícita:

```
class C{
    int i;
    string s;
};

void f(){
    C* c1 = new C;
    delete c1; // C has a destructor
}
```

```
class C{
    int m;
private:
    ~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f(){
    C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}
```

c ++ 11

En C ++ 11, un desarrollador puede anular este comportamiento impidiendo que el compilador proporcione un destructor predeterminado.

```
class C{
    int m;
public:
    ~C() = delete; // does NOT have implicit destructor
};

void f{
    C c1;
```

```
} // Compile ERROR: C has no destructor
```

Además, un desarrollador también puede ser explícito acerca de querer que el compilador proporcione un destructor predeterminado.

```
class C{
    int m;
public:
    ~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
    C c1;
} // C has a destructor -- c1 properly destroyed
```

c ++ 11

Puede determinar si un tipo tiene un destructor (o es un tipo primitivo) usando `std::is_destructible` de `<type_traits>` :

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Lea Funciones especiales para miembros en línea:
https://riptutorial.com/es/cplusplus/topic/1476/funciones-especiales-para-miembros

Capítulo 60: Funciones miembro no estáticas

Sintaxis

- // Llamando:
    - variable.member_function ();
    - variable_pointer-> miembro_función ();
- // Definición:
    - ret_type class_name :: member_function () cv-qualifiers {
        - cuerpo;
    - }
- // Prototipo:
    - clase nombre_clase {
        - virt-specifier ret_type member_function () cv-qualifiers virt-specifier-seq;
        - // virt-specifier: "virtual", si corresponde.
        - // calificadores cv: "const" y / o "volátil", si corresponde.
        - // virt-specifier-seq: "anular" y / o "final", si corresponde.
    - }

Observaciones

Una función miembro no `static` es una función miembro `class` / `struct` / `union` , que se llama en una instancia particular y opera en dicha instancia. A diferencia de las funciones miembro `static` , no se puede llamar sin especificar una instancia.

Para obtener información sobre clases, estructuras y uniones, consulte el tema principal .

Examples

Funciones miembro no estáticas

Una `class` o `struct` puede tener funciones miembro así como variables miembro. Estas funciones tienen una sintaxis mayoritariamente similar a las funciones independientes, y pueden definirse dentro o fuera de la definición de clase; Si se define fuera de la definición de la clase, el nombre de la función tiene como prefijo el nombre de la clase y el operador de alcance (`::` :).

```
class CL {
  public:
    void  definedInside() {}
    void definedOutside();
};
```

```
void CL::definedOutside() {}
```

Estas funciones se invocan en una instancia (o referencia a una instancia) de la clase con el operador de punto (`.`), O un puntero a una instancia con el operador de flecha (`->`), y cada llamada está vinculada a la instancia de la función fue llamado en cuando se llama a una función miembro en una instancia, tiene acceso a todos los campos de esa instancia (a través de `this` puntero), pero solo puede acceder a los campos de otras instancias si esas instancias se suministran como parámetros.

```
struct ST {
    ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) { }

    int get_i() const { return i; }
    bool compare_i(const ST& other) const { return (i == other.i); }

  private:
    std::string s;
    int i;
};
ST st1;
ST st2("Species", 8472);

int  i = st1.get_i(); // Can access st1.i, but not st2.i.
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

Estas funciones pueden acceder a las variables miembro y / u otras funciones miembro, independientemente de la variable o los modificadores de acceso de la función. También se pueden escribir fuera de orden, accediendo a las variables de miembro y / o llamando a las funciones de miembro declaradas antes de ellas, ya que se debe analizar toda la definición de la clase antes de que el compilador pueda comenzar a compilar una clase.

```
class Access {
  public:
    Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}

    int i;
    int get_k() const { return k; }
    bool private_no_more() const { return i_be_private(); }
  protected:
    int j;
    int get_i() const { return i; }
  private:
    int k;
    int get_j() const { return j; }
    bool i_be_private() const { return ((i > j) && (k < j)); }
};
```

Encapsulacion

Un uso común de las funciones miembro es para la encapsulación, usando un elemento de *acceso* (comúnmente conocido como captador) y un *mutador* (comúnmente conocido como configurador) en lugar de acceder a los campos directamente.

```
class Encapsulator {
    int encapsulated;

  public:
    int  get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e)  { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

Dentro de la clase, se puede acceder libremente a `encapsulated` mediante cualquier función miembro no estática; fuera de la clase, el acceso a él está regulado por las funciones miembro, usando `get_encapsulated()` para leerlo y `set_encapsulated()` para modificarlo. Esto evita modificaciones no intencionales a la variable, ya que se usan funciones separadas para leerla y escribirla. [Hay muchas discusiones sobre si los captadores y los definidores proporcionan o rompen la encapsulación, con buenos argumentos para ambas afirmaciones; este acalorado debate está fuera del alcance de este ejemplo.]

Nombre ocultar e importar

Cuando una clase base proporciona un conjunto de funciones sobrecargadas, y una clase derivada agrega otra sobrecarga al conjunto, esto oculta todas las sobrecargas proporcionadas por la clase base.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1);    // Output:  int
hb.f(true); // Output:  bool
hb.f(s);    // Output:  std::string;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  float
hd.f(true); // Output:  float
hd.f(s);    // Error: Can't convert from std::string to float.
```

Esto se debe a las reglas de resolución de nombres: durante la búsqueda de nombres, una vez que se encuentra el nombre correcto, dejamos de buscar, incluso si claramente no hemos encontrado la *versión* correcta de la entidad con ese nombre (como en `hd.f(s)`); debido a esto, la

sobrecarga de la función en la clase derivada evita que la búsqueda de nombres descubra las sobrecargas en la clase base. Para evitar esto, se puede usar una declaración de uso para "importar" nombres de la clase base a la clase derivada, de modo que estén disponibles durante la búsqueda de nombres.

```
struct HidingDerived : HiddenBase {
     // All members named HiddenBase::f shall be considered members of HidingDerived for
lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  int
hd.f(true); // Output:  bool
hd.f(s);    // Output:  std::string
```

Si una clase derivada importa nombres con una declaración de uso, pero también declara funciones con la misma firma que las funciones en la clase base, las funciones de la clase base serán anuladas u ocultadas silenciosamente.

```
struct NamesHidden {
    virtual void hide_me()      {}
    virtual void hide_me(float) {}
    void hide_me(int)           {}
    void hide_me(bool)          {}
};

struct NameHider : NamesHidden {
    using NamesHidden::hide_me;

    void hide_me()    {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

También se puede utilizar una declaración de uso para cambiar los modificadores de acceso, siempre que la entidad importada fuera `public` o esté `protected` en la clase base.

```
struct ProMem {
  protected:
    void func() {}
};

struct BecomesPub : ProMem {
    using ProMem::func;
};

// ...

ProMem pm;
BecomesPub bp;
```

```
pm.func(); // Error: protected.
bp.func(); // Good.
```

De manera similar, si queremos llamar explícitamente a una función miembro de una clase específica en la jerarquía de herencia, podemos calificar el nombre de la función al llamar a la función, especificando esa clase por nombre.

```
struct One {
    virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
        One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
        Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // Normal syntax.
t.Two::f(); // Calls version of f() defined in Two.
t.One::f(); // Calls version of f() defined in One.
```

Funciones de miembro virtual

Las funciones miembro también pueden ser declaradas `virtual` . En este caso, si se llama a un puntero o referencia a una instancia, no se accederá directamente a ellos; más bien, buscarán la función en la tabla de funciones virtuales (una lista de funciones de punteros a miembros para funciones virtuales, más comúnmente conocida como `vtable` o `vftable`), y la usarán para llamar a la versión apropiada para la dinámica de la instancia Tipo (real). Si la función se llama directamente, desde una variable de una clase, no se realiza ninguna búsqueda.

```
struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }

// ...
```

```
Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d;   // References.

b.func();   // Output:  In Base.
d.func();   // Output:  In Derived.

pb->func(); // Output:  In Base.
pd->func(); // Output:  In Derived.

rb.func();  // Output:  In Base.
rd.func();  // Output:  In Derived.

slicer(b);  // Output:  In Base.
slicer(d);  // Output:  In Base.
```

Tenga en cuenta que si bien `pd` es `Base*` , y `rd` es una `Base&` , llama `func()` en cualquiera de las dos llamadas `Derived::func()` lugar de `Base::func()` ; esto se debe a que `vtable` for `Derived` actualiza la entrada `Base::func()` para que, en cambio, apunte a `Derived::func()` . A la inversa, observe cómo pasar una instancia a `slicer()` siempre da como resultado que se llame a `Base::func()` , incluso cuando la instancia pasada es un `Derived` ; esto se debe a algo conocido como *división de datos* , donde pasar una instancia `Derived` a un parámetro `Base` por valor hace que la parte de la instancia `Derived` que no es inaccesible a una instancia `Base` .

Cuando una función de miembro se define como virtual, todas las funciones de miembro de clase derivadas con la misma firma lo anulan, independientemente de si la función de anulación se especifica como `virtual` o no. Sin embargo, esto puede hacer que las clases derivadas sean más difíciles de analizar para los programadores, ya que no hay ninguna indicación de qué función (es) es / son `virtual` .

```
struct B {
    virtual void f() {}
};

struct D : B {
    void f() {} // Implicitly virtual, overrides B::f.
                //  You'd have to check B to know that, though.
};
```

Sin embargo, tenga en cuenta que una función derivada solo reemplaza una función base si sus firmas coinciden; incluso si una función derivada se declara explícitamente `virtual` , en su lugar creará una nueva función virtual si las firmas no coinciden.

```
struct BadB {
    virtual void f() {}
};

struct BadD : BadB {
    virtual void f(int i) {} // Does NOT override BadB::f.
};
```

C ++ 11

A partir de C ++ 11, la intención de anular se puede hacer explícita con la `override` palabra clave sensible al contexto. Esto le dice al compilador que el programador espera que anule una función de clase base, lo que hace que el compilador omita un error si *no* anula nada.

```
struct CPP11B {
    virtual void f() {}
};

struct CPP11D : CPP11B {
    void f() override {}
    void f(int i) override {} // Error: Doesn't actually override anything.
};
```

Esto también tiene la ventaja de decirle a los programadores que la función es virtual y también declarada en al menos una clase base, lo que puede hacer que las clases complejas sean más fáciles de analizar.

Cuando una función se declara `virtual` y se define fuera de la definición de la clase, el especificador `virtual` debe incluirse en la declaración de la función y no repetirse en la definición.

C ++ 11

Esto también es válido para `override` .

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

Si una clase base sobrecarga una función `virtual` , solo las sobrecargas que se especifiquen explícitamente como `virtual` serán virtuales.

```
struct BOverload {
    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).
```

Para más información, vea el tema relevante .

Const Correccion

Uno de los usos principales de `this` calificador de `const` es la *corrección* `const` . Esta es la práctica de garantizar que solo los accesos que *necesitan* modificar un objeto *puedan* modificar el objeto, y que cualquier función (miembro o no miembro) que no necesite modificar un objeto no tiene acceso de escritura para eso. objeto (ya sea directa o indirectamente). Esto evita modificaciones no intencionales, haciendo que el código sea menos propenso a errores. También permite que cualquier función que no necesite modificar el estado pueda tomar un objeto `const` o no `const` , sin necesidad de reescribir o sobrecargar la función.

`const` corrección de `const` , debido a su naturaleza, comienza de abajo hacia arriba: cualquier función miembro de la clase que no necesite cambiar de estado se declara como `const` , de modo que se puede llamar en instancias de `const` . Esto, a su vez, permite pasar por referencia parámetros para ser declarado `const` cuando no necesitan ser modificados, lo que permite que las funciones que llevan ya sea `const` o no `const` objetos sin quejarse, y `const` -ness puede propagarse hacia el exterior en este manera. Debido a esto, los captadores suelen estar `const` , al igual que cualquier otra función que no necesite modificar el estado lógico.

```
class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(const Field& f) : fld(f) {}     // Modifies.

    const Field& get_field()       { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; }    // Modifies.

    void do_something(int i) {                     // Modifies.
        fld.insert_value(i);
    }
    void do_nothing()        { }                   // Doesn't modify; should be const.
};

class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f) : fld(f) {}       // Not const: Modifies.

    const Field& get_field() const { return fld; } // const: Doesn't modify.
    void set_field(const Field& f) { fld = f; }    // Not const: Modifies.

    void do_something(int i) {                     // Not const: Modifies.
        fld.insert_value(i);
    }
    void do_nothing() const  { }                   // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
  // Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
  // Error: Same as above.
// Oops.
```

```
const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
but_i_can.do_nothing();          // Good.
```

Como lo ilustran los comentarios en `ConstIncorrect` y `ConstCorrect` , las funciones que califican para cv también sirven como documentación. Si una clase es `const` correcta, cualquier función que no está `const` puede suponerse con seguridad para cambiar de estado, y cualquier función que es `const` puede suponer con seguridad que no cambie de estado.

Lea Funciones miembro no estáticas en línea:
https://riptutorial.com/es/cplusplus/topic/5661/funciones-miembro-no-estaticas

Capítulo 61: Futuros y Promesas

Introducción

Las promesas y los futuros se utilizan para transportar un solo objeto de un hilo a otro.

El objeto que genera el resultado establece un objeto `std::promise` .

Se puede usar un objeto `std::future` para recuperar un valor, para probar si un valor está disponible, o para detener la ejecución hasta que el valor esté disponible.

Examples

std :: futuro y std :: promesa

El siguiente ejemplo establece la promesa de ser consumido por otro hilo:

```
    {
        auto promise = std::promise<std::string>();

        auto producer = std::thread([&]
        {
            promise.set_value("Hello World");
        });

        auto future = promise.get_future();

        auto consumer = std::thread([&]
        {
            std::cout << future.get();
        });

        producer.join();
        consumer.join();
}
```

Ejemplo de asíncrono diferido

Este código implementa una versión de `std::async` , pero se comporta como si `async` se llamara siempre con la política de lanzamiento `deferred` . Esta función tampoco tiene un comportamiento `future` especial de `async` ; El `future` devuelto puede ser destruido sin adquirir nunca su valor.

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    using result_type = decltype(func());

    auto promise = std::promise<result_type>();
    auto future  = promise.get_future();

    std::thread(std::bind([=](std::promise<result_type>& promise)
```

```
    {
        try
        {
            promise.set_value(func());
            // Note: Will not work with std::promise<void>. Needs some meta-template
programming which is out of scope for this example.
        }
        catch(...)
        {
            promise.set_exception(std::current_exception());
        }
    }, std::move(promise))).detach();

    return future;
}
```

std :: packaged_task y std :: futuro

`std::packaged_task` agrupa una función y la promesa asociada para su tipo de retorno:

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task   = std::packaged_task<decltype(func())()>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}
```

El hilo comienza a ejecutarse inmediatamente. Podemos separarlo o unirlo al final del alcance. Cuando la llamada de función a std :: thread finaliza, el resultado está listo.

Tenga en cuenta que esto es ligeramente diferente de `std::async` donde el `std::future` devuelto cuando se destruya se **bloqueará** hasta que el hilo finalice.

std :: future_error y std :: future_errc

Si las restricciones para std :: promise y std :: future no se cumplen, se lanza una excepción de tipo std :: future_error.

El miembro del código de error en la excepción es de tipo std :: future_errc y los valores son los siguientes, junto con algunos casos de prueba:

```
enum class future_errc {
    broken_promise             = /* the task is no longer shared */,
    future_already_retrieved   = /* the answer was already retrieved */,
    promise_already_satisfied  = /* the answer was stored already */,
    no_state                   = /* access to a promise in non-shared state */
};
```

Promesa inactiva

```
int test()
{
    std::promise<int> pr;
    return 0; // returns ok
}
```

Promesa activa, sin uso:

```
  int test()
    {
        std::promise<int> pr;
        auto fut = pr.get_future(); //blocks indefinitely!
        return 0;
    }
```

Doble recuperacion:

```
int test()
{
    std::promise<int> pr;
    auto fut1 = pr.get_future();

    try{
        auto fut2 = pr.get_future();    //   second attempt to get future
        return 0;
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The future has already been retrieved
from the promise or packaged_task."
        return -1;
    }
    return fut2.get();
}
```

Estableciendo std :: promesa de valor dos veces:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future();
    try{
        std::promise<int> pr2(std::move(pr));
        pr2.set_value(10);
        pr2.set_value(10);  // second attempt to set promise throws exception
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The state of the promise has already been
set."
        return -1;
    }
    return fut.get();
}
```

std :: futuro y std :: async

En el siguiente ejemplo de clasificación de combinación paralela ingenua, `std::async` se utiliza para iniciar múltiples tareas de combinación de combinación paralelas. `std::future` se usa para esperar los resultados y sincronizarlos:

```
#include <iostream>
using namespace std;


void merge(int low,int mid,int high, vector<int>&num)
{
    vector<int> copy(num.size());
    int h,i,j,k;
    h=low;
    i=low;
    j=mid+1;

    while((h<=mid)&&(j<=high))
    {
        if(num[h]<=num[j])
        {
            copy[i]=num[h];
            h++;
        }
        else
        {
            copy[i]=num[j];
            j++;
        }
        i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    else
    {
        for(k=h;k<=mid;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    for(k=low;k<=high;k++)
        swap(num[k],copy[k]);
}


void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low<high)
    {
        mid = low + (high-low)/2;
        auto future1    =  std::async(std::launch::deferred,[&]()
                                      {
                                        merge_sort(low,mid,num);
```

```
                                        });
        auto future2    =  std::async(std::launch::deferred, [&]()
                                        {
                                           merge_sort(mid+1,high,num) ;
                                        });

        future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}
```

Nota: En el ejemplo `std::async` se inicia con la directiva `std::launch_deferred` . Esto es para evitar que se cree un nuevo hilo en cada llamada. En el caso de nuestro ejemplo, las llamadas a `std::async` se hacen fuera de orden, se sincronizan en las llamadas para `std::future::get()` .

`std::launch_async` obliga a `std::launch_async` un nuevo hilo en cada llamada.

La política predeterminada es `std::launch::deferred| std::launch::async` , lo que significa que la implementación determina la política para crear nuevos subprocesos.

Clases de operaciones asincrónicas

- std :: async: realiza una operación asíncrona.
- std :: future: proporciona acceso al resultado de una operación asíncrona.
- std :: promise: empaqueta el resultado de una operación asíncrona.
- std :: packaged_task: agrupa una función y la promesa asociada para su tipo de retorno.

Lea Futuros y Promesas en línea: https://riptutorial.com/es/cplusplus/topic/9840/futuros-y-promesas

Capítulo 62: Generación de números aleatorios

Observaciones

La generación de números aleatorios en C ++ es proporcionada por el encabezado `<random>` . Este encabezado define dispositivos aleatorios, generadores pseudoaleatorios y distribuciones.

Los dispositivos aleatorios devuelven números aleatorios proporcionados por el sistema operativo. Deben utilizarse para la inicialización de generadores pseudoaleatorios o directamente con fines criptográficos.

Los generadores seudoaleatorios devuelven números enteros pseudoaleatorios basados en su semilla inicial. El rango de números pseudoaleatorios generalmente abarca todos los valores de un tipo sin signo. Todos los generadores seudoaleatorios en la biblioteca estándar devolverán los mismos números para la misma semilla inicial para todas las plataformas.

Las distribuciones consumen números aleatorios de generadores seudoaleatorios o dispositivos aleatorios y producen números aleatorios con la distribución necesaria. Las distribuciones no son independientes de la plataforma y pueden producir diferentes números para los mismos generadores con las mismas semillas iniciales en diferentes plataformas.

Examples

Generador de valor aleatorio verdadero

Para generar valores aleatorios verdaderos que se pueden usar para la criptografía `std::random_device` se debe usar como generador.

```
#include <iostream>
#include <random>

int main()
{
   std::random_device crypto_random_generator;
   std::uniform_int_distribution<int> int_distribution(0,9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(crypto_random_generator);
       actual_distribution[result]++;
   }

   for(int i = 0; i < 10; i++) {
       std::cout << actual_distribution[i] << " ";
   }
```

```
    return 0;
}
```

`std::random_device` se usa de la misma manera que se usa un generador de valores pseudoaleatorios.

Sin embargo, `std::random_device` **puede implementarse en términos de un motor de números pseudoaleatorios definido por la implementación** si una fuente no determinista (por ejemplo, un dispositivo de hardware) no está disponible para la implementación.

Detectar dichas implementaciones debería ser posible a través de la función de miembro de `entropy` (que devuelve cero cuando el generador es completamente determinista), pero muchas bibliotecas populares (tanto libstdc ++ de GCC como libc ++ de LLVM) siempre devuelven cero, incluso cuando utilizan aleatoriedad externa de alta calidad .

Generando un número pseudoaleatorio

Un generador de números pseudoaleatorios genera valores que se pueden calcular basándose en valores generados previamente. En otras palabras: es determinista. No utilice un generador de números pseudoaleatorios en situaciones donde se requiera un número aleatorio verdadero.

```
#include <iostream>
#include <random>

int main()
{
    std::default_random_engine pseudo_random_generator;
    std::uniform_int_distribution<int> int_distribution(0, 9);

    int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

    for(int i = 0; i < 10000; i++) {
        int result = int_distribution(pseudo_random_generator);
        actual_distribution[result]++;
    }

    for(int i = 0; i <= 9; i++) {
        std::cout << actual_distribution[i] << " ";
    }

    return 0;
}
```

Este código crea un generador de números aleatorios y una distribución que genera números enteros en el rango [0,9] con la misma probabilidad. Luego cuenta cuántas veces se generó cada resultado.

El parámetro de plantilla de `std::uniform_int_distribution<T>` especifica el tipo de entero que se debe generar. Use `std::uniform_real_distribution<T>` para generar flotantes o dobles.

Uso del generador para múltiples distribuciones.

El generador de números aleatorios puede (y debe) ser usado para múltiples distribuciones.

```
#include <iostream>
#include <random>

int main()
{
   std::default_random_engine pseudo_random_generator;
   std::uniform_int_distribution<int> int_distribution(0, 9);
   std::uniform_real_distribution<float> float_distribution(0.0, 1.0);
   std::discrete_distribution<int> rigged_dice({1,1,1,1,1,100});

   std::cout << int_distribution(pseudo_random_generator) << std::endl;
   std::cout << float_distribution(pseudo_random_generator) << std::endl;
   std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

   return 0;
}
```

En este ejemplo, solo se define un generador. Posteriormente se utiliza para generar un valor aleatorio en tres distribuciones diferentes. La distribución `rigged_dice` generará un valor entre 0 y 5, pero casi siempre genera un `5` , porque la posibilidad de generar un `5` es `100 / 105` .

Lea Generación de números aleatorios en línea:
https://riptutorial.com/es/cplusplus/topic/1541/generacion-de-numeros-aleatorios

Capítulo 63: Gestión de la memoria

Sintaxis

- : :(*opt*) nuevo (*expresión-lista*) (*opt*) *new-type-id new-initializer* (*opt*)
- : :(*opt*) nuevo (*expresión-lista*) (*opt*) (*type-id*) *new-initializer* (*opt*)
- : :(*opt)* borrar *fundido expresión*
- : :(*Opción*) borrar [] *expresión-cast*
- std :: unique_ptr < *type-id* > var_name (nuevo *type-id* (*opt*)); // C ++ 11
- std :: shared_ptr < *type-id* > var_name (nuevo *type-id* (*opt*)); // C ++ 11
- std :: shared_ptr < *type-id* > var_name = std :: make_shared < *type-id* > (*opt*); // C ++ 11
- std :: unique_ptr < *type-id* > var_name = std :: make_unique < *type-id* > (*opt*); // C ++ 14

Observaciones

Un líder `::` obliga a buscar el operador nuevo o de borrado en el ámbito global, anulando cualquier operador nuevo o de borrado específico de clase sobrecargado.

Los argumentos opcionales que siguen a la `new` palabra clave se usan generalmente para llamar ubicación nueva , pero también se pueden usar para pasar información adicional al asignador, como una etiqueta que solicita que la memoria se asigne de un grupo elegido.

El tipo asignado generalmente se especifica explícitamente, *por ejemplo,* `new Foo` , pero también se puede escribir como `auto` (desde C ++ 11) o `decltype(auto)` (desde C ++ 14) para deducirlo del inicializador.

La inicialización del objeto asignado ocurre de acuerdo con las mismas reglas que la inicialización de las variables locales. En particular, el objeto se inicializará por defecto si se omite el inicializador, y cuando se asigna dinámicamente un tipo escalar o una matriz de tipo escalar, no hay garantía de que la memoria se pondrá a cero.

Un objeto de matriz creado por una *nueva expresión se* debe destruir utilizando `delete[]` , independientemente de si la *nueva expresión* se escribió con `[]` o no. Por ejemplo:

```
using IA = int[4];
int* pIA = new IA;
delete[] pIA;  // right
// delete pIA;  // wrong
```

Examples

Apilar

La pila es una pequeña región de memoria en la que se colocan valores temporales durante la ejecución. La asignación de datos en la pila es muy rápida en comparación con la asignación de

almacenamiento dinámico, ya que toda la memoria ya se ha asignado para este propósito.

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

La pila se llama así porque las cadenas de llamadas a funciones tendrán su memoria temporal "apilada" una encima de la otra, cada una utilizando una pequeña sección separada de la memoria.

```
float bar() {
    //f will be placed on the stack after anything else
    float f = 2;
    return f;
}

double foo() {
    //d will be placed just after anything within main()
    double d = bar();
    return d;
}

int main() {
    //The stack has no user variables stored in it until foo() is called
    return (int)foo();
}
```

Los datos almacenados en la pila solo son válidos mientras el alcance que asignó la variable aún esté activo.

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //Undefined behavior, the value pointed to by pA is no longer in scope
    a = *pA;
}
```

Almacenamiento gratuito (Heap, asignación dinámica ...)

El término **'montón'** es un término de computación general que significa un área de la memoria desde la cual se pueden asignar y desasignar partes independientemente de la memoria proporcionada por la **pila** .

En C++ el *Estándar se* refiere a esta área como la **Tienda libre,** que se considera un término más preciso.

Las áreas de memoria asignadas desde **Free Store** pueden vivir más tiempo que el alcance original en el que se asignó. Los datos demasiado grandes para ser almacenados en la pila también pueden asignarse desde **Free Store** .

La memoria sin formato se puede asignar y desasignar mediante las palabras clave *new* y *delete* .

```
float *foo = nullptr;
{
    *foo = new float; // Allocates memory for a float
    float bar;              // Stack allocated
} // End lifetime of bar, while foo still alive

delete foo;          // Deletes the memory for the float at pF, invalidating the pointer
foo = nullptr;       // Setting the pointer to nullptr after delete is often considered good
practice
```

También es posible asignar matrices de tamaño fijo con *nuevas* y *eliminar* , con una sintaxis ligeramente diferente. La asignación de arrays no es compatible con la asignación sin arrays, y mezclar los dos llevará a la corrupción del montón. La asignación de una matriz también asigna memoria para rastrear el tamaño de la matriz para su posterior eliminación de una manera definida por la implementación.

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

Al usar *new* y *delete en* lugar de *malloc* y *free* , el constructor y el destructor se ejecutarán (similar a los objetos basados en la pila). Esta es la razón por la cual *nuevo* y *eliminar* son preferidos a *malloc* y *gratis* .

```
struct ComplexType {
    int a = 0;

    ComplexType() { std::cout << "Ctor" << std::endl; }
    ~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// Allocates memory for a ComplexType, and calls its constructor
ComplexType *foo = new ComplexType();
//Calls the destructor for ComplexType() and deletes memory for a Complextype at pC
delete foo;
```

C ++ 11

A partir de C ++ 11, se recomienda el uso de punteros inteligentes para indicar la propiedad.

C ++ 14

C ++ 14 agregó `std::make_unique` a la STL, cambiando la recomendación para favorecer `std::make_unique` o `std::make_shared` lugar de usar desnudo *nuevo* y *eliminar* .

Colocación nueva

Hay situaciones en las que no queremos confiar en Free Store para asignar memoria y queremos usar asignaciones de memoria personalizadas utilizando `new` .

Para estas situaciones, podemos usar `Placement New` , donde podemos decirle al operador 'nuevo' que asigne memoria desde una ubicación de memoria asignada

Por ejemplo

```
int a4byteInteger;

char *a4byteChar = new (&a4byteInteger) char[4];
```

En este ejemplo, la memoria apuntada por `a4byteChar` es de 4 bytes asignados a 'pila' a través de la variable entera `a4byteInteger` .

El beneficio de este tipo de asignación de memoria es el hecho de que los programadores controlan la asignación. En el ejemplo anterior, dado que `a4byteInteger` está asignado en la pila, no necesitamos hacer una llamada explícita para 'eliminar a4byteChar`.

El mismo comportamiento se puede lograr para la memoria asignada dinámica también. Por ejemplo

```
int *a8byteDynamicInteger = new int[2];

char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

En este caso, el puntero de memoria de `a8byteChar` se `a8byteChar` a la memoria dinámica asignada por `a8byteDynamicInteger` . En este caso, sin embargo, debemos llamar explícitamente a `delete a8byteDynamicInteger` para liberar la memoria

Otro ejemplo para la clase de C ++

```
struct ComplexType {
    int a;

    ComplexType() : a(0) {}
    ~ComplexType() {}
};

int main() {
    char* dynArray = new char[256];

    //Calls ComplexType's constructor to initialize memory as a ComplexType
    new((void*)dynArray) ComplexType();

    //Clean up memory once we're done
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();
    delete[] dynArray;

    //Stack memory can also be used with placement new
    alignas(ComplexType) char localArray[256]; //alignas() available since C++11

    new((void*)localArray) ComplexType();
```

```
    //Only need to call the destructor for stack memory
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();

    return 0;
}
```

Lea Gestión de la memoria en línea: https://riptutorial.com/es/cplusplus/topic/2873/gestion-de-la-memoria

Capítulo 64: Herramientas y Técnicas de Depuración y Prevención de Depuración de C ++

Introducción

Mucho tiempo de los desarrolladores de C ++ se dedica a la depuración. El objetivo de este tema es ayudar con esta tarea y proporcionar inspiración para las técnicas. No espere una lista extensa de problemas y soluciones corregidos por las herramientas o un manual sobre las herramientas mencionadas.

Observaciones

Este tema aún no está completo, los ejemplos sobre las siguientes técnicas / herramientas serían útiles:

- Mencione más herramientas de análisis estático
- Herramientas de instrumentación binaria (como UBSan, TSan, MSan, ESan ...)
- Endurecimiento (CFI ...)
- Fuzzing

Examples

Mi programa de C ++ termina con segfault - valgrind

Tengamos un programa básico de falla:

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p3 << std::endl;
    }
}

int main() {
    fail();
}
```

Constrúyalo (agregue -g para incluir información de depuración):

```
g++ -g -o main main.cpp
```

Correr:

```
$ ./main
Segmentation fault (core dumped)
$
```

Vamos a depurarlo con valgrind:

```
$ valgrind ./main
==8515== Memcheck, a memory error detector
==8515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==8515== Command: ./main
==8515==
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==8515==
==8515==
==8515== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==8515==  Access not within mapped region at address 0x0
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  If you believe this happened as a result of a stack
==8515==  overflow in your program's main thread (unlikely but
==8515==  possible), you can try to increase the size of the
==8515==  main thread stack using the --main-stacksize= flag.
==8515==  The main thread stack size used in this run was 8388608.
==8515==
==8515== HEAP SUMMARY:
==8515==     in use at exit: 72,704 bytes in 1 blocks
==8515==   total heap usage: 1 allocs, 0 frees, 72,704 bytes allocated
==8515==
==8515== LEAK SUMMARY:
==8515==    definitely lost: 0 bytes in 0 blocks
==8515==    indirectly lost: 0 bytes in 0 blocks
==8515==      possibly lost: 0 bytes in 0 blocks
==8515==    still reachable: 72,704 bytes in 1 blocks
==8515==         suppressed: 0 bytes in 0 blocks
==8515== Rerun with --leak-check=full to see details of leaked memory
==8515==
==8515== For counts of detected and suppressed errors, rerun with: -v
==8515== Use --track-origins=yes to see where uninitialised values come from
==8515== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
$
```

Primero nos centramos en este bloque:

```
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

La primera línea nos dice que segfault es causado por la lectura de 4 bytes. Las líneas segunda y tercera son de pila de llamadas. Significa que la lectura no válida se realiza en la función `fail()` , línea 8 de main.cpp, a la que llama main, línea 13 de main.cpp.

Mirando la línea 8 de main.cpp vemos

```
std::cout << *p3 << std::endl;
```

Pero primero revisamos el puntero, entonces, ¿qué pasa? Veamos el otro bloque:

```
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
```

Nos dice que hay una variable unificada en la línea 7 y la leemos:

```
if (p3) {
```

Lo que nos señala la línea donde comprobamos p3 en lugar de p2. Pero, ¿cómo es posible que p3 no esté inicializado? Lo inicializamos por:

```
int *p3 = p1;
```

Valgrind nos aconseja volver a ejecutar con `--track-origins=yes` , hagámoslo:

```
valgrind --track-origins=yes ./main
```

El argumento para valgrind es justo después de valgrind. Si lo ponemos después de nuestro programa, sería pasado a nuestro programa.

La salida es casi la misma, solo hay una diferencia:

```
==8517== Conditional jump or move depends on uninitialised value(s)
==8517==    at 0x400813: fail() (main.cpp:7)
==8517==    by 0x40083F: main (main.cpp:13)
==8517==  Uninitialised value was created by a stack allocation
==8517==    at 0x4007F6: fail() (main.cpp:3)
```

Lo que nos dice que el valor no inicializado que usamos en la línea 7 se creó en la línea 3:

```
int *p1;
```

Lo que nos guía a nuestro puntero sin inicializar.

Análisis de Segfault con GDB

Vamos a usar el mismo código que el anterior para este ejemplo.

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

Primero vamos a compilarlo.

```
g++ -g -o main main.cpp
```

Vamos a ejecutarlo con gdb

```
gdb ./main
```

Ahora estaremos en gdb shell. Escriba ejecutar.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/opencog/code-snippets/stackoverflow/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400850 in fail () at debugging_with_gdb.cc:11
11            std::cout << *p2 << std::endl;
```

Vemos que la falla de segmentación está ocurriendo en la línea 11. Por lo tanto, la única variable que se usa en esta línea es el puntero p2. Permite examinar su contenido escribiendo print.

```
(gdb) print p2
$1 = (int *) 0x0
```

Ahora vemos que p2 se inicializó a 0x0, lo que significa NULL. En esta línea, sabemos que estamos intentando eliminar la referencia de un puntero NULO. Así que vamos y lo arreglamos.

Código limpio

La depuración comienza con la comprensión del código que está intentando depurar.

Código incorrecto:

```
int main() {
    int value;
    std::vector<int> vectorToSort;
```

```
    vectorToSort.push_back(42); vectorToSort.push_back(13);
    for (int i = 52; i; i = i - 1)
    {
    vectorToSort.push_back(i *2);
    }
    /// Optimized for sorting small vectors
    if (vectorToSort.size() == 1);
    else
        {
        if (vectorToSort.size() <= 2)
            std::sort(vectorToSort.begin(), std::end(vectorToSort));
        }
    for (value : vectorToSort) std::cout << value << ' ';
return 0; }
```

Mejor código:

```
std::vector<int> createSemiRandomData() {
    std::vector<int> data;
    data.push_back(42);
    data.push_back(13);
    for (int i = 52; i; --i)
        vectorToSort.push_back(i *2);
    return data;
}

/// Optimized for sorting small vectors
void sortVector(std::vector &v) {
    if (vectorToSort.size() == 1)
        return;
    if (vectorToSort.size() > 2)
        return;

    std::sort(vectorToSort.begin(), vectorToSort.end());
}

void printVector(const std::vector<int> &v) {
    for (auto i : v)
        std::cout << i << ' ';
}

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
 }
```

Independientemente de los estilos de codificación que prefiera y use, tener un estilo de codificación (y formato) consistente lo ayudará a comprender el código.

Mirando el código anterior, uno puede identificar un par de mejoras para mejorar la legibilidad y la depuración:

El uso de funciones separadas para acciones separadas.

El uso de funciones separadas le permite saltear algunas funciones en el depurador si no está interesado en los detalles. En este caso específico, es posible que no esté interesado en la creación o impresión de los datos y solo desee ingresar en la clasificación.

Otra ventaja es que necesita leer menos código (y memorizarlo) mientras recorre el código. Ahora solo necesita leer 3 líneas de código en `main()` para poder entenderlo, en lugar de toda la función.

La tercera ventaja es que simplemente tiene menos código que ver, lo que ayuda a un ojo entrenado a detectar este error en segundos.

Usando formateo / construcciones consistentes

El uso de un formato y construcciones consistentes eliminará el desorden del código, lo que hará que sea más fácil concentrarse en el código en lugar de texto. Se han alimentado muchas discusiones sobre el estilo de formato 'correcto'. Independientemente de ese estilo, tener un solo estilo consistente en el código mejorará la familiaridad y hará que sea más fácil enfocarse en el código.

Como el código de formato es una tarea que requiere mucho tiempo, se recomienda usar una herramienta dedicada para esto. La mayoría de los IDE tienen al menos algún tipo de soporte para esto y pueden hacer un formateo más consistente que los humanos.

Es posible que tenga en cuenta que el estilo no se limita a espacios y líneas nuevas, ya que ya no mezclamos el estilo libre y las funciones miembro para comenzar / finalizar el contenedor. (`v.begin()` VS `std::end(v)`).

Señala la atención a las partes importantes de tu código.

Independientemente del estilo que decida elegir, el código anterior contiene un par de marcadores que podrían darle una pista sobre lo que podría ser importante:

- Un comentario afirmando `optimized` , esto indica algunas técnicas de fantasía.
- Algunos resultados tempranos en `sortVector()` indican que estamos haciendo algo especial
- El `std::ref()` indica que algo está sucediendo con el `sortVector()`

Conclusión

Tener un código limpio lo ayudará a comprender el código y reducirá el tiempo que necesita para depurarlo. En el segundo ejemplo, un revisor de código podría incluso detectar el error a primera vista, mientras que el error podría estar oculto en los detalles del primero. (PS: El error está en la comparación con `2`)

Análisis estático

El análisis estático es la técnica mediante la cual el código comprueba los patrones vinculados a errores conocidos. Sin embargo, el uso de esta técnica requiere menos tiempo que la revisión de

un código, sin embargo, sus verificaciones solo se limitan a las programadas en la herramienta.

Las comprobaciones pueden incluir el punto y coma incorrecto detrás de la sentencia `if (var);` (`if (var);`) hasta algoritmos de gráficos avanzados que determinan si una variable no está inicializada.

Advertencias del compilador

Habilitar el análisis estático es fácil, la versión más simplista ya está incorporada en su compilador:

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

Si habilita estas opciones, notará que cada compilador encontrará errores que otros no, y que obtendrá errores en técnicas que podrían ser válidas en un contexto específico. `while (staticAtomicBool);` podría ser aceptable incluso si `while (localBool);` no es

Así que, a diferencia de la revisión de código, estás luchando contra una herramienta que entiende tu código, te dice muchos errores útiles y, a veces, no está de acuerdo contigo. En este último caso, es posible que deba suprimir la advertencia localmente.

Como las opciones anteriores habilitan todas las advertencias, pueden habilitar las advertencias que no desea. (¿Por qué debería ser compatible su código con C ++ 98?) Si es así, simplemente puede desactivar esa advertencia específica:

- `clang++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `g++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

Cuando las advertencias del compilador le ayudan durante el desarrollo, ralentizan bastante la compilación. Es por eso que es posible que no siempre desee habilitarlos de forma predeterminada. O los ejecuta de forma predeterminada o habilita cierta integración continua con los cheques más caros (o todos ellos).

Herramientas externas

Si decide tener alguna integración continua, el uso de otras herramientas no es tan difícil. Una herramienta como clang-tidy tiene una lista de comprobaciones que cubre una amplia gama de problemas, algunos ejemplos:

- Errores reales
  - Prevención de rebanar
  - Afirma con efectos secundarios.
- Controles de legibilidad
  - Sangría engañosa
  - Comprobar el nombre del identificador
- Controles de modernización

    - Utilice make_unique ()
    - Utilizar nullptr
- Verificaciones de rendimiento
    - Encuentra copias innecesarias
    - Encuentra ineficientes llamadas de algoritmo

Es posible que la lista no sea tan grande, ya que Clang ya tiene muchas advertencias de compilación, sin embargo, te acercará un paso más a una base de código de alta calidad.

Otras herramientas

Existen otras herramientas con fines similares, como:

- El analizador estático visual studio como herramienta externa.
- Clazy , un plugin de compilador Clang para verificar el código Qt

Conclusión

Existen muchas herramientas de análisis estático para C ++, ambas integradas en el compilador como herramientas externas. Probarlos no toma mucho tiempo para configuraciones fáciles y encontrarán errores que podría pasar por alto en la revisión del código.

Apilamiento seguro (corrupciones de la pila)

Las corrupciones de la pila son errores molestos a la vista. Como la pila está dañada, el depurador a menudo no puede darle un buen seguimiento de la pila de dónde se encuentra y cómo llegó allí.

Aquí es donde entra en juego la pila segura. En lugar de usar una sola pila para tus hilos, usará dos: una pila segura y una pila peligrosa. La pila segura funciona exactamente igual que antes, excepto que algunas partes se mueven a la pila peligrosa.

¿Qué partes de la pila se mueven?

Cada parte que tenga el potencial de dañar la pila se moverá de la pila segura. Tan pronto como una variable en la pila se pasa por referencia o una toma la dirección de esta variable, el compilador decidirá asignarla en la segunda pila en lugar de la segura.

Como resultado, cualquier operación que realice con esos punteros, cualquier modificación que realice en la memoria (basada en esos punteros / referencias) solo puede afectar a la memoria en la segunda pila. Como uno nunca recibe un puntero que está cerca de la pila segura, la pila no puede corromper la pila y el depurador aún puede leer todas las funciones de la pila para dar un buen seguimiento.

¿Para qué se usa realmente?

La pila segura no se inventó para darle una mejor experiencia de depuración, sin embargo, es un efecto secundario agradable para los insectos desagradables. Su propósito original es como parte del Proyecto de integridad de puntero de código (CPI) , en el que intentan evitar la anulación de las direcciones de retorno para evitar la inyección de código. En otras palabras, intentan evitar la ejecución de un código de hackers.

Por este motivo, la función se ha activado en cromo y se ha informado que tiene una sobrecarga de CPU <1%.

¿Cómo habilitarlo?

En este momento, la opción solo está disponible en el compilador `-fsanitize=safe-stack` , donde se puede pasar `-fsanitize=safe-stack` al compilador. Se hizo una propuesta para implementar la misma característica en GCC.

Conclusión

La corrupción de la pila puede ser más fácil de depurar cuando la pila segura está habilitada. Debido a una baja sobrecarga de rendimiento, incluso puede activarse de forma predeterminada en la configuración de su compilación.

Lea Herramientas y Técnicas de Depuración y Prevención de Depuración de C ++ en línea: https://riptutorial.com/es/cplusplus/topic/9814/herramientas-y-tecnicas-de-depuracion-y-prevencion-de-depuracion-de-c-plusplus

Capítulo 65: Idioma Pimpl

Observaciones

El **idioma pimpl (p** ointer a **impl** ementation, a veces se hace referencia a la *técnica puntero* o *gato de Cheshire* como *opaco),* reduce los tiempos de compilación de una clase moviendo todos sus miembros de datos privados en una estructura definida en el archivo .cpp.

La clase posee un puntero a la implementación. De esta manera, se puede reenviar, para que el archivo de encabezado no necesite `#include` clases que se usan en las variables de miembros privados.

Cuando se utiliza el lenguaje pimpl, cambiar un miembro de datos privados no requiere volver a compilar las clases que dependen de él.

Examples

Lenguaje básico de Pimpl

C ++ 11

En el archivo de cabecera:

```
// widget.h

#include <memory>  // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
    public:
        Widget();
        ~Widget();
        void DoSomething();

    private:
        // the pImpl idiom is named after the typical variable name used
        // ie, pImpl:
        struct Impl;                    // forward declaration
        std::experimental::propagate_const<std::unique_ptr< Impl >> pImpl;  // ptr to actual
implementation
};
```

En el archivo de implementación:

```
// widget.cpp

#include "widget.h"
#include "reallycomplextype.h" // no need to include this header inside widget.h
```

```
struct Widget::Impl
{
    // the attributes needed from Widget go here
    ReallyComplexType rct;
};

Widget::Widget() :
    pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // do the stuff here with pImpl
}
```

La `pImpl` contiene el estado del `Widget` (o parte / la mayor parte de él). En lugar de que la descripción del estado del `Widget` se exponga en el archivo de encabezado, solo se puede exponer dentro de la implementación.

`pImpl` significa "puntero a la implementación". La implementación "real" de `Widget` está en `pImpl` .

Peligro: `unique_ptr` cuenta que para que esto funcione con `unique_ptr` , `~Widget()` debe implementarse en un punto en un archivo donde el `Impl` sea completamente visible. Puede `=default` allí, pero si `=default` donde `Impl` no está definido, el programa puede fácilmente tener un formato incorrecto, no se requiere diagnóstico.

Lea Idioma Pimpl en línea: https://riptutorial.com/es/cplusplus/topic/2143/idioma-pimpl

Capítulo 66: Implementación de patrones de diseño en C ++

Introducción

En esta página, puede encontrar ejemplos de cómo se implementan los patrones de diseño en C ++. Para obtener detalles sobre estos patrones, puede consultar la documentación de patrones de diseño .

Observaciones

Un patrón de diseño es una solución general reutilizable para un problema que ocurre comúnmente dentro de un contexto dado en el diseño de software.

Examples

Patrón observador

La intención de Observer Pattern es definir una dependencia de uno a muchos entre objetos para que cuando un objeto cambie de estado, todos sus dependientes sean notificados y actualizados automáticamente.

El sujeto y los observadores definen la relación uno a muchos. Los observadores dependen del tema, de modo que cuando el estado del sujeto cambia, los observadores reciben una notificación. Dependiendo de la notificación, los observadores también pueden actualizarse con nuevos valores.

Este es el ejemplo del libro "Patrones de diseño" de Gamma.

```
#include <iostream>
#include <vector>

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
     virtual ~Subject() = default;
     void Attach(Observer& o) { observers.push_back(&o); }
     void Detach(Observer& o)
     {
```

```
        observers.erase(std::remove(observers.begin(), observers.end(), &o));
    }
    void Notify()
    {
        for (auto* o : observers) {
            o->Update(*this);
        }
    }
private:
    std::vector<Observer*> observers;
};

class ClockTimer : public Subject
{
public:

    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
        this->minute = minute;
        this->second = second;

        Notify();
    }

    int GetHour() const { return hour; }
    int GetMinute() const { return minute; }
    int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
    explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~DigitalClock() { subject.Detach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "Digital time is " << hour << ":"
                  << minute << ":"
                  << second << std::endl;
    }

private:
    ClockTimer& subject;
};
```

```
class AnalogClock: public Observer
{
public:
     explicit AnalogClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
     ~AnalogClock() { subject.Detach(*this); }
     void Update(Subject& theChangedSubject) override
     {
         if (&theChangedSubject == &subject) {
             Draw();
         }
     }
     void Draw()
     {
         int hour = subject.GetHour();
         int minute = subject.GetMinute();
         int second = subject.GetSecond();

         std::cout << "Analog time is " << hour << ":"
                   << minute << ":"
                   << second << std::endl;
     }
private:
     ClockTimer& subject;
};

int main()
{
    ClockTimer timer;

    DigitalClock digitalClock(timer);
    AnalogClock analogClock(timer);

    timer.SetTime(14, 41, 36);
}
```

Salida:

```
Digital time is 14:41:36
Analog time is 14:41:36
```

Aquí está el resumen del patrón:

1. Los objetos (objeto `DigitalClock` o `AnalogClock`) utilizan las interfaces de Asunto (`Attach()` o `Detach()`) para suscribirse (registrarse) como observadores o anular la suscripción (eliminar) de ser observadores (`subject.Attach(*this); subject.Detach(*this);`

2. Cada sujeto puede tener muchos observadores (`vector<Observer*> observers;`).

3. Todos los observadores necesitan implementar la interfaz de observador. Esta interfaz solo tiene un método, `Update()` , que se llama cuando cambia el estado del sujeto (`Update(Subject &)`)

4. Además de los métodos `Attach()` y `Detach()` , el sujeto concreto implementa un método `Notify()` que se utiliza para actualizar a todos los observadores actuales cada vez que cambia el estado. Pero en este caso, todos ellos se realizan en la clase principal, `Subject` (

`Subject::Attach (Observer&)` , `void Subject::Detach(Observer&)` y `void Subject::Notify()` .

5. El objeto concreto también puede tener métodos para establecer y obtener su estado.
6. Los observadores concretos pueden ser de cualquier clase que implemente la interfaz Observer. Cada observador se suscribe (registra) con un sujeto concreto para recibir la actualización (`subject.Attach(*this);`).
7. Los dos objetos de Observer Pattern están **ligeramente acoplados** , pueden interactuar pero con poco conocimiento el uno del otro.

Variación:

Señal y ranuras

Signals and Slots es una construcción de lenguaje introducida en Qt, que facilita la implementación del patrón Observer y evita el código repetitivo. El concepto es que los controles (también conocidos como widgets) pueden enviar señales que contienen información de eventos que pueden ser recibidos por otros controles utilizando funciones especiales conocidas como ranuras. La ranura en Qt debe ser un miembro de la clase declarado como tal. El sistema de señal / ranura encaja bien con la forma en que se diseñan las interfaces gráficas de usuario. De manera similar, el sistema de señal / ranura se puede usar para notificaciones de eventos de E / S asíncronas (incluidos sockets, tuberías, dispositivos serie, etc.) o para asociar eventos de tiempo de espera con instancias de objetos, métodos o funciones apropiadas. No es necesario escribir ningún código de registro / cancelación de registro / invocación, porque el Meta Object Compiler (MOC) de Qt genera automáticamente la infraestructura necesaria.

El lenguaje C # también admite una construcción similar, aunque con una terminología y una sintaxis diferentes: los eventos desempeñan el papel de las señales y los delegados son los espacios. Además, un delegado puede ser una variable local, como un puntero a una función, mientras que una ranura en Qt debe ser un miembro de la clase declarado como tal.

Patrón de adaptador

Convertir la interfaz de una clase en otra interfaz que los clientes esperan. El adaptador (o Wrapper) permite que las clases trabajen juntas y que de otra manera no podrían debido a interfaces incompatibles. La motivación del patrón de adaptador es que podemos reutilizar el software existente si podemos modificar la interfaz.

1. Patrón de adaptador se basa en la composición del objeto.
2. El cliente llama a la operación en el objeto adaptador.
3. El adaptador llama a Adaptee para realizar la operación.
4. En STL, pila adaptada del vector: cuando la pila ejecuta push (), el vector subyacente hace vector :: push_back ().

Ejemplo:

```
#include <iostream>

// Desired interface (Target)
class Rectangle
{
  public:
    virtual void draw() = 0;
};

// Legacy component (Adaptee)
class LegacyRectangle
{
  public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }
    void oldDraw() {
        std::cout << "LegacyRectangle:  oldDraw(). \n";
    }
  private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
         std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};

int main()
{
  int x = 20, y = 50, w = 300, h = 200;
  Rectangle *r = new RectangleAdapter(x,y,w,h);
  r->draw();
}

//Output:
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)
```

Resumen del código:

1. El cliente cree que está hablando con un `Rectangle`

2. El objetivo es la clase `Rectangle` . Esto es en lo que el cliente invoca el método.

```
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();
```

3. Tenga en cuenta que la clase de adaptador utiliza herencia múltiple.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}
```

4. El adaptador `RectangleAdapter` permite que `LegacyRectangle` responda a request (`draw()` en un `Rectangle`) heredando AMBAS clases.

5. La clase `LegacyRectangle` no tiene los mismos métodos (`draw()`) que `Rectangle` , pero el `Adapter(RectangleAdapter)` puede tomar las llamadas al método `Rectangle` y girar e invocar el método en el `LegacyRectangle` , `oldDraw()` .

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};
```

El patrón de diseño del **adaptador** traduce la interfaz para una clase en una interfaz compatible pero diferente. Por lo tanto, esto es similar al patrón de **proxy** en que es un contenedor de un solo componente. Pero la interfaz para la clase de adaptador y la clase original pueden ser diferentes.

Como hemos visto en el ejemplo anterior, este patrón de **adaptador** es útil para exponer una interfaz diferente para una API existente para permitir que funcione con otro código. Además, al usar un patrón de adaptador, podemos tomar interfaces heterogéneas y transformarlas para proporcionar una API consistente.

El patrón de Bridge tiene una estructura similar a un adaptador de objetos, pero Bridge tiene una intención diferente: está destinado a **separar** una interfaz de su implementación para que puedan variarse de forma fácil e independiente. Un **adaptador** está destinado a **cambiar la interfaz** de un objeto **existente** .

Patrón de fábrica

El patrón de fábrica desacopla la creación de objetos y permite la creación por nombre utilizando una interfaz común:

```
class Animal{
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
    virtual std::string  getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};


class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string  getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string&   name )
    {
      if ( name == "bear" )
        return std::make_shared<Bear>();
      if ( name == "cat" )
        return std::shared_ptr<Cat>();

    return nullptr;
    }


};
```

Patrón de constructor con API fluida

El patrón del generador desacopla la creación del objeto del propio objeto. La idea principal detrás es que **un objeto no tiene que ser responsable de su propia creación** . El ensamblaje correcto y válido de un objeto complejo puede ser una tarea complicada en sí misma, por lo que esta tarea puede delegarse a otra clase.

Inspirado por el Email Builder en C # , he decidido hacer una versión de C ++ aquí. Un objeto de correo electrónico no es necesariamente un *objeto muy complejo* , pero puede demostrar el patrón.

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

// Forward declaring the builder
class EmailBuilder;

class Email
{
  public:
    friend class EmailBuilder;  // the builder can access Email's privates

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;
        stream << "from: " << m_from
               << "\nto: " << m_to
               << "\nsubject: " << m_subject
               << "\nbody: " << m_body;
        return stream.str();
    }

  private:
    Email() = default; // restrict construction to builder

    string m_from;
    string m_to;
    string m_subject;
    string m_body;
};

class EmailBuilder
{
  public:
    EmailBuilder& from(const string &from) {
        m_email.m_from = from;
        return *this;
    }

    EmailBuilder& to(const string &to) {
        m_email.m_to = to;
        return *this;
    }

    EmailBuilder& subject(const string &subject) {
        m_email.m_subject = subject;
        return *this;
    }

    EmailBuilder& body(const string &body) {
        m_email.m_body = body;
        return *this;
    }

    operator Email&&() {
        return std::move(m_email); // notice the move
    }
```

```
  private:
    Email m_email;
};

EmailBuilder Email::make()
{
    return EmailBuilder();
}

// Bonus example!
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}


int main()
{
    Email mail = Email::make().from("me@mail.com")
                              .to("you@mail.com")
                              .subject("C++ builders")
                              .body("I like this API, don't you?");

    cout << mail << endl;
}
```

Para versiones anteriores de C ++, uno puede simplemente ignorar la operación `std::move` y eliminar el && del operador de conversión (aunque esto creará una copia temporal).

El constructor finaliza su trabajo cuando libera el correo electrónico creado por el `operator Email&&()` . En este ejemplo, el constructor es un objeto temporal y devuelve el correo electrónico antes de ser destruido. También puede usar una operación explícita como `Email EmailBuilder::build() {...}` lugar del operador de conversión.

Pasar el constructor alrededor

Una gran característica que proporciona el patrón **de creación** es la capacidad de **usar varios actores para construir un objeto juntos.** Esto se hace pasando el constructor a los otros actores que cada uno dará más información al objeto construido. Esto es especialmente poderoso cuando está construyendo algún tipo de consulta, agregando filtros y otras especificaciones.

```
void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
           .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("I know the subject")
           .body("And the body. Someone else knows the addresses.");
}

int main()
```

```
{
    EmailBuilder builder;
    add_addresses(builder);
    compose_mail(builder);

    Email mail = builder;
    cout << mail << endl;
}
```

Variante de diseño: objeto mutable

Puede cambiar el diseño de este patrón para adaptarse a sus necesidades. Te daré una variante.

En el ejemplo dado, el objeto de correo electrónico es inmutable, es decir, sus propiedades no se pueden modificar porque no hay acceso a ellas. Esta era una característica deseada. Si necesita modificar el objeto después de su creación, debe proporcionarle algunos configuradores. Dado que esos definidores se duplicarían en el constructor, puede considerar hacerlo todo en una clase (ya no se necesita ninguna clase de constructor). Sin embargo, consideraría la necesidad de hacer que el objeto construido sea mutable en primer lugar.

Lea Implementación de patrones de diseño en C ++ en línea: https://riptutorial.com/es/cplusplus/topic/4335/implementacion-de-patrones-de-diseno-en-c-plusplus

Capítulo 67: Incompatibilidades C

Introducción

Esto describe qué código C se romperá en un compilador de C ++.

Examples

Palabras clave reservadas

El primer ejemplo son palabras clave que tienen un propósito especial en C ++: lo siguiente es legal en C, pero no en C ++.

```
int class = 5
```

Estos errores son fáciles de solucionar: simplemente cambie el nombre de la variable.

Punteros débilmente escritos

En C, los punteros se pueden convertir en un `void*` , que necesita una conversión explícita en C ++. Lo siguiente es ilegal en C ++, pero legal en C:

```
void* ptr;
int* intptr = ptr;
```

Agregar un reparto explícito hace que esto funcione, pero puede causar problemas adicionales.

goto o cambiar

En C ++, no puede omitir las inicializaciones con `goto` o `switch` . Lo siguiente es válido en C, pero no en C ++:

```
goto foo;
int skipped = 1;
foo;
```

Estos errores pueden requerir un nuevo diseño.

Lea Incompatibilidades C en línea:
https://riptutorial.com/es/cplusplus/topic/9645/incompatibilidades-c

Capítulo 68: Inferencia de tipos

Introducción

Este tema trata sobre la inferencia de tipos que implica el tipo `auto` palabra clave que está disponible en C ++ 11.

Observaciones

Por lo general, es mejor declarar `const` , `&` y `constexpr` siempre que use `auto` si alguna vez se requiere para prevenir comportamientos no deseados como copiar o mutaciones. Esos consejos adicionales aseguran que el compilador no genere ninguna otra forma de inferencia. Tampoco es aconsejable el uso excesivo `auto` y debe utilizarse solo cuando la declaración real es muy larga, especialmente con las plantillas STL.

Examples

Tipo de datos: Auto

Este ejemplo muestra las inferencias de tipo básico que puede realizar el compilador.

```
auto a = 1;         //    a = int
auto b = 2u;        //    b = unsigned int
auto c = &a;        //    c = int*
const auto  d = c; //    d = const int*
const auto& e = b; //    e = const unsigned int&

auto x = a + b      //    x = int, #compiler warning unsigned and signed

auto v = std::vector<int>;    //    v = std::vector<int>
```

Sin embargo, la palabra clave auto no siempre realiza la inferencia de tipo esperada sin sugerencias adicionales para `&` o `const` o `constexpr`

```
//    y = unsigned int,
//    note that y does not infer as const unsigned int&
//    The compiler would have generated a copy instead of a reference value to e or b
auto y = e;
```

Lambda auto

La palabra clave auto del tipo de datos es una forma conveniente para que los programadores declaren funciones lambda. Ayuda al acortar la cantidad de texto que los programadores necesitan escribir para declarar un puntero a función.

```
auto DoThis = [](int a, int b) { return a + b; };
```

```
//    Do this is of type (int)(*DoThis)(int, int)
//    else we would have to write this long
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2);    //    c = int
auto d = pDothis(1, 2);   //    d = int

//    using 'auto' shortens the definition for lambda functions
```

De forma predeterminada, si el tipo de retorno de las funciones lambda no está definido, se deducirá automáticamente de los tipos de expresión de retorno.

Estos 3 es básicamente lo mismo.

```
[](int a, int b) -> int  { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

Bucles y auto

Este ejemplo muestra cómo se puede usar auto para acortar la declaración de tipos para los bucles

```
std::map<int, std::string> Map;
for (auto pair : Map)            //    pair = std::pair<int, std::string>
for (const auto pair : Map)      //    pair = const std::pair<int, std::string>
for (const auto& pair : Map)     //    pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i)         //    i = int
for (auto i = 0; i < Map.size(); ++i)   //    Note that i = int and not size_t
for (auto i = Map.size(); i > 0; --i)   //    i = size_t
```

Lea Inferencia de tipos en línea: https://riptutorial.com/es/cplusplus/topic/8233/inferencia-de-tipos

Capítulo 69: Internacionalización en C ++

Observaciones

El lenguaje C ++ no dicta ningún conjunto de caracteres, algunos compiladores pueden ***admitir*** el uso de UTF-8, o incluso de UTF-16. Sin embargo, no hay certeza de que se proporcione algo más que simples caracteres ANSI / ASCII.

Por lo tanto, toda la compatibilidad con idiomas internacionales está definida por la implementación, y depende de qué plataforma, sistema operativo y compilador esté utilizando.

Varias bibliotecas de terceros (como la Biblioteca del Comité Internacional de Unicode) que pueden utilizarse para ampliar el soporte internacional de la plataforma.

Examples

Entendiendo las características de la cadena C ++

```
#include <iostream>
#include <string>

int main()
{
    const char * C_String = "This is a line of text w";
    const char * C_Problem_String = "This is a line of text ";
  std::string Std_String("This is a second line of text w");
    std::string Std_Problem_String("This is a second line of ex ");

  std::cout << "String Length: " << Std_String.length() << '\n';
    std::cout << "String Length: " << Std_Problem_String.length() << '\n';

    std::cout << "CString Length: " << strlen(C_String) << '\n';
    std::cout << "CString Length: " << strlen(C_Problem_String) << '\n';
    return 0;
}
```

Según la plataforma (Windows, OSX, etc.) y el compilador (GCC, MSVC, etc.), este programa **puede no compilar, mostrar diferentes valores o mostrar los mismos valores** .

Ejemplo de salida bajo el compilador de Microsoft MSVC:

> Longitud de la cuerda: 31
> Longitud de la cuerda: 31
> Longitud CString: 24
> Longitud CString: 24

Esto muestra que bajo MSVC cada uno de los caracteres extendidos utilizados se considera un "carácter" único, y esta plataforma es totalmente compatible con los idiomas internacionalizados. *Sin embargo, debe tenerse en cuenta que este comportamiento es inusual, estos caracteres*

internacionales se almacenan internamente como Unicode y, por lo tanto, tienen varios bytes de longitud. ***Esto puede causar errores inesperados.***

Bajo el compilador GNC / GCC la salida del programa es:

> Longitud de la cuerda: 31
> Longitud de la cuerda: 36
> Longitud CString: 24
> Longitud CString: 26

Este ejemplo demuestra que si bien el compilador GCC utilizado en esta plataforma (Linux) admite estos caracteres extendidos, también usa (*correctamente)* varios bytes para almacenar un carácter individual.
En este caso, es posible el uso de caracteres Unicode, pero el programador debe tener mucho cuidado al recordar que la longitud de una "cadena" en este escenario es la ***longitud en bytes*** , no la ***longitud en caracteres legibles*** .

Estas diferencias se deben a la forma en que se manejan los idiomas internacionales por plataforma y, lo que es más importante, que las cadenas C y C ++ utilizadas en este ejemplo pueden considerarse ***una matriz de bytes*** , por lo que (para este uso) ***el lenguaje C ++ considera Un carácter (char) para ser un solo byte*** .

Lea Internacionalización en C ++ en línea:
https://riptutorial.com/es/cplusplus/topic/5270/internacionalizacion-en-c-plusplus

Capítulo 70: Iteración

Examples

descanso

Salta del bucle de cierre más cercano o de la instrucción `switch` .

```
// print the numbers to a file, one per line
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d\n", num);
    if (errno == ENOSPC) {
        fprintf(stderr, "no space left on device; output will be truncated\n");
        break;
    }
}
```

continuar

Salta al final del bucle envolvente más pequeño.

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // equivalent to: if (x >= 0) sum += x;
}
```

hacer

Introduce un bucle do-while .

```
// Gets the next non-whitespace character from standard input
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

para

Introduce un bucle for o, en C ++ 11 y posteriores, un bucle for basado en rango .

```
// print 10 asterisks
for (int i = 0; i < 10; i++) {
```

```
    putchar('*');
}
```

mientras

Introduce un bucle while .

```
int i = 0;
// print 10 asterisks
while (i < 10) {
    putchar('*');
    i++;
}
```

rango basado en bucle

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

Lea Iteración en línea: https://riptutorial.com/es/cplusplus/topic/7841/iteracion

Capítulo 71: Iteradores

Examples

Iteradores C (Punteros)

```
// This creates an array with 5 values.
const int array[] = { 1, 2, 3, 4, 5 };

#ifdef BEFORE_CPP11

// You can use `sizeof` to determine how many elements are in an array.
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);

// Then you can iterate over the array by incrementing a pointer until
// it reaches past the end of our array.
for (const int* i = first; i < afterLast; ++i) {
    std::cout << *i << std::endl;
}

#else

// With C++11, you can let the STL compute the start and end iterators:
for (auto i = std::begin(array); i != std::end(array); ++i) {
    std::cout << *i << std::endl;
}

#endif
```

Este código emitiría los números del 1 al 5, uno en cada línea de la siguiente manera:

1
2
3
4
5

Rompiendolo

```
const int array[] = { 1, 2, 3, 4, 5 };
```

Esta línea crea una nueva matriz de enteros con 5 valores. Las matrices C son solo punteros a la memoria donde cada valor se almacena junto en un bloque contiguo.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

Estas líneas crean dos punteros. El primer puntero recibe el valor del puntero de matriz, que es la dirección del primer elemento de la matriz. El operador `sizeof` cuando se utiliza en una matriz C

devuelve el tamaño de la matriz en bytes. Dividido por el tamaño de un elemento, se obtiene el número de elementos en la matriz. Podemos usar esto para encontrar la dirección del bloque *después de* la matriz.

```
for (const int* i = first; i < afterLast; ++i) {
```

Aquí creamos un puntero que usaremos como iterador. Se inicia con la dirección del primer elemento que queremos para repetir, y vamos a seguir para recorrer todo el tiempo que `i` es menor que `afterLast` , lo que significa el tiempo que `i` está apuntando a una dirección dentro de `array` .

```
    std::cout << *i << std::endl;
```

Por último, dentro del bucle podemos acceder al valor al que apunta nuestro iterador `i` mediante la anulación de la referencia. Aquí el operador de desreferencia `*` devuelve el valor en la dirección en `i` .

Visión general

Los iteradores son posiciones

Los iteradores son un medio para navegar y operar en una secuencia de elementos y son una extensión generalizada de punteros. Conceptualmente es importante recordar que los iteradores son posiciones, no elementos. Por ejemplo, tome la siguiente secuencia:

```
A B C
```

La secuencia contiene tres elementos y cuatro posiciones.

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

Los elementos son cosas dentro de una secuencia. Las posiciones son lugares donde pueden suceder operaciones significativas a la secuencia. Por ejemplo, uno se inserta en una posición, *antes* o *después del* elemento A, no en un elemento. Incluso la eliminación de un elemento (`erase(A)`) se realiza primero encontrando su posición y luego eliminándola.

De los iteradores a los valores

Para convertir de una posición a un valor, un iterador se *elimina de referencia* :

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

Uno puede pensar en un iterador como una desreferencia al valor al que se refiere en la secuencia. Esto es especialmente útil para comprender por qué nunca debe desreferenciar el iterador `end()` en una secuencia:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           +-- An iterator here has no value. Do not dereference it!
  +-------------- An iterator here dereferences to the value A.
```

En todas las secuencias y contenedores que se encuentran en la biblioteca estándar de C ++, `begin()` devolverá un iterador a la primera posición, y `end()` devolverá un iterador a una pasada la última posición (¡ *no* la última posición!). En consecuencia, los nombres de estos iteradores en algoritmos a menudo se etiquetan `first` y `last` :

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           |
  +- first    +- last
```

También es posible obtener un iterador para *cualquier secuencia* , porque incluso una secuencia vacía contiene al menos una posición:

```
+---+
|   |
+---+
```

En una secuencia vacía, `begin()` y `end()` estarán en la misma posición, y *ninguna de ellas* puede ser referenciada:

```
+---+
|   |
+---+
  ↑
  |
  +- empty_sequence.begin()
  |
  +- empty_sequence.end()
```

La visualización alternativa de los iteradores es que marcan las posiciones *entre los* elementos:

```
+---+---+---+
| A | B | C |
+---+---+---+
↑   ^   ^   ↑
|           |
+- first    +- last
```

y la anulación de la referencia a un iterador devuelve una referencia al elemento que viene

después del iterador. Algunas situaciones donde esta vista es particularmente útil son:

- `insert` operaciones de `insert` insertarán elementos en la posición indicada por el iterador,
- `erase` operaciones de `erase` devolverán un iterador correspondiente a la misma posición que la pasada,
- un iterador y su correspondiente iterador inverso están ubicados en la misma posición entre los elementos

Iteradores inválidos

Un iterador se *invalida* si (por ejemplo, en el curso de una operación) su posición ya no forma parte de una secuencia. No se puede anular la referencia a un iterador invalidado hasta que se haya reasignado a una posición válida. Por ejemplo:

```
std::vector<int>::iterator first;
{
    std::vector<int> foo;
    first = foo.begin(); // first is now valid
} // foo falls out of scope and is destroyed
// At this point first is now invalid
```

Los muchos algoritmos y las funciones de miembro de secuencia en la biblioteca estándar de C ++ tienen reglas que rigen cuándo se invalidan los iteradores. Cada algoritmo es diferente en la forma en que tratan (e invalidan) los iteradores.

Navegando con iteradores

Como sabemos, los iteradores son para navegar por secuencias. Para hacer eso, un iterador debe migrar su posición a lo largo de la secuencia. Los iteradores pueden avanzar en la secuencia y algunos pueden avanzar hacia atrás:

```
auto first = my_vector.begin();
++first;                                          // advance the iterator 1 position
std::advance(first, 1);                           // advance the iterator 1 position
first = std::next(first);                         // returns iterator to the next element
std::advance(first, -1);                          // advance the iterator 1 position
backwards
first = std::next(first, 20);                     // returns iterator to the element 20
position forward
first = std::prev(first, 5);                      // returns iterator to the element 5
position backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two
iterators.
```

Tenga en cuenta que el segundo argumento de std :: distance debe ser accesible desde el primero (o, en otras palabras, `first` debe ser menor o igual que el `second`).

Aunque puede realizar operadores aritméticos con iteradores, no todas las operaciones están definidas para todos los tipos de iteradores. `a = b + 3;` funcionaría para los iteradores de acceso

aleatorio, pero no funcionaría para los iteradores delanteros o bidireccionales, que aún pueden avanzar en 3 posiciones con algo como `b = a; ++b; ++b; ++b;` . Por lo tanto, se recomienda utilizar funciones especiales en caso de que no esté seguro de qué tipo de iterador (por ejemplo, en una función de plantilla que acepta iterador).

Conceptos de iterador

El estándar de C ++ describe varios conceptos diferentes de iteradores. Estos se agrupan de acuerdo a cómo se comportan en las secuencias a las que se refieren. Si conoce el concepto que *modela* un iterador (se comporta como), puede estar seguro del comportamiento de ese iterador *independientemente de la secuencia a la que pertenezca* . A menudo se describen en orden de la más a la menos restrictiva (porque el siguiente concepto de iterador es un paso mejor que su predecesor):

- Iteradores de entrada: pueden ser referenciados *solo una vez* por posición. Solo puede avanzar, y solo una posición a la vez.
- Iteradores de avance: un iterador de entrada que puede ser referenciado en cualquier cantidad de veces.
- Iteradores bidireccionales: un iterador hacia adelante que también puede avanzar *hacia atrás* una posición a la vez.
- Iteradores de acceso aleatorio: un iterador bidireccional que puede avanzar o retroceder cualquier número de posiciones a la vez.
- Iteradores contiguos (desde C ++ 17): un iterador de acceso aleatorio que garantiza que los datos subyacentes son contiguos en la memoria.

Los algoritmos pueden variar según el concepto modelado por los iteradores que se les dan. Por ejemplo, aunque `random_shuffle` se puede implementar para los iteradores hacia adelante, se podría proporcionar una variante más eficiente que requiera iteradores de acceso aleatorio.

Rasgos del iterador

Los rasgos del iterador proporcionan una interfaz uniforme a las propiedades de los iteradores. Le permiten recuperar valores, diferencias, punteros, tipos de referencia y también la categoría de iterador:

```
template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val)  {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}
```

La categoría de iterador se puede utilizar para especializar algoritmos:

```
template<class BidirIt>
void test(BidirIt a, std::bidirectional_iterator_tag)  {
    std::cout << "Bidirectional iterator is used" << std::endl;
}

template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag)  {
    std::cout << "Forward iterator is used" << std::endl;
}

template<class Iter>
void test(Iter a)  {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}
```

Las categorías de iteradores son básicamente conceptos de iteradores, excepto que los iteradores contiguos no tienen su propia etiqueta, ya que se encontró que rompe el código.

Iteradores inversos

Si queremos iterar hacia atrás a través de una lista o vector, podemos usar un `reverse_iterator` . Un iterador inverso se realiza a partir de un iterador de acceso aleatorio o bidireccional que mantiene como miembro al que se puede acceder a través de `base()` .

Para iterar hacia atrás, utilice `rbegin()` y `rend()` como los iteradores para el final de la colección y el inicio de la colección, respectivamente.

Por ejemplo, para iterar hacia atrás use:

```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

Un iterador inverso se puede convertir en un iterador directo a través de la función miembro `base()` . La relación es que el iterador inverso hace referencia a un elemento más allá del iterador `base()` :

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&*r == &*(i-1)); // always true if r, (i-1) are dereferenceable
                        // and are not proxy iterators

 +---+---+---+---+---+---+---+
 |   | 1 | 2 | 3 | 4 | 5 |   |
 +---+---+---+---+---+---+---+
   ↑   ↑               ↑   ↑
   |   |               |   |
rend() |         rbegin()  end()
       |                   rbegin().base()
     begin()
     rend().base()
```

En la visualización donde los iteradores marcan posiciones entre elementos, la relación es más simple:

```
  +---+---+---+---+---+
  | 1 | 2 | 3 | 4 | 5 |
  +---+---+---+---+---+
  ↑                   ↑
  |                   |
  |                 end()
  |                 rbegin()
begin()             rbegin().base()
rend()
rend().base()
```

Iterador de vectores

`begin` devuelve un `iterator` al primer elemento en el contenedor de secuencia.

`end` devuelve un `iterator` al primer elemento más allá del final.

Si el objeto vectorial es `const` , tanto `begin` como `end` devuelven un `const_iterator` . Si desea que se `const_iterator` un `const_iterator` incluso si su vector no es `const` , puede usar `cbegin` y `cend` .

Ejemplo:

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };  //intialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Salida:

```
1 2 3 4 5
```

Iterador de mapas

Un iterador para el primer elemento en el contenedor.

Si un objeto de mapa está constante, la función devuelve un `const_iterator` . De lo contrario, devuelve un `iterator` .

```
// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
```

```
mymap['c'] = 300;

// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Salida:

a => 200
b => 100
c => 300

Iteradores de corriente

Los iteradores de flujo son útiles cuando necesitamos leer una secuencia o imprimir datos formateados desde un contenedor:

```
// Data stream. Any number of various whitespace characters will be OK.
std::istringstream istr("1\t 2     3 4");
std::vector<int> v;

// Constructing stream iterators and copying data from stream into vector.
std::copy(
    // Iterator which will read stream data as integers.
    std::istream_iterator<int>(istr),
    // Default constructor produces end-of-stream iterator.
    std::istream_iterator<int>(),
    std::back_inserter(v));

// Print vector contents.
std::copy(v.begin(), v.end(),
    //Will print values to standard output as integers delimeted by " -- ".
    std::ostream_iterator<int>(std::cout, " -- "));
```

El programa de ejemplo imprimirá `1 -- 2 -- 3 -- 4 --` en salida estándar.

Escribe tu propio iterador respaldado por generador

Un patrón común en otros idiomas es tener una función que produce un "flujo" de objetos, y poder usar un código de bucle para recorrerlo.

Podemos modelar esto en C ++ como

```
template<class T>
struct generator_iterator {
  using difference_type=std::ptrdiff_t;
  using value_type=T;
  using pointer=T*;
  using reference=T;
  using iterator_category=std::input_iterator_tag;
  std::optional<T> state;
  std::function< std::optional<T>() > operation;
  // we store the current element in "state" if we have one:
  T operator*() const {
```

```
    return *state;
  }
  // to advance, we invoke our operation.  If it returns a nullopt
  // we have reached the end:
  generator_iterator& operator++() {
    state = operation();
    return *this;
  }
  generator_iterator operator++(int) {
    auto r = *this;
    ++(*this);
    return r;
  }
  // generator iterators are only equal if they are both in the "end" state:
  friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
    if (!lhs.state && !rhs.state) return true;
    return false;
  }
  friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
    return !(lhs==rhs);
  }
  // We implicitly construct from a std::function with the right signature:
  generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
  {
    if (operation)
      state = operation();
  }
  // default all special member functions:
  generator_iterator( generator_iterator && ) =default;
  generator_iterator( generator_iterator const& ) =default;
  generator_iterator& operator=( generator_iterator && ) =default;
  generator_iterator& operator=( generator_iterator const& ) =default;
  generator_iterator() =default;
};
```

ejemplo vivo .

Almacenamos el elemento generado temprano para que podamos detectar más fácilmente si ya estamos al final.

Como la función de un iterador de generador final nunca se usa, podemos crear un rango de iteradores de generador copiando solo la `std::function` . Un iterador de generador construido por defecto se compara igual a sí mismo y con todos los demás iteradores de generador final.

Lea Iteradores en línea: https://riptutorial.com/es/cplusplus/topic/473/iteradores

Capítulo 72: La Regla De Tres, Cinco Y Cero

Examples

Regla de cinco

C ++ 11

C ++ 11 introduce dos nuevas funciones miembro especiales: el constructor de movimientos y el operador de asignación de movimientos. Por las mismas razones por las que desea seguir la Regla de tres en C ++ 03, por lo general, debe seguir la Regla de cinco en C ++ 11: si una clase requiere UNA de las cinco funciones miembro especiales y si mueve la semántica son deseados, entonces lo más probable es que requiera TODOS CINCO de ellos.

Sin embargo, tenga en cuenta que no seguir la Regla de cinco no suele considerarse un error, sino una oportunidad de optimización perdida, siempre que se siga la Regla de tres. Si no hay un constructor de movimientos o un operador de asignación de movimientos disponible cuando el compilador normalmente usaría uno, en su lugar utilizará la semántica de copia, de ser posible, lo que resultará en una operación menos eficiente debido a operaciones de copia innecesarias. Si la semántica de movimiento no se desea para una clase, entonces no es necesario declarar un constructor de movimiento u operador de asignación.

El mismo ejemplo que para la Regla de los Tres:

```
class Person
{
    char* name;
    int age;

public:
    // Destructor
    ~Person() { delete [] name; }

    // Implement Copy Semantics
    Person(Person const& other)
        : name(new char[std::strlen(other.name) + 1])
        , age(other.age)
    {
        std::strcpy(name, other.name);
    }

    Person &operator=(Person const& other)
    {
        // Use copy and swap idiom to implement assignment.
        Person copy(other);
        swap(*this, copy);
        return *this;
    }

    // Implement Move Semantics
    // Note: It is usually best to mark move operators as noexcept
    //       This allows certain optimizations in the standard library
```

```
    //       when the class is used in a container.

    Person(Person&& that) noexcept
        : name(nullptr)               // Set the state so we know it is undefined
        , age(0)
    {
        swap(*this, that);
    }

    Person& operator=(Person&& that) noexcept
    {
        swap(*this, that);
        return *this;
    }

    friend void swap(Person& lhs, Person& rhs) noexcept
    {
        std::swap(lhs.name, rhs.name);
        std::swap(lhs.age, rhs.age);
    }
};
```

Alternativamente, tanto el operador de asignación de copia como el de movimiento pueden reemplazarse con un operador de asignación única, que toma una instancia por valor en lugar de referencia o rvalue de referencia para facilitar el uso del lenguaje de copia e intercambio.

```
Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}
```

La extensión de la Regla de tres a la Regla de cinco es importante por razones de rendimiento, pero no es estrictamente necesaria en la mayoría de los casos. Agregar el constructor de copia y el operador de asignación garantiza que mover el tipo no perderá memoria (la construcción de movimientos simplemente volverá a copiarse en ese caso), pero realizará copias que la persona que llama probablemente no anticipó.

Regla de cero

C ++ 11

Podemos combinar los principios de la Regla de cinco y RAII para obtener una interfaz mucho más sencilla: la Regla de cero: cualquier recurso que deba administrarse debe ser de su propio tipo. Ese tipo tendría que seguir la Regla de los Cinco, pero todos los usuarios de ese recurso no necesitan escribir *ninguna* de las cinco funciones de miembro especiales y simplemente pueden `default` todas ellas.

Usando la clase de `Person` introducida en el ejemplo de la Regla de tres , podemos crear un objeto de administración de recursos para los `cstrings` :

```
class cstring {
private:
```

```
    char* p;

public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* other members as appropriate */
};
```

Y una vez que esto está separado, nuestra clase `Person` vuelve mucho más simple:

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* other members as appropriate */
};
```

Los miembros especiales en `Person` ni siquiera necesitan ser declarados explícitamente; el compilador los tomará de forma predeterminada o los eliminará de manera adecuada, según el contenido de la `Person` . Por lo tanto, el siguiente es también un ejemplo de la regla de cero.

```
struct Person {
    cstring name;
    int arg;
};
```

Si el `cstring` fuera un tipo de solo movimiento, con un constructor de `delete` / copia / operador de asignación, la `Person` también se movería automáticamente solo.

El término regla de cero fue introducido por R. Martinho Fernandes

Regla de tres

c ++ 03

La Regla de los Tres establece que si un tipo necesita tener un constructor de copia, operador de asignación de copia o destructor definido por el usuario, entonces debe tener *los tres* .

El motivo de la regla es que una clase que necesita cualquiera de los tres administra algún recurso (manejadores de archivos, memoria asignada dinámicamente, etc.), y los tres son necesarios para administrar ese recurso de manera consistente. Las funciones de copia tratan sobre cómo el recurso se copia entre los objetos, y el destructor destruiría el recurso, de acuerdo

con los principios de RAII .

Considere un tipo que administra un recurso de cadena:

```
class Person
{
    char* name;
    int age;

public:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
       std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};
```

Dado que el `name` fue asignado en el constructor, el destructor lo desasigna para evitar pérdidas de memoria. ¿Pero qué pasa si se copia ese objeto?

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

Primero, se construirá `p1` . Entonces `p2` se copiará de `p1` . Sin embargo, el constructor de copia generado por C ++ copiará cada componente del tipo tal como está. Lo que significa que `p1.name` y `p2.name` apuntan a la **misma** cadena.

Cuando finalice el `main` , se llamará a los destructores. El primer destructor de `p2` será llamado; se eliminará la cadena. Entonces se llamará al destructor de `p1` . Sin embargo, la cadena *ya* está *eliminada* . Llamar a `delete` en la memoria que ya se eliminó produce un comportamiento indefinido.

Para evitar esto, es necesario proporcionar un constructor de copia adecuado. Un enfoque es implementar un sistema de referencia de recuento, donde diferentes instancias de `Person` comparten la misma cadena de datos. Cada vez que se realiza una copia, se incrementa el recuento de referencias compartidas. El destructor luego disminuye el conteo de referencia, liberando solo la memoria si el conteo es cero.

O podríamos implementar semántica de valor y comportamiento de copia profunda :

```
Person(Person const& other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
```

```
}

Person &operator=(Person const& other)
{
    // Use copy and swap idiom to implement assignment
    Person copy(other);
    swap(copy);            //  assume swap() exchanges contents of *this and copy
    return *this;
}
```

La implementación del operador de asignación de copia se complica por la necesidad de liberar un búfer existente. La técnica de copia e intercambio crea un objeto temporal que contiene un nuevo búfer. Al intercambiar el contenido de `*this` y `copy` otorga la propiedad de la `copy` del búfer original. La destrucción de la `copy` , a medida que la función regresa, libera el búfer que anteriormente era propiedad de `*this` .

Protección de autoasignación

Al escribir un operador de asignación de copia, es *muy* importante que pueda trabajar en caso de autoasignación. Es decir, tiene que permitir esto:

```
SomeType t = ...;
t = t;
```

La autoasignación usualmente no ocurre de una manera tan obvia. Normalmente ocurre a través de una ruta tortuosa a través de varios sistemas de códigos, donde la ubicación de la asignación simplemente tiene dos punteros o referencias `Person` y no tiene idea de que son el mismo objeto.

Cualquier operador de asignación de copia que escriba debe poder tener esto en cuenta.

La forma típica de hacerlo es envolver toda la lógica de asignación en una condición como esta:

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //Do assignment logic.
    }
    return *this;
}
```

Nota: es importante pensar en la autoasignación y asegurarse de que su código se comporte correctamente cuando suceda. Sin embargo, la autoasignación es un caso muy raro y la optimización para evitarlo puede en realidad pesarse sobre el caso normal. Dado que el caso normal es mucho más común, el pesimismo para la autoasignación puede reducir la eficiencia de su código (así que tenga cuidado al usarlo).

Como ejemplo, la técnica normal para implementar el operador de asignación es el `copy and swap idiom` . La implementación normal de esta técnica no se molesta en probar la autoasignación (aunque la autoasignación es costosa porque se hace una copia). La razón es que la pesimización del caso normal ha demostrado ser mucho más costosa (ya que ocurre con más

frecuencia).

c ++ 11

Los operadores de asignación de movimientos también deben estar protegidos contra la autoasignación. Sin embargo, la lógica de muchos de estos operadores se basa en `std::swap`, que puede manejar el intercambio de / a la misma memoria. Entonces, si su lógica de asignación de movimientos no es más que una serie de operaciones de intercambio, entonces no necesita protección de autoasignación.

Si este no es el caso, *debe* tomar medidas similares a las anteriores.

Lea La Regla De Tres, Cinco Y Cero en línea: https://riptutorial.com/es/cplusplus/topic/1206/la-regla-de-tres--cinco-y-cero

Capítulo 73: Lambdas

Sintaxis

- [*captura por defecto* , *lista de captura*] (*lista de argumentos*) *atributos de especificación de lanzamiento* mutable -> *return-type* { *lambda-body* } // Orden de los especificadores y atributos de lambda.
- [*lista de captura*] (*lista de argumentos*) { *lambda-cuerpo* } // Definición común de lambda.
- [=] (*argument-list*) { *lambda-body* } // Captura todas las variables locales necesarias por valor.
- [&] (*argument-list*) { *lambda-body* } // Captura todas las variables locales necesarias por referencia.
- [*capture-list*] { *lambda-body* } // Se pueden omitir la lista de argumentos y los especificadores.

Parámetros

Parámetro	Detalles
captura por defecto	Especifica cómo se capturan todas las variables no listadas. Puede ser `=` (captura por valor) o `&` (captura por referencia). Si se omite, las variables no listadas son inaccesibles dentro del *cuerpo lambda* . La *captura por defecto* debe preceder a la *lista de captura* .
lista de captura	Especifica cómo las variables locales se hacen accesibles dentro del *cuerpo lambda* . Las variables sin prefijo son capturadas por valor. Las variables con el prefijo `&` son capturadas por referencia. Dentro de un método de clase, `this` se puede usar para hacer que todos sus miembros sean accesibles por referencia. Las variables no listadas son inaccesibles, a menos que la lista esté precedida por una *captura predeterminada* .
lista de argumentos	Especifica los argumentos de la función lambda.
mudable	*(Opcional)* Normalmente las variables capturadas por valor son `const` . Especificando `mutable` hace no const. Los cambios en esas variables se mantienen entre las llamadas.
especificaciones de lanzamiento	*(opcional)* Especifica el comportamiento de lanzamiento de la excepción de la función lambda. Por ejemplo: `noexcept` o `throw(std::exception)` .
atributos	*(Opcional)* Cualquier atributo para la función lambda. Por ejemplo, si el *lambda-body* siempre lanza una excepción, se puede usar `[[noreturn]]` .

Parámetro	Detalles
-> tipo de retorno	*(opcional)* Especifica el tipo de retorno de la función lambda. Se requiere cuando el compilador no puede determinar el tipo de retorno.
cuerpo lambda	Un bloque de código que contiene la implementación de la función lambda.

Observaciones

C ++ 17 (el borrador actual) introduce `constexpr` lambdas, básicamente lambdas que pueden evaluarse en tiempo de compilación. Un lambda es automáticamente `constexpr` si satisface los requisitos de `constexpr` , pero también puede especificarlo usando la palabra clave `constexpr` :

```
//Explicitly define this lambdas as constexpr
[]() constexpr {
    //Do stuff
}
```

Examples

¿Qué es una expresión lambda?

Una **expresión lambda** proporciona una forma concisa de crear objetos de funciones simples. Una expresión lambda es un prvalue cuyo objeto de resultado se llama objeto de cierre , que se comporta como un objeto de función.

El nombre 'expresión lambda' se origina en el cálculo lambda , que es un formalismo matemático inventado en la década de 1930 por Alonzo Church para investigar cuestiones sobre lógica y computabilidad. El cálculo lambda formó la base de LISP , un lenguaje de programación funcional. En comparación con el cálculo lambda y LISP, las expresiones lambda de C ++ comparten las propiedades de no tener nombre y capturan variables del contexto circundante, pero carecen de la capacidad de operar y devolver funciones.

Una expresión lambda se usa a menudo como un argumento para funciones que toman un objeto llamable. Eso puede ser más simple que crear una función nombrada, que solo se usaría cuando se pase como argumento. En tales casos, las expresiones lambda generalmente se prefieren porque permiten definir los objetos de función en línea.

Una lambda consta normalmente de tres partes: una lista de captura `[]` , una lista de parámetros opcional `()` y un cuerpo `{}` , todos los cuales pueden estar vacíos:

```
[](){}                // An empty lambda, which does and returns nothing
```

Lista de captura

`[]` es la **lista de captura** . Por defecto, no se puede acceder a las variables del ámbito adjunto

por un lambda. *Capturar* una variable lo hace accesible dentro de la lambda, ya sea como una copia o como una referencia . Las variables capturadas se convierten en parte de la lambda; en contraste con los argumentos de la función, no se tienen que pasar al llamar a lambda.

```
int a = 0;                          // Define an integer variable
auto f = []()   { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]()  { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                    //      Note: It is the responsibility of the programmer
                                    //      to ensure that a is not destroyed before the
                                    //      lambda is called.
auto b = f();                       // Call the lambda function. a is taken from the capture list
and not passed here.
```

Lista de parámetros

`()` es la **lista de parámetros** , que es casi la misma que en las funciones normales. Si la lambda no toma argumentos, estos paréntesis se pueden omitir (excepto si necesita declarar la lambda `mutable`). Estas dos lambdas son equivalentes:

```
auto call_foo  = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

C ++ 14

La lista de parámetros puede usar el tipo de marcador de posición `auto` lugar de los tipos reales. Al hacerlo, este argumento se comporta como un parámetro de plantilla de una plantilla de función. Las siguientes lambdas son equivalentes cuando desea ordenar un vector en código genérico:

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs)
{ return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Cuerpo de funcion

`{}` es el **cuerpo** , que es el mismo que en las funciones regulares.

Llamando a un lambda

El objeto de resultado de una expresión lambda es un cierre , que se puede llamar usando el `operator()` (como con otros objetos de función):

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // Prints 10

multiplier = 15;
std::out << timesFive(2); // Still prints 2*5 == 10
```

Tipo de retorno

De forma predeterminada, se deduce el tipo de retorno de una expresión lambda.

```
[](){ return true; };
```

En este caso el tipo de retorno es `bool` .

También puede especificar manualmente el tipo de retorno usando la siguiente sintaxis:

```
[]() -> bool { return true; };
```

Lambda mutable

Los objetos capturados por valor en la lambda son, por defecto, inmutables. Esto se debe a que el `operator()` del objeto de cierre generado es `const` de forma predeterminada.

```
auto func = [c = 0](){++c; std::cout << c;};  // fails to compile because ++c
                                              // tries to mutate the state of
                                              // the lambda.
```

Se puede permitir la modificación utilizando la palabra clave `mutable` , que hace que el `operator()` del objeto más cercano no sea `const` :

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

Si se usa junto con el tipo de retorno, `mutable` viene antes.

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

Un ejemplo para ilustrar la utilidad de las lambdas.

Antes de C ++ 11:

C ++ 11

```
// Generic functor used for comparison
struct islessthan
{
    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// Declare a vector
const int arr[] = { 1, 2, 3, 4, 5 };
```

```
std::vector<int> vec(arr, arr+5);

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));
```

Desde C ++ 11:

C ++ 11

```
// Declare a vector
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value <
threshold; });
```

Especificando el tipo de retorno

Para las lambdas con una sola declaración de retorno, o múltiples declaraciones de retorno cuyas expresiones son del mismo tipo, el compilador puede deducir el tipo de retorno:

```
// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
}
```

Para las lambdas con múltiples declaraciones de retorno de *diferentes* tipos, el compilador no puede deducir el tipo de devolución:

```
// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

En este caso, debe especificar el tipo de retorno explícitamente:

```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

Las reglas para esto coinciden con las reglas para `auto` deducción de `auto` tipo. Lambdas sin tipos de retorno explícitamente especificados nunca devuelve referencias, por lo que si se desea un

tipo de referencia, también debe especificarse explícitamente:

```
auto copy = [](X& x) { return x; };       // 'copy' returns an X, so copies its input
auto ref  = [](X& x) -> X& { return x; }; // 'ref' returns an X&, no copy
```

Captura por valor

Si especifica el nombre de la variable en la lista de captura, lambda la capturará por valor. Esto significa que el tipo de cierre generado para la lambda almacena una copia de la variable. Esto también requiere que el tipo de variable sea *construible por copia* :

```
int a = 0;

[a]() {
    return a;   // Ok, 'a' is captured by value
};
```

C ++ 14

```
auto p = std::unique_ptr<T>(...);

[p]() {         // Compile error; `unique_ptr` is not copy-constructible
    return p->createWidget();
};
```

A partir de C ++ 14, es posible inicializar variables en el lugar. Esto permite mover solo tipos para ser capturados en la lambda.

C ++ 14

```
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

Aunque una lambda captura variables por valor cuando se les da su nombre, tales variables no pueden modificarse dentro del cuerpo lambda por defecto. Esto se debe a que el tipo de cierre coloca el cuerpo lambda en una declaración de `operator() const` .

La `const` aplica a los accesos a las variables miembro del tipo de cierre, y las variables capturadas que son miembros del cierre (todas las apariencias de lo contrario):

```
int a = 0;

[a]() {
    a = 2;      // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

Para eliminar la `const` , debe especificar la palabra clave `mutable` en la lambda:

```
int a = 0;

[a]() mutable {
    a = 2;      // OK, 'a' can be modified
    return a;
};
```

Debido a que `a` se capturó por valor, cualquier modificación realizada al llamar a la lambda no afectará `a` . El valor de `a` se copió en la lambda cuando se construyó, por lo que la copia de `a` de la lambda está separada de la variable externa `a` .

```
int a = 5 ;
auto plus5Val = [a] (void) { return a + 5 ; } ;
auto plus5Ref = [&a] (void) {return a + 5 ; } ;

a = 7 ;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref() ;
// The result will be "7, value 10, reference 12"
```

Captura generalizada

C ++ 14

Lambdas puede capturar expresiones, en lugar de solo variables. Esto permite que las lambdas almacenen tipos de solo movimiento:

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
  p->SomeFunc();
};
```

Esto mueve la variable `p` externa a la variable de captura lambda, también llamada `p` . `lamb` ahora posee la memoria asignada por `make_unique` . Debido a que el cierre contiene un tipo que no se puede copiar, esto significa que el `lamb` no puede copiarse. Pero se puede mover:

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //legal.
```

Ahora `lamb_move` posee la memoria.

Tenga en cuenta que `std::function<>` requiere que los valores almacenados sean copiables. Puedes escribir tu propia `std::function` , que requiere solo movimiento , o simplemente puedes meter el lambda en un contenedor `shared_ptr` :

```
auto shared_lambda = [](auto&& f){
  return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))]
```

```
  (auto&&...args)->decltype(auto) {
    return (*spf)(decltype(args)(args)...);
  };
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

toma nuestro lambda de solo movimiento y rellena su estado en un puntero compartido y luego devuelve un lambda que *se* puede copiar y luego almacenar en una `std::function` o similar.

La captura generalizada utiliza `auto` deducción de tipo `auto` para el tipo de variable. Declarará estas capturas como valores por defecto, pero también pueden ser referencias:

```
int a = 0;

auto lamb = [&v = a](int add) //Note that `a` and `v` have different names
{
  v += add; //Modifies `a`
};

lamb(20); //`a` becomes 20.
```

Generalize capture no necesita capturar una variable externa en absoluto. Puede capturar una expresión arbitraria:

```
auto lamb = [p = std::make_unique<T>(...)]()
{
    p->SomeFunc();
}
```

Esto es útil para dar a los lambdas valores arbitrarios que pueden mantener y potencialmente modificar, sin tener que declararlos externamente a la lambda. Por supuesto, eso solo es útil si no tiene la intención de acceder a esas variables después de que la lambda haya completado su trabajo.

Captura por referencia

Si precede el nombre de una variable local con un `&` , entonces la variable se capturará por referencia. Conceptualmente, esto significa que el tipo de cierre de la lambda tendrá una variable de referencia, inicializada como una referencia a la variable correspondiente desde fuera del alcance de la lambda. Cualquier uso de la variable en el cuerpo lambda se referirá a la variable original:

```
// Declare variable 'a'
int a = 0;

// Declare a lambda which captures 'a' by reference
auto set = [&a]() {
    a = 1;
};

set();
```

```
assert(a == 1);
```

La palabra clave `mutable` no es necesaria, porque `a` sí mismo no es `const` .

Por supuesto, capturar por referencia significa que la lambda **no debe** escapar al alcance de las variables que captura. Por lo tanto, puede llamar a funciones que toman una función, pero no debe llamar a una función que *almacenará* el lambda más allá del alcance de sus referencias. Y no debes devolver la lambda.

Captura por defecto

De forma predeterminada, no se puede acceder a las variables locales que no se especifican explícitamente en la lista de captura desde dentro del cuerpo lambda. Sin embargo, es posible capturar implícitamente variables nombradas por el cuerpo lambda:

```
int a = 1;
int b = 2;

// Default capture by value
[=]() { return a + b; }; // OK; a and b are captured by value

// Default capture by reference
[&]() { return a + b; }; // OK; a and b are captured by reference
```

La captura explícita todavía se puede hacer junto con la captura implícita por defecto. La definición de captura explícita anulará la captura predeterminada:

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // Illegal; 'a' is capture by value, and lambda is not 'mutable'
    b = 2; // OK; 'b' is captured by reference
};
```

Lambdas genericas

c ++ 14

Las funciones Lambda pueden tomar argumentos de tipos arbitrarios. Esto permite que un lambda sea más genérico:

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

Esto se implementa en C ++ al hacer que el `operator()` de tipo de cierre `operator()` sobrecargue una función de plantilla. El siguiente tipo tiene un comportamiento equivalente al cierre lambda anterior:

```
struct _unique_lambda_type
{
  template<typename T>
  auto operator() (T x) const {return x + x;}
};
```

No todos los parámetros en un lambda genérico necesitan ser genéricos:

```
[](auto x, int y) {return x + y;}
```

Aquí, `x` se deduce en función del primer argumento de la función, mientras que `y` siempre será `int` .

Las lambdas genéricas también pueden tomar argumentos por referencia, usando las reglas habituales para `auto` y `&` . Si un parámetro genérico se toma como `auto&&` , esta es una referencia de *reenvío* al argumento pasado y no una referencia de *rvalue* :

```
auto lamb1 = [](int &&x) {return x + 5;};
auto lamb2 = [](auto &&x) {return x + 5;};
int x = 10;
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

Las funciones Lambda pueden ser variadas y perfectamente remiten sus argumentos:

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

o:

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

que solo funciona "correctamente" con variables de tipo `auto&&` .

Una razón importante para usar lambdas genéricas es para visitar la sintaxis.

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
  std::cout << e;
});
```

Aquí estamos visitando de manera polimórfica; pero en otros contextos, los nombres del tipo que estamos pasando no son interesantes:

```
mutex_wrapped<std::ostream&> os = std::cout;
os.write([&](auto&& os){
  os << "hello world\n";
});
```

Repitiendo el tipo de `std::ostream&` is noise here; Sería como tener que mencionar el tipo de variable cada vez que la uses. Aquí estamos creando un visitante, pero no polimórfico; `auto` se

utiliza por el mismo motivo por el que puede usar `auto` en un bucle `for(:)` .

Conversión a puntero de función.

Si la lista de capturas de una lambda está vacía, entonces la lambda tiene una conversión implícita a un puntero de función que toma los mismos argumentos y devuelve el mismo tipo de retorno:

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};

using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter; // implicit conversion
```

Dicha conversión también se puede hacer cumplir con el operador unario plus:

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

Llamar a este puntero de función se comporta exactamente como invocar `operator()` en la lambda. Este puntero de función no depende en modo alguno de la existencia del cierre lambda de origen. Por lo tanto, puede sobrevivir el cierre lambda.

Esta característica es principalmente útil para usar lambdas con API que tratan con punteros de función, en lugar de objetos de función C ++.

C ++ 14

La conversión a un puntero de función también es posible para las lambdas genéricas con una lista de captura vacía. Si es necesario, la deducción de argumentos de la plantilla se utilizará para seleccionar la especialización correcta.

```
auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };
using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter;  // deduces int, int
// note however that the following is ambiguous
// func_ptr sorter_func2 = +sorter;
```

Clase lambdas y captura de esta.

Una expresión lambda evaluada en la función miembro de una clase es implícitamente un amigo de esa clase:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    // definition of a member function
    void Test()
```

```
    {
        auto lamb = [](Foo &foo, int val)
        {
            // modification of a private member variable
            foo.i = val;
        };

        // lamb is allowed to access a private member, because it is a friend of Foo
        lamb(*this, 30);
    }
};
```

Tal lambda no es solo un amigo de esa clase, sino que tiene el mismo acceso que la clase en la que está declarada.

Lambdas puede capturar `this` puntero que representa la instancia de objeto en la que se activó la función externa. Esto se hace agregando `this` a la lista de captura:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture the this pointer by value
        auto lamb = [this](int val)
        {
            i = val;
        };

        lamb(30);
    }
};
```

Cuando `this` es capturado, la lambda puede utilizar nombres de los miembros de su clase que contiene como si se tratara de su clase que contiene. Así que un implícito `this->` se aplica a tales miembros.

Tenga en cuenta que `this` se captura por valor, pero no por el valor del tipo. Es capturado por el valor de `this` , que es un *puntero* . Como tal, la lambda no *posee* `this` . Si el lambda out vive el tiempo de vida del objeto que lo creó, el lambda puede dejar de ser válido.

Esto también significa que la lambda puede modificar `this` sin ser declarado `mutable` . Es el puntero el que es `const` , no el objeto al que se apunta. Es decir, a menos que la función miembro externa fuera en sí misma una función `const` .

Además, tenga en cuenta que las cláusulas de captura predeterminadas, tanto `[=]` como `[&]` , *también* capturarán `this` implícitamente. Y ambos lo capturan por el valor del puntero. De hecho, es un error especificar `this` en la lista de captura cuando se da un valor predeterminado.

C ++ 17

Lambdas puede capturar una copia de `this` objeto, creado en el momento en que se crea la lambda. Esto se hace agregando `*this` a la lista de captura:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture a copy of the object given by the this pointer
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // does not change this->i
    }
};
```

Portar funciones lambda a C ++ 03 usando functores

Las funciones de Lambda en C ++ son azúcar sintáctica que proporcionan una sintaxis muy concisa para escribir functores . Como tal, se puede obtener una funcionalidad equivalente en C ++ 03 (aunque mucho más detallado) al convertir la función lambda en un functor:

```
// Some dummy types:
struct T1 {int dummy;};
struct T2 {int dummy;};
struct R {int dummy;};

// Code using a lambda function (requires C++11)
R use_lambda(T1 val, T2 ref) {
  // Use auto because the type of the lambda is unknown.
  auto lambda = [val, &ref](int arg1, int arg2) -> R {
    /* lambda-body */
    return R();
  };
  return lambda(12, 27);
}

// The functor class (valid C++03)
// Similar to what the compiler generates for the lambda function.
class Functor {
  // Capture list.
  T1 val;
  T2& ref;

public:
  // Constructor
  inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

  // Functor body
  R operator()(int arg1, int arg2) const {
```

```
    /* lambda-body */
    return R();
  }
};

// Equivalent to use_lambda, but uses a functor (valid C++03).
R use_functor(T1 val, T2 ref) {
  Functor functor(val, ref);
  return functor(12, 27);
}

// Make this a self-contained example.
int main() {
  T1 t1;
  T2 t2;
  use_functor(t1,t2);
  use_lambda(t1,t2);
  return 0;
}
```

Si la función lambda es `mutable` entonces haga que el operador de llamadas del funtor no sea constante, es decir:

```
R operator()(int arg1, int arg2) /*non-const*/ {
  /* lambda-body */
  return R();
}
```

Lambdas recursivas

Digamos que deseamos escribir el `gcd()` de Euclid `gcd()` como un lambda. Como una función, es:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

Pero un lambda no puede ser recursivo, no tiene forma de invocarse. Un lambda no tiene nombre y usar `this` dentro del cuerpo de un lambda se refiere a un `this` capturado (asumiendo que el lambda se crea en el cuerpo de una función miembro, de lo contrario es un error). Entonces, ¿cómo resolvemos este problema?

Usa `std::function`

Podemos tener una captura lambda de una referencia a una `std::function` aún no construida:

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

Esto funciona, pero debe usarse con moderación. Es lento (estamos usando el borrado de tipo ahora en lugar de una llamada de función directa), es frágil (copiar `gcd` o devolver `gcd` se romperá ya que la lambda se refiere al objeto original), y no funcionará con lambdas genéricos.

Utilizando dos punteros inteligentes:

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>();
*gcd_self = std::make_unique<std::function<int(int, int)>>(
  [gcd_self](int a, int b){
    return b == 0 ? a : (**gcd_self)(b, a%b);
  };
};
```

Esto agrega mucha indirección (que es una sobrecarga), pero se puede copiar / devolver, y todas las copias comparten el estado. Le permite devolver el lambda y, por lo demás, es menos frágil que la solución anterior.

Usa un combinador en Y

Con la ayuda de una estructura de utilidad corta, podemos resolver todos estos problemas:

```
template <class F>
struct y_combinator {
    F f; // the lambda will be stored here

    // a forwarding operator():
    template <class... Args>
    decltype(auto) operator()(Args&&... args) const {
        // we pass ourselves to f, then the arguments.
        // the lambda should take the first argument as `auto&& recurse` or similar.
        return f(*this, std::forward<Args>(args)...);
    }
};
// helper function that deduces the type of the lambda:
template <class F>
y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
    return {std::forward<F>(f)};
}
// (Be aware that in C++17 we can do better than a `make_` function)
```

podemos implementar nuestro `gcd` como:

```
auto gcd = make_y_combinator(
  [](auto&& gcd, int a, int b){
    return b == 0 ? a : gcd(b, a%b);
  }
);
```

El `y_combinator` es un concepto del cálculo lambda que te permite tener una recursión sin poder `y_combinator` hasta que estés definido. Este es exactamente el problema que tienen las lambdas.

Crea un lambda que toma "recurse" como su primer argumento. Cuando quieres recursionar, pasas los argumentos para recursionar.

El `y_combinator` luego devuelve un objeto de función que llama a esa función con sus argumentos, pero con un objeto "recurse" adecuado (es decir, el propio `y_combinator`) como su primer argumento. Reenvía el resto de los argumentos con los que llama al `y_combinator` a la lambda

también.

En breve:

```
auto foo = make_y_combinator( [&](auto&& recurse, some arguments) {
  // write body that processes some arguments
  // when you want to recurse, call recurse(some other arguments)
});
```

y tiene recursión en un lambda sin restricciones serias ni gastos generales significativos.

Usando lambdas para desempaquetar paquetes de parámetros en línea

C ++ 14

El paquete de parámetros que se desempaqueta tradicionalmente requiere escribir una función auxiliar cada vez que quiera hacerlo.

En este ejemplo de juguete:

```
template<std::size_t...Is>
void print_indexes( std::index_sequence<Is...> ) {
  using discard=int[];
  (void)discard{0,((void)(
    std::cout << Is << '\n' // here Is is a compile-time constant.
  ),0)...};
}
template<std::size_t I>
void print_indexes_upto() {
  return print_indexes( std::make_index_sequence<I>{} );
}
```

El `print_indexes_upto` quiere crear y desempaquetar un paquete de parámetros de índices. Para hacerlo, debe llamar a una función auxiliar. Cada vez que desee desempaquetar un paquete de parámetros que haya creado, tendrá que crear una función auxiliar personalizada para hacerlo.

Esto se puede evitar con las lambdas.

Puede desempaquetar paquetes de parámetros en un conjunto de invocaciones de un lambda, como esto:

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    using discard=int[];
    (void)discard{0,(void(
      f( index<Is> )
    ),0)...};
  };
```

```
}

template<std::size_t N>
auto index_over(index_t<N> = {}) {
  return index_over( std::make_index_sequence<N>{} );
}
```

C ++ 17

Con expresiones de plegado, `index_over()` se puede simplificar para:

```
template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    ((void)(f(index<Is>)), ...);
  };
}
```

Una vez que haya hecho eso, puede usar esto para reemplazar tener que desempaquetar manualmente los paquetes de parámetros con una segunda sobrecarga en otro código, permitiéndole desempaquetar los paquetes de parámetros "en línea":

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
  using T = std::remove_reference_t<Tup>;
  using std::tuple_size;
  auto from_zero_to_N = index_over< tuple_size<T>{} >();

  from_zero_to_N(
    [&](auto i){
      using std::get;
      f( get<i>( std::forward<Tup>(tup) ) );
    }
  );
}
```

El `auto i` pasé a la lambda por `index_over` es un `std::integral_constant<std::size_t, ???>` . Esto tiene una conversión `constexpr` a `std::size_t` que no depende del estado de `this` , por lo que podemos usarlo como una constante de tiempo de compilación, como cuando lo pasamos a `std::get<i>` arriba.

Para volver al ejemplo de juguete en la parte superior, vuelva a escribirlo como:

```
template<std::size_t I>
void print_indexes_upto() {
  index_over(index<I>)([](auto i){
    std::cout << i << '\n'; // here i is a compile-time constant
  });
}
```

que es mucho más corto, y mantiene la lógica en el código que lo usa.

Ejemplo vivo para jugar.

Lea Lambdas en línea: https://riptutorial.com/es/cplusplus/topic/572/lambdas

Capítulo 74: Literales

Introducción

Tradicionalmente, un literal es una expresión que denota una constante cuyo tipo y valor son evidentes a partir de su ortografía. Por ejemplo, `42` es un literal, mientras que `x` no lo es, ya que uno debe ver su declaración para conocer su tipo y leer las líneas de código anteriores para conocer su valor.

Sin embargo, C ++ 11 también agregó literales definidos por el usuario , que no son literales en el sentido tradicional, pero se pueden usar como una abreviatura para llamadas a funciones.

Examples

cierto

Una palabra clave que denota uno de los dos valores posibles de tipo `bool` .

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

falso

Una palabra clave que denota uno de los dos valores posibles de tipo `bool` .

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

nullptr

C ++ 11

Una palabra clave que denota una constante de puntero nula. Se puede convertir a cualquier tipo de puntero o puntero a miembro, produciendo un puntero nulo del tipo resultante.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Tenga en cuenta que `nullptr` no es en sí mismo un puntero. El tipo de `nullptr` es un tipo fundamental conocido como `std::nullptr_t` .

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

esta

Dentro de una función miembro de una clase, la palabra clave de `this` es un puntero a la instancia de la clase en la que se llama a la función. `this` no se puede utilizar en una función miembro estática.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

El tipo de `this` depende de la calificación cv de la función miembro: si `X::f` es `const` , entonces el tipo de `this` dentro de `f` es `const X*` , por `this` que no se puede usar para modificar miembros de datos no estáticos desde dentro Función miembro `const` . Del mismo modo, `this` hereda la calificación `volatile` de la función en la que aparece.

C ++ 11

`this` también se puede usar en un *inicializador* de *paréntesis o igual* para un miembro de datos no estáticos.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` es un valor, por lo que no se puede asignar a.

Literal entero

Un literal entero es una expresión primaria de la forma

- decimal-literal

Es un dígito decimal distinto de cero (1, 2, 3, 4, 5, 6, 7, 8, 9), seguido de cero o más dígitos decimales (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- octal-literal

Es el dígito cero (0) seguido de cero o más dígitos octales (0, 1, 2, 3, 4, 5, 6, 7)

```
int o = 052
```

- hex-literal

Es la secuencia de caracteres 0x o la secuencia de caracteres 0X seguida de uno o más dígitos hexadecimales (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C , d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- literal binario (desde C ++ 14)

Es la secuencia de caracteres 0b o la secuencia de caracteres 0B seguida de uno o más dígitos binarios (0, 1)

```
int b = 0b101010; // C++14
```

El sufijo entero, si se proporciona, puede contener uno o ambos de los siguientes (si se proporcionan ambos, pueden aparecer en cualquier orden:

- sufijo sin signo (el carácter u o el carácter U)

```
unsigned int u_1 = 42u;
```

- sufijo largo (el carácter l o el carácter L) o el sufijo largo-largo (la secuencia de caracteres ll o la secuencia de caracteres LL) (desde C ++ 11)

Las siguientes variables también se inicializan al mismo valor:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Notas

Las letras en los literales enteros no distinguen entre mayúsculas y minúsculas: 0xDeAdBaBeU y 0XdeadBABEu representan el mismo número (una excepción es el sufijo long-long, que es ll o LL, nunca lL o Ll)

No hay literales enteros negativos. Expresiones como -1 aplican el operador menos unario al valor representado por el literal, que puede implicar conversiones de tipo implícitas.

En C antes de C99 (pero no en C ++), se permite que los valores decimales sin sufijo que no caben en long int tengan el tipo unsigned long int.

Cuando se usa en una expresión de control de #if o #elif, todas las constantes enteras con signo actúan como si tuvieran el tipo std :: intmax_t y todas las constantes enteras sin signo actúen como si tuvieran el tipo std :: uintmax_t.

Lea Literales en línea: https://riptutorial.com/es/cplusplus/topic/7836/literales

Capítulo 75: Literales definidos por el usuario

Examples

Literales definidos por el usuario con valores dobles largos.

```
#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 km = " << 3.0_km << " m\n";
    std::cout << "3 mi = " << 3.0_mi << " m\n";
    return 0;
}
```

La salida de este programa es la siguiente:

```
3 km = 3000 m
3 mi = 4828.03 m
```

Literales estándar definidos por el usuario para la duración

C ++ 14

Esos siguientes literales usuario duración se declaran en el `namespace std::literals::chrono_literals` , donde ambos `literals` y `chrono_literals` son espacios de nombres en línea . El acceso a estos operadores se puede obtener `using namespace std::literals` `using namespace std::chrono_literals` , `using namespace std::literals::chrono_literals` `using namespace std::chrono_literals` , y `using namespace std::literals::chrono_literals` .

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
    std::chrono::milliseconds t3 = 51ms;
    std::chrono::seconds t4 = 61s;
```

```
    std::chrono::minutes t5 = 88min;
    auto t6 = 2 * 0.5h;

    auto total = t1 + t2 + t3 + t4 + t5 + t6;

    std::cout.precision(13);
    std::cout << total.count() << " nanoseconds" << std::endl; // 8941051042600 nanoseconds
    std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
              << " hours" << std::endl; // 2 hours
}
```

Literales estándar definidos por el usuario para cuerdas.

C ++ 14

Los siguientes literales de usuario de cadena se declaran en el `namespace std::literals::string_literals` , donde tanto `literals` como `string_literals` son espacios de nombres en línea . El acceso a estos operadores se puede obtener `using namespace std::literals` `using namespace std::string_literals` , `using namespace std::literals::string_literals` `using namespace std::string_literals` , y `using namespace std::literals::string_literals` .

```
#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;

    std::cout << s << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
    std::cout << utf16conv.to_bytes(s16) << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
    std::cout << utf32conv.to_bytes(s32) << std::endl;

    std::wcout << ws << std::endl;
}
```

Nota:

La cadena literal puede contener `\0`

```
std::string s1 = "foo\0\0bar";  // constructor from C-string: results in "foo"s
std::string s2 = "foo\0\0bar"s; // That string contains 2 '\0' in its middle
```

Literales estándar definidos por el usuario para complejos.

C ++ 14

Los siguientes literales de usuarios complejos se declaran en el `namespace std::literals::complex_literals` , donde tanto `literals` como `complex_literals` son espacios de nombres en línea . El acceso a estos operadores se puede obtener `using namespace std::literals` `using namespace std::complex_literals` , `using namespace std::literals::complex_literals` `using namespace std::complex_literals` y `using namespace std::literals::complex_literals` .

```
#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;        // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;      // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl;   // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}
```

Literales auto-hechos definidos por el usuario para binarios

A pesar de que puedes escribir un número binario en C ++ 14 como:

```
int number =0b0001'0101; // ==21
```

Aquí viene un ejemplo famoso con una implementación hecha a sí misma para números binarios:

Nota: El programa de expansión de toda la plantilla se ejecuta en tiempo de compilación.

```
template< char FIRST, char... REST > struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "invalid binary digit" ) ;
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value  } ;
};

template<> struct binary<'0'> { enum { value = 0 } ; };
template<> struct binary<'1'> { enum { value = 1 } ; };


// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value ; }

// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value ; }

#include <iostream>

int main()
{
    std::cout  << 10101_B << ", " << 011011000111_b  << '\n' ; // prints 21, 1735
```

```
}
```

Lea Literales definidos por el usuario en línea:
https://riptutorial.com/es/cplusplus/topic/2745/literales-definidos-por-el-usuario

Capítulo 76: Manipulación de bits

Observaciones

Para usar `std::bitset` , deberá incluir el encabezado `<bitset>` .

```
#include <bitset>
```

`std::bitset` sobrecarga todas las funciones del operador para permitir el mismo uso que el manejo en estilo c de los bitsets.

Referencias

- Trucos de Bit Twiddling

Examples

Poniendo un poco

Manipulación de bits estilo C

Se puede configurar un bit utilizando el operador OR bit a bit (`|`).

```
// Bit x will be set
number |= 1LL << x;
```

Usando std :: bitset

`set(x)` o `set(x,true)` : establece el bit en la posición `x` en `1` .

```
std::bitset<5> num(std::string("01100"));
num.set(0);      // num is now 01101
num.set(2);      // num is still 01101
num.set(4,true); // num is now 11110
```

Despejando un poco

Manipulación de bits estilo C

Se puede borrar un bit usando el operador AND a bit (`&`).

```
// Bit x will be cleared
number &= ~(1LL << x);
```

Usando std :: bitset

`reset(x)` o `set(x,false)` : borra el bit en la posición `x` .

```
std::bitset<5> num(std::string("01100"));
num.reset(2);     // num is now 01000
num.reset(0);     // num is still 01000
num.set(3,false); // num is now 00000
```

Toggling un poco

Manipulación de bits estilo C

Se puede alternar un bit usando el operador XOR (`^`).

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Usando std :: bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

Revisando un poco

Manipulación de bits estilo C

El valor del bit se puede obtener desplazando el número a la derecha `x` veces y luego ejecutando el bit AND (`&`) en él:

```
(number >> x) & 1LL;  // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

La operación de desplazamiento a la derecha puede implementarse como un cambio aritmético (con signo) o como un cambio lógico (sin signo). Si el `number` en la expresión `number >> x` tiene un tipo con signo y un valor negativo, el valor resultante está definido por la implementación.

Si necesitamos el valor de ese bit directamente en el lugar, podríamos cambiar la máscara a la izquierda:

```
(number & (1LL << x));  // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

Cualquiera de los dos puede usarse como condicional, ya que todos los valores distintos de cero se consideran verdaderos.

Usando std :: bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1);  // bit_val value is set to true;
```

Cambiando el nth bit a x

Manipulación de bits estilo C

```
// Bit n will be set if x is 1 and cleared if x is 0.
number ^= (-x ^ number) & (1LL << n);
```

Usando std :: bitset

`set(n,val)` - establece el bit `n` al valor `val` .

```
std::bitset<5> num(std::string("00100"));
num.set(0,true);  // num is now 00101
num.set(2,false); // num is now 00001
```

Establecer todos los bits

Manipulación de bits estilo C

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(Vea aquí una explicación de por qué esto funciona y en realidad es el mejor enfoque).

Usando std :: bitset

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'
```

Eliminar el bit de ajuste más a la derecha

Manipulación de bits estilo C

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
unsigned"); // For c++11 and later
    return n & (n - 1);
}
```

Explicación

- si `n` es cero, tenemos `0 & 0xFF..FF` que es cero
- de lo contrario `n` puede escribirse `0bxxxxxx10..00` y `n - 1` es `0bxxxxxx011..11` , por lo que `n & (n - 1)` es `0bxxxxxx000..00` .

Set de bits de conteo

El recuento poblacional de una cadena de bits a menudo se necesita en criptografía y otras aplicaciones y el problema ha sido ampliamente estudiado.

La forma ingenua requiere una iteración por bit:

```
unsigned value = 1234;
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
  bits += value & 1;
```

Un buen truco (basado en Eliminar el bit del conjunto más a la derecha) es:

```
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (; value; ++bits)
  value &= value - 1;
```

Pasa por tantas iteraciones como bits establecidos, por lo que es bueno cuando se espera que el `value` tenga pocos bits distintos de cero.

El método fue propuesto por primera vez por Peter Wegner (en CACM 3/322 - 1960) y es bien conocido ya que aparece en *lenguaje de programación C* por Brian W. Kernighan y Dennis M. Ritchie.

Esto requiere 12 operaciones aritméticas, una de las cuales es una combinación múltiple:

```
unsigned popcount(std::uint64_t x)
{
  const std::uint64_t m1  = 0x5555555555555555;  // binary: 0101...
```

```
  const std::uint64_t m2  = 0x3333333333333333;  // binary: 00110011..
  const std::uint64_t m4  = 0x0f0f0f0f0f0f0f0f;  // binary: 0000111100001111

  x -= (x >> 1) & m1;             // put count of each 2 bits into those 2 bits
  x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
  x = (x + (x >> 4)) & m4;        // put count of each 8 bits into those 8 bits
  return (x * h01) >> 56;  // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

Este tipo de implementación tiene el mejor comportamiento en el peor de los casos (ver el peso de Hamming para más detalles).

Muchas CPU tienen una instrucción específica (como el `popcnt` de x86) y el compilador podría ofrecer una función integrada específica (**no estándar**). Por ejemplo, con g ++ hay:

```
int __builtin_popcount (unsigned x);
```

Compruebe si un entero es una potencia de 2

El `n & (n - 1)` truco (ver Eliminar el bit del conjunto más a la derecha) también es útil para determinar si un entero es una potencia de 2:

```
bool power_of_2 = n && !(n & (n - 1));
```

Tenga en cuenta que sin la primera parte de la comprobación (`n &&`), `0` se considera incorrectamente una potencia de 2.

Aplicación de manipulación de bits: letra pequeña a mayúscula

Una de las varias aplicaciones de la manipulación de bits es convertir una letra de pequeña a mayúscula o viceversa al elegir una **máscara** y una **operación de bit** adecuada. Por ejemplo, la **carta** tiene esta representación binaria `01(1)00001` mientras que su contraparte capital tiene `01(0)00001` . Difieren únicamente en el bit entre paréntesis. En este caso, convertir **una** letra de pequeña a mayúscula es básicamente establecer el bit entre paréntesis en uno. Para ello, hacemos lo siguiente:

```
/**************************************
convert small letter to captial letter.
**************************************
     a: 01100001
  mask: 11011111  <-- (0xDF)  11(0)11111
      :---------
a&mask: 01000001  <-- A letter
**************************************/
```

El código para convertir una letra a una letra es

```
#include <cstdio>
```

```
int main()
{
    char op1 = 'a';  // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c   &   0xDF = %c\n", op1, op1 & mask);

    return 0;
}
```

El resultado es

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a   &   0xDF = A
```

Lea Manipulación de bits en línea: https://riptutorial.com/es/cplusplus/topic/3016/manipulacion-de-bits

Capítulo 77: Manipuladores de corriente

Introducción

Los manipuladores son funciones auxiliares especiales que ayudan a controlar los flujos de entrada y salida utilizando el `operator >>` o el `operator <<` .

Todos ellos pueden ser incluidos por `#include <iomanip>` .

Observaciones

Los manipuladores se pueden utilizar de otra manera. Por ejemplo:

1. `os.width(n);` es igual a `os << std::setw(n);`
   `is.width(n);` igual a `is >> std::setw(n);`

2. `os.precision(n);` es igual a `os << std::setprecision(n);`
   `is.precision(n);` igual a `is >> std::setprecision(n);`

3. `os.setfill(c);` es igual a `os << std::setfill(c);`

4. `str >> std::setbase(base);` o `str << std::setbase(base);` igual a

```
str.setf(base ==  8 ? std::ios_base::oct :
            base == 10 ? std::ios_base::dec :
                base == 16 ? std::ios_base::hex :
                     std::ios_base::fmtflags(0),
         std::ios_base::basefield);
```

5. `os.setf(std::ios_base::flag);` es igual a `os << std::flag;`
   `is.setf(std::ios_base::flag);` es igual a `is >> std::flag;`
   `os.unsetf(std::ios_base::flag);` es igual a `os << std::no ## flag;`
   `is.unsetf(std::ios_base::flag);` es igual a `is >> std::no ## flag;`
   (donde ## - es el *operador de concatenación*)
   para la siguiente `flag` s: **`boolalpha`** , **`showbase`** , **`showpoint`** , **`showpos`** , **`skipws`** , **`uppercase`** .

6. **`std::ios_base::basefield`** .
   Para `flag` s: **`dec`** , **`hex`** y **`oct`** :

- `os.setf(std::ios_base::flag, std::ios_base::basefield);` es igual a `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::basefield);` es igual a `is >> std::flag;`
  (1)
- `str.unsetf(std::ios_base::flag, std::ios_base::basefield);` es igual a
  `str.setf(std::ios_base::fmtflags(0), std::ios_base::basefield);`
  (2)

7. **`std::ios_base::adjustfield`** .
   Para `flag` s: **`left`** , **`right`** e **`internal`** :

- `os.setf(std::ios_base::flag, std::ios_base::adjustfield);` es igual a `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::adjustfield);` es igual a `is >> std::flag;`
  (1)
- `str.unsetf(std::ios_base::flag, std::ios_base::adjustfield);` es igual a
  `str.setf(std::ios_base::fmtflags(0), std::ios_base::adjustfield);`
  (2)

(1) Si la bandera del campo correspondiente previamente establecido ya se ha desactivado por `unsetf` .
(2) Si se establece la `flag` .

8. **`std::ios_base::floatfield`** .

- `os.setf(std::ios_base::flag, std::ios_base::floatfield);` es igual a `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::floatfield);` es igual a `is >> std::flag;`
  Para `flag` s: **`fixed`** y **`scientific`** .
- `os.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` es igual a `os << std::defaultfloat;`
  `is.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` igual a `is >> std::defaultfloat;`

9. `str.setf(std::ios_base::fmtflags(0), std::ios_base::flag);` es igual a
   `str.unsetf(std::ios_base::flag)`
   para la `flag` s: **`basefield`** , **`adjustfield`** **`basefield`** , **`adjustfield`** **`floatfield`** .

10. `os.setf(mask)` es igual a `os << setiosflags(mask);`
    `is.setf(mask)` es igual a `is >> setiosflags(mask);`
    `os.unsetf(mask)` es igual a `os << resetiosflags(mask);`
    `is.unsetf(mask)` es igual a `is >> resetiosflags(mask);`
    Para casi todas las `mask` de tipo `std::ios_base::fmtflags` .

Examples

Manipuladores de corriente

`std::boolalpha` and `std::noboolalpha` - cambia entre la representación textual y numérica de los booleanos.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \"" << std::boolalpha << boolValue
          << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` y `std::noshowbase` - controlan si se usa el prefijo que indica la base numérica.

`std::dec` (decimal), `std::hex` (hexadecimal) y `std::oct` (octal) - se usan para cambiar la base de enteros.

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
          << std::hex << 29 << ' - '
          << std::showbase << std::oct << 29 << ' - '
          << std::noshowbase << 29  '\n';
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

Los valores predeterminados son `std::ios_base::noshowbase` y `std::ios_base::dec` .

Si desea ver más información sobre `std::istringstream` consulte el encabezado < sstream >.

`std::uppercase` y `std::nouppercase` - controlan si los caracteres en mayúsculas se utilizan en coma flotante y en la salida de enteros hexadecimales. No tiene efecto en los flujos de entrada.

```
std::cout << std::hex << std::showbase
              << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
              << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

El valor predeterminado es `std::nouppercase` .

`std::setw(n)` : cambia el ancho del siguiente campo de entrada / salida a `n` exactamente.

La propiedad de ancho `n` se restablece a `0` cuando se llaman algunas funciones (la lista completa está aquí).

```
std::cout << "no setw:" << 51 << '\n'
          << "setw(7): " << std::setw(7) << 51 << '\n'
          << "setw(7), more output: " << 13
          << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';

char* input = "Hello, world!";
char arr[10];
std::cin >> std::setw(6) >> arr;
std::cout << "Input from \"Hello, world!\" with setw(6) gave \"" << arr << "\"\n";

// Output: 51
// setw(7):      51
// setw(7), more output: 13*****67 94

// Input: Hello, world!
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

El valor predeterminado es `std::setw(0)` .

`std::left` , `std::right` y `std::internal` - modifique la posición predeterminada de los caracteres de relleno configurando `std::ios_base::adjustfield` a `std::ios_base::left` , `std::ios_base::right` y `std::ios_base::internal` correspondiente. `std::left` y `std::right` aplican a cualquier salida, `std::internal` - para entero, punto flotante y salida monetaria. No tiene efecto en los flujos de entrada.

```
#include <locale>
...

std::cout.imbue(std::locale("en_US.utf8"));

std::cout << std::left << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, false) << '\n';
// Output:
// flt: -9.87**********
// hex: 41*************
//   $: $3.67**********
// usd: USD *3.67******
// usd: $3.67

std::cout << std::internal << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
```

```
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: -**********9.87
// hex: *************41
//   $: $3.67**********
// usd: USD *******3.67
// usd: USD        3.67

std::cout << std::right << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: **********-9.87
// hex: *************41
//   $: **********$3.67
// usd: ******USD *3.67
// usd:       USD  3.67
```

El valor predeterminado es `std::left` .

`std::fixed` , `std::scientific` , `std::hexfloat` [C ++ 11] y `std::defaultfloat` [C ++ 11] - cambia el formato de entrada / salida de punto flotante.

`std::fixed` configura `std::ios_base::floatfield` to `std::ios_base::fixed` ,
`std::scientific` - to `std::ios_base::scientific` ,
`std::hexfloat` - a `std::ios_base::fixed | std::ios_base::scientific` y
`std::defaultfloat` - to `std::ios_base::fmtflags(0)` .

`fmtflags`

```
#include <sstream>
...

std::cout << '\n'
          << "The number 0.07 in fixed:      " << std::fixed << 0.01 << '\n'
          << "The number 0.07 in scientific: " << std::scientific << 0.01 << '\n'
          << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << '\n'
          << "The number 0.07 in default:    " << std::defaultfloat << 0.01 << '\n';

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';

// Output:
// The number 0.01 in fixed:      0.070000
// The number 0.01 in scientific: 7.000000e-02
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:    0.07
```

```
// Parsing 0x1P-1022 as hex gives 2.22507e-308
```

El valor predeterminado es `std::ios_base::fmtflags(0)` .

Hay un **error** en algunos compiladores que causa

```
double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0
```

`std::showpoint` y `std::noshowpoint` : controlan si el punto decimal siempre se incluye en la representación de punto flotante. No tiene efecto en los flujos de entrada.

```
std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
          << "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7
```

El valor predeterminado es `std::showpoint` .

`std::showpos` y `std::noshowpos` : controlan la visualización del signo `+` en *la* salida *no negativa* . No tiene efecto en los flujos de entrada.

```
std::cout << "With showpos: " << std::showpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n'
          << "Without showpos: " << std::noshowpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17
```

Predeterminado if `std::noshowpos` .

`std::unitbuf` , `std::nounitbuf` - controla el flujo de salida de descarga después de cada operación. No tiene efecto en el flujo de entrada. `std::unitbuf` causa enrojecimiento.

`std::setbase(base)` : establece la base numérica de la secuencia.

`std::setbase(8)` equivale a configurar `std::ios_base::basefield` a `std::ios_base::oct` ,
`std::setbase(16)` - a `std::ios_base::hex` ,
`std::setbase(10)` - a `std::ios_base::dec` .

Si la `base` es distinta de `8` , `10` o `16` entonces `std::ios_base::basefield` se establece en

`std::ios_base::fmtflags(0)` . Significa salida decimal y entrada dependiente del prefijo.

Como predeterminado `std::ios_base::basefield` es `std::ios_base::dec` luego por defecto `std::setbase(10)` .

`std::setprecision(n)` - cambia la precisión del punto flotante.

```
#include <cmath>
#include <limits>
...

typedef std::numeric_limits<long double> ld;
const long double pi = std::acos(-1.L);

std::cout << '\n'
          << "default precision (6):   pi: " << pi << '\n'
          << "                       10pi: " << 10 * pi << '\n'
          << "std::setprecision(4):  10pi: " << std::setprecision(4) << 10 * pi << '\n'
          << "                    10000pi: " << 10000 * pi << '\n'
          << "std::fixed:         10000pi: " << std::fixed << 10000 * pi << std::defaultfloat
<< '\n'
          << "std::setprecision(10):   pi: " << std::setprecision(10) << pi << '\n'
          << "max-1 radix precicion:   pi: " << std::setprecision(ld::digits - 1) << pi <<
'\n'
          << "max+1 radix precision:   pi: " << std::setprecision(ld::digits + 1) << pi <<
'\n'
          << "significant digits prec: pi: " << std::setprecision(ld::digits10) << pi << '\n';

// Output:
// default precision (6):   pi: 3.14159
//                        10pi: 31.4159
// std::setprecision(4):  10pi: 31.42
//                     10000pi: 3.142e+04
// std::fixed:         10000pi: 31415.9265
// std::setprecision(10):   pi: 3.141592654
// max-1 radix precicion:   pi:
3.1415926535897932385128089594061862044327426701784133911132812 5
// max+1 radix precision:   pi:
3.1415926535897932385128089594061862044327426701784133911132812 5
// significant digits prec: pi: 3.14159265358979324
```

El valor predeterminado es `std::setprecision(6)` .

`std::setiosflags(mask)` y `std::resetiosflags(mask)` - establece y borra las banderas especificadas en la `mask` de tipo `std::ios_base::fmtflags` .

```
#include <sstream>
...

std::istringstream in("10 010 10 010 10 010");
int num1, num2;

in >> std::oct >> num1 >> num2;
```

```
std::cout << "Parsing \"10 010\" with std::oct gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
                              std::ios_base::uppercase |
                              std::ios_base::showbase) << 42 << '\n';
// Output: OX2A
```

`std::skipws` and `std::noskipws` : controle la omisión de los espacios en blanco `std::noskipws` mediante las funciones de entrada con formato. No tiene efecto en las corrientes de salida.

```
#include <sstream>
...

char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default  behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';

std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';
// Output: Default  behavior:  c1 = a  c2 = b  c3 = c
// noskipws behavior:  c1 = a  c2 =    c3 = b
```

El valor predeterminado es `std::ios_base::skipws` .

`std::quoted(s[, delim[, escape]])` [C ++ 14]: inserta o extrae cadenas citadas con espacios incrustados.

`s` - la cadena para insertar o extraer.
`delim` : el carácter a usar como delimitador, `"` por defecto.
`escape` : el carácter a usar como el carácter de escape, `\` por defecto.

```
#include <sstream>
...

std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in     [" << in << "]\n"
          << "stored as   [" << ss.str() << "]\n";

ss >> std::quoted(out);
```

```
std::cout << "written out [" << out << "]\n";
// Output:
// read in     [String with spaces, and embedded "quotes" too]
// stored as   ["String with spaces, and embedded \"quotes\" too"]
// written out [String with spaces, and embedded "quotes" too]
```

Para más información vea el enlace de arriba.

Manipuladores de flujo de salida

`std::ends` : inserta un carácter nulo `'\0'` en el flujo de salida. Más formalmente la declaración de este manipulador parece

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

y este manipulador coloca el carácter llamando a `os.put(charT())` cuando se usa en una expresión

```
os << std::ends;
```

`std::endl` y `std::flush` ambos `out.flush()` flujo de `out` llamando a `out.flush()` . Provoca producción inmediata. Pero `std::endl` inserta el `std::endl` final de línea `'\n'` antes de vaciar.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
// Output: First line.
// Second line. Still second line.
```

`std::setfill(c)` - cambia el carácter de relleno a `c` . A menudo se utiliza con `std::setw` .

```
std::cout << "\nDefault fill: " << std::setw(10) << 79 << '\n'
          << "setfill('#'): " << std::setfill('#')
          << std::setw(10) << 42 << '\n';
// Output:
// Default fill:         79
// setfill('#'): ########79
```

`std::put_money(mon[, intl])` [C ++ 11]. En una expresión `out << std::put_money(mon, intl)` , convierte el valor monetario `mon` (del tipo `long double` o `std::basic_string`) en su representación de caracteres según lo especificado por la faceta `std::money_put` del locale actualmente imbuido en `out` Use cadenas internacionales de moneda si `intl` es `true` , use símbolos de moneda de lo contrario.

```
long double money = 123.45;
// or std::string money = "123.45";
```

```
std::cout.imbue(std::locale("en_US.utf8"));
std::cout << std::showbase << "en_US: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: en_US: $1.23 or USD  1.23

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "ru_RU: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ru_RU: 1.23 руб or 1.23 RUB

std::cout.imbue(std::locale("ja_JP.utf8"));
std::cout << "ja_JP: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ja_JP: ￥123 or JPY  123
```

`std::put_time(tmb, fmt)` [C ++ 11]: formatea y genera un valor de fecha / hora en `std::tm` acuerdo con el formato especificado `fmt` .

`tmb` : puntero a la estructura horaria del calendario `const std::tm*` según se obtiene de `localtime()` o `gmtime()` .
`fmt` : puntero a una cadena de caracteres terminada en `const CharT*` especifica el formato de conversión.

```
#include <ctime>
...

std::time_t t = std::time(nullptr);
std::tm tm = *std::localtime(&t);

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "\nru_RU: " << std::put_time(&tm, "%c %Z") << '\n';
// Possible output:
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

Para más información vea el enlace de arriba.

Manipuladores de flujo de entrada

`std::ws` : consume espacios en blanco iniciales en el flujo de entrada. Es diferente de `std::skipws` .

```
#include <sstream>
...

std::string str;
std::istringstream("  \v\n\r\t    Wow!There   is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There   is no whitespaces!
```

`std::get_money(mon[, intl])` [C ++ 11]. En una expresión `in >> std::get_money(mon, intl)` analiza la entrada del carácter como un valor monetario, según lo especificado por la faceta

`std::money_get` del entorno local imbuido actualmente `in` , y almacena el valor en `mon` (de `long double` o `std::basic_string` type). El manipulador espera las cadenas de moneda internacionales *requeridas* si `intl` es `true` , de lo contrario espera símbolos de moneda *opcionales* .

```
#include <sstream>
#include <locale>
...

std::istringstream in("$1,234.56 2.22 USD  3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
              << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD  3.33" parsed as: 123456, 222, 333
```

`std::get_time(tmb, fmt)` [C ++ 11] - analiza un valor de fecha / hora almacenado en `tmb` del formato especificado `fmt` .

`tmb` : puntero válido al objeto `const std::tm*` donde se almacenará el resultado.
`fmt` : puntero a una cadena de caracteres terminada en `const CharT*` especifica el formato de conversión.

```
#include <sstream>
#include <locale>
...

std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
else {
    std::cout << std::put_time(&t, "%c") << '\n';
}
// Possible output:
// Sun Feb 18 23:12:34 2011
```

Para más información vea el enlace de arriba.

Lea Manipuladores de corriente en línea:
https://riptutorial.com/es/cplusplus/topic/10699/manipuladores-de-corriente

Capítulo 78: Más comportamientos indefinidos en C ++

Introducción

Más ejemplos de cómo C ++ puede salir mal.

Continuación del comportamiento indefinido

Examples

Refiriéndose a los miembros no estáticos en las listas de inicializadores

Hacer referencia a miembros no estáticos en las listas de inicializadores antes de que el constructor haya comenzado a ejecutarse puede dar como resultado un comportamiento indefinido. Esto resulta ya que no todos los miembros están construidos en este momento. A partir del borrador estándar:

> §12.7.1: para un objeto con un constructor no trivial, refiriéndose a cualquier miembro o clase base no estática del objeto antes de que el constructor comience a ejecutar los resultados en un comportamiento indefinido.

Ejemplo

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
   int *p;
   X x;
   Y() : p(&x.j) { // undefined, x is not yet constructed
   }
};
```

Lea Más comportamientos indefinidos en C ++ en línea:
https://riptutorial.com/es/cplusplus/topic/9885/mas-comportamientos-indefinidos-en-c-plusplus

Capítulo 79: Mejoramiento

Introducción

Al compilar, el compilador a menudo modificará el programa para aumentar el rendimiento. Esto está permitido por la regla "como si" , que permite cualquier y todas las transformaciones que no cambian el comportamiento observable.

Examples

Expansión en línea / en línea

La expansión en línea (también conocida como en línea) es la optimización del compilador que reemplaza una llamada a una función con el cuerpo de esa función. Esto ahorra la sobrecarga de llamadas a la función, pero a costa de espacio, ya que la función puede duplicarse varias veces.

```
// source:

int process(int value)
{
    return 2 * value;
}

int foo(int a)
{
    return process(a);
}

// program, after inlining:

int foo(int a)
{
    return 2 * a; // the body of process() is copied into foo()
}
```

La alineación se realiza más comúnmente para funciones pequeñas, donde la sobrecarga de llamadas a funciones es significativa en comparación con el tamaño del cuerpo de la función.

Optimización de la base vacía

Se requiere que el tamaño de cualquier objeto o subobjeto miembro sea al menos 1, incluso si el tipo es un tipo de `class` vacío (es decir, una `class` o `struct` que no tiene miembros de datos no estáticos) para poder garantizar que Las direcciones de objetos distintos del mismo tipo son siempre distintas.

Sin embargo, `class` subobjetos de la `class` base no están tan restringidos, y pueden optimizarse completamente desde el diseño del objeto:

```
#include <cassert>
```

```
struct Base {}; // empty class

struct Derived1 : Base {
    int i;
};

int main() {
    // the size of any object of empty class type is at least 1
    assert(sizeof(Base) == 1);

    // empty base optimization applies
    assert(sizeof(Derived1) == sizeof(int));
}
```

La optimización de la base vacía se usa comúnmente en las clases de bibliotecas estándar compatibles con el asignador (`std::vector` , `std::function` , `std::shared_ptr` , etc.) para evitar ocupar cualquier almacenamiento adicional para su miembro asignador si el asignador no tiene estado. Esto se logra almacenando uno de los miembros de datos requeridos (p. Ej., Puntero de `begin` , `end` o `capacity` para el `vector`).

Referencia: cppreference

Lea Mejoramiento en línea: https://riptutorial.com/es/cplusplus/topic/9767/mejoramiento

Capítulo 80: Metaprogramacion

Introducción

En C ++, la metaprogramación se refiere al uso de macros o plantillas para generar código en tiempo de compilación.

En general, las macros están mal vistas en este rol y se prefieren las plantillas, aunque no son tan genéricas.

La metaprogramación de plantillas a menudo hace uso de cálculos en tiempo de compilación, ya sea a través de plantillas o funciones `constexpr` , para lograr sus objetivos de generación de código, sin embargo, los cálculos en tiempo de compilación no son metaprogramación en sí.

Observaciones

La metaprogramación (o más específicamente, la metaprogramación de plantillas) es la práctica de usar plantillas para crear constantes, funciones o estructuras de datos en tiempo de compilación. Esto permite que los cálculos se realicen una vez en tiempo de compilación en lugar de en cada tiempo de ejecución.

Examples

Cálculo de factoriales

Los factoriales se pueden calcular en tiempo de compilación utilizando técnicas de metaprogramación de plantillas.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

`factorial` es una estructura, pero en la metaprogramación de plantillas se trata como una metafunción de plantillas. Por convención, las metafunciones de la plantilla se evalúan verificando un miembro en particular, ya sea `::type` para las metafunciones que resultan en tipos, o `::value` para las metafunciones que generan valores.

En el código anterior, evaluamos la metafunción `factorial` mediante la creación de una instancia de la plantilla con los parámetros que queremos aprobar, y usando `::value` para obtener el resultado de la evaluación.

La metafunción en sí misma se basa en la instanciación recursiva de la misma metafunción con valores más pequeños. La especialización `factorial<0>` representa la condición de terminación. La metaprogramación de plantillas tiene la mayoría de las restricciones de un lenguaje de programación funcional , por lo que la recursión es la construcción primaria de "bucle".

Dado que las metafunciones de la plantilla se ejecutan en tiempo de compilación, sus resultados se pueden usar en contextos que requieren valores de tiempo de compilación. Por ejemplo:

```
int my_array[factorial<5>::value];
```

Las matrices automáticas deben tener un tamaño definido en tiempo de compilación. Y el resultado de una metafunción es una constante de tiempo de compilación, por lo que puede usarse aquí.

Limitación : la mayoría de los compiladores no permitirán una profundidad de recursión más allá de un límite. Por ejemplo, el compilador `g++` por defecto limita la recursividad a 256 niveles. En el caso de `g++` , el programador puede establecer la profundidad de recursión usando la `-ftemplate-depth-X` .

C ++ 11

Desde C ++ 11, la plantilla `std::integral_constant` se puede usar para este tipo de cálculo de plantilla:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial :
  std::integral_constant<long long, n * factorial<n - 1>::value> {};

template<>
struct factorial<0> :
  std::integral_constant<long long, 1> {};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

Además, `constexpr` funciones `constexpr` convierten en una alternativa más limpia.

```
#include <iostream>

constexpr long long factorial(long long n)
{
  return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
  char test[factorial(3)];
  std::cout << factorial(7) << '\n';
}
```

El cuerpo de `factorial()` se escribe como una sola declaración porque en C ++ 11 `constexpr` funciones `constexpr` solo pueden usar un subconjunto bastante limitado del lenguaje.

C ++ 14

Desde C ++ 14, se han eliminado muchas restricciones para `constexpr` funciones `constexpr` y ahora se pueden escribir mucho más convenientemente:

```
constexpr long long factorial(long long n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

O incluso:

```
constexpr long long factorial(int n)
{
  long long result = 1;
  for (int i = 1; i <= n; ++i) {
    result *= i;
  }
  return result;
}
```

C ++ 17

Como c ++ 17 se puede usar la expresión de pliegue para calcular factorial:

```
#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
```

```
    std::cout << factorial<int, 5>::value << std::endl;
}
```

Iterando sobre un paquete de parámetros

A menudo, necesitamos realizar una operación sobre cada elemento en un paquete de parámetros de plantilla variadic. Hay muchas maneras de hacer esto, y las soluciones son más fáciles de leer y escribir con C ++ 17. Supongamos que simplemente queremos imprimir cada elemento en un paquete. La solución más simple es repetir:

C ++ 11

```
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}
```

En su lugar, podríamos usar el truco de expansión, para realizar toda la transmisión en una sola función. Esto tiene la ventaja de no necesitar una segunda sobrecarga, pero tiene la desventaja de una legibilidad menor a la estelar:

C ++ 11

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Para una explicación de cómo funciona esto, vea la excelente respuesta de TC .

C ++ 17

Con C ++ 17, tenemos dos nuevas herramientas poderosas en nuestro arsenal para resolver este problema. El primero es una expresión de plegado:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

Y el segundo es `if constexpr` , que nos permite escribir nuestra solución recursiva original en una sola función:

```
template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Iterando con std :: integer_sequence

Desde C ++ 14, el estándar proporciona la plantilla de clase.

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

y una metafunción generadora para ello:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

Si bien esto es estándar en C ++ 14, esto puede implementarse utilizando las herramientas de C ++ 11.

Podemos usar esta herramienta para llamar a una función con un `std::tuple` de argumentos (estandarizados en C ++ 17 como `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...> ) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }
}

template <class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& tpl) {
    return detail::apply_impl(std::forward<F>(f),
        std::forward<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
}


// this will print 3
int f(int, char, double);
```

```
auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

Despacho de etiquetas

Una forma sencilla de seleccionar entre funciones en tiempo de compilación es enviar una función a un par de funciones sobrecargadas que toman una etiqueta como un argumento (generalmente el último). Por ejemplo, para implementar `std::advance()` , podemos enviar en la categoría de iterador:

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }
}

template <class Iter, class Distance>
void advance(Iter& it, Distance n) {
    details::advance(it, n,
            typename std::iterator_traits<Iter>::iterator_category{} );
}
```

Los argumentos `std::XY_iterator_tag` de los `details::advance` sobrecargados `details::advance` funciones `details::advance` son parámetros de función no utilizados. La implementación real no importa (en realidad está completamente vacía). Su único propósito es permitir que el compilador seleccione una sobrecarga en función de los `details::advance` clase de etiqueta `details::advance` se llama a `details::advance` .

En este ejemplo, `advance` utiliza los `iterator_traits<T>::iterator_category` metafunción que devuelve uno de los `iterator_tag` clases, dependiendo del tipo real de `Iter` . Un objeto construido por defecto del tipo `iterator_category<Iter>::type` permite al compilador seleccionar una de las diferentes sobrecargas de `details::advance` . (Es probable que este parámetro de función se optimice por completo, ya que es un objeto construido por defecto de una `struct` vacía y nunca se usa).

El envío de etiquetas puede darle un código que es mucho más fácil de leer que los equivalentes

utilizando SFINAE y `enable_if` .

Nota: mientras que C ++ 17, `if constexpr` puede simplificar la implementación del `advance` en particular, no es adecuado para implementaciones abiertas a diferencia del envío de etiquetas.

Detectar si la expresión es válida

Es posible detectar si se puede llamar a un operador o función en un tipo. Para probar si una clase tiene una sobrecarga de `std::hash` , uno puede hacer esto:

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type and std::true_type
#include <utility> // for std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};
```

C ++ 17

Desde C ++ 17, `std::void_t` puede usarse para simplificar este tipo de construcción

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type, std::true_type, std::void_t
#include <utility> // for std::declval

template<class, class = std::void_t<> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) > >
    : std::true_type
{};
```

donde `std::void_t` se define como:

```
template< class... > using void_t = void;
```

Para detectar si un operador, como el `operator<` está definido, la sintaxis es casi la misma:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
```

```
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};
```

Se pueden usar para usar `std::unordered_map<T>` si `T` tiene una sobrecarga para `std::hash` , pero de lo contrario intente usar `std::map<T>` :

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K,V>>;
```

Cálculo de la potencia con C ++ 11 (y superior)

Con C ++ 11 y los cálculos superiores en tiempo de compilación puede ser mucho más fácil. Por ejemplo, el cálculo de la potencia de un número dado en el momento de la compilación será el siguiente:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

Keyword `constexpr` es responsable de calcular la función en el tiempo de compilación, entonces y solo entonces, cuando se cumplan todos los requisitos para esto (ver más en la referencia de la palabra clave constexpr), por ejemplo, todos los argumentos deben ser conocidos en el momento de la compilación.

Nota: En C ++ 11, la función `constexpr` debe componerse solo desde una declaración de retorno.

Ventajas: Comparando esto con la forma estándar de cálculo del tiempo de compilación, este método también es útil para los cálculos de tiempo de ejecución. Esto significa que si los argumentos de la función no se conocen en el momento de la compilación (por ejemplo, el valor y la potencia se dan como entrada a través del usuario), entonces la función se ejecuta en un tiempo de compilación, por lo que no es necesario duplicar un código (ya que sería forzado en los estándares más antiguos de C ++).

P.ej

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
                                // as both arguments are known at compilation time
                                // and used for a constant expression.
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3);  // runtime calculated,
                                      // because value is known only at runtime.
}
```

C ++ 17

Otra forma de calcular la potencia en tiempo de compilación puede utilizar la expresión de plegado de la siguiente manera:

```
#include <iostream>
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};

int main() {
   std::cout << power<int, 4, 2>::value << std::endl;
}
```

Distinción manual de los tipos cuando se da cualquier tipo T

Al implementar SFINAE utilizando `std::enable_if` , a menudo es útil tener acceso a las plantillas de ayuda que determinan si un tipo `T` dado coincide con un conjunto de criterios.

Para ayudarnos con eso, el estándar ya proporciona dos tipos analógicos a `true` y `false` que son `std::true_type` y `std::false_type` .

El siguiente ejemplo muestra cómo detectar si un tipo `T` es un puntero o no, la plantilla `is_pointer` imita el comportamiento del `std::is_pointer` estándar `std::is_pointer` helper:

```
template <typename T>
struct is_pointer_: std::false_type {};

template <typename T>
struct is_pointer_<T*>: std::true_type {};

template <typename T>
struct is_pointer: is_pointer_<typename std::remove_cv<T>::type> { }
```

Hay tres pasos en el código anterior (a veces solo necesitas dos):

1. La primera declaración de `is_pointer_` es el *caso predeterminado* y se hereda de `std::false_type` . El caso *predeterminado* siempre debe heredarse de `std::false_type` ya que es análogo a una "condición `false` ".

2. La segunda declaración especializa la plantilla `is_pointer_` para el puntero `T*` sin importar qué es realmente `T` Esta versión hereda de `std::true_type` .

3. La tercera declaración (la real) simplemente elimina cualquier información innecesaria de `T` (en este caso, eliminamos `volatile` calificadores `const` y `volatile`) y luego retrocedemos a una de las dos declaraciones anteriores.

Dado que `is_pointer<T>` es una clase, para acceder a su valor necesita:

- Use `::value` , por ejemplo, `is_pointer<int>::value` - `value` es un miembro de clase estática de tipo `bool` heredado de `std::true_type` o `std::false_type` ;
- Construya un objeto de este tipo, por ejemplo, `is_pointer<int>{}` - Esto funciona porque `std::is_pointer` hereda su constructor predeterminado de `std::true_type` o `std::false_type` (que tienen `constexpr` constructores) y `std::true_type` y `std::false_type` tiene `constexpr` operadores de conversión a `bool` .

Es un buen hábito proporcionar "plantillas de ayuda auxiliar" que le permiten acceder directamente al valor:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

C ++ 17

En C ++ 17 y superior, la mayoría de las plantillas de ayuda ya proporcionan una versión `_v` , por ejemplo:

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

Si-entonces-de lo contrario

C ++ 11

El tipo `std::conditional` en el encabezado de la biblioteca estándar `<type_traits>` puede seleccionar un tipo u otro, basado en un valor booleano en tiempo de compilación:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

Esta estructura contiene un puntero a `T` si `T` es más grande que el tamaño de un puntero, o `T` si es más pequeño o igual al tamaño de un puntero. Por lo tanto, `sizeof(ValueOrPointer)` siempre será <= `sizeof(void*)` .

Generic Min / Max con cuenta de argumento variable

C ++ 11

Es posible escribir una función genérica (por ejemplo, `min`) que acepte varios tipos numéricos y un conteo de argumentos arbitrarios mediante la metaprogramación de la plantilla. Esta función declara un `min` para dos argumentos y recursivamente para más.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
```

```
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

Lea Metaprogramacion en línea: https://riptutorial.com/es/cplusplus/topic/462/metaprogramacion

Capítulo 81: Metaprogramacion aritmica

Introducción

Estos son ejemplos del uso de la metaprogramación de plantillas C ++ en el procesamiento de operaciones aritméticas en tiempo de compilación.

Examples

Cálculo de la potencia en O (log n)

Este ejemplo muestra una forma eficiente de calcular la potencia mediante la metaprogramación de plantillas.

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
   static const int value = 1;
   static_assert(base != 0, "power<0, 0> is not allowed");
};


template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

Ejemplo de uso:

```
std::cout << power<2, 9>::value;
```

C ++ 14

Este también maneja exponentes negativos:

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
    static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
    static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs
% 2>::intermediateValue;
```

```
    constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) :
intermediateValue;

};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};


template <int base>
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};


int main()
{
    std::cout << powerDouble<2,-3>::value;
}
```

Lea Metaprogramacion aritmica en línea:
https://riptutorial.com/es/cplusplus/topic/10907/metaprogramacion-aritmica

Capítulo 82: Modelo de memoria C ++ 11

Observaciones

Los diferentes subprocesos que intentan acceder a la misma ubicación de memoria participan en una *carrera de datos* si al menos una de las operaciones es una modificación (también conocida como *operación de almacenamiento*). Estas *razas de datos* causan *un comportamiento indefinido* . Para evitarlos, es necesario evitar que estos subprocesos ejecuten simultáneamente dichas operaciones en conflicto.

Las primitivas de sincronización (exclusión mutua, sección crítica y similares) pueden proteger tales accesos. El modelo de memoria introducido en C ++ 11 define dos nuevas formas portátiles de sincronizar el acceso a la memoria en un entorno de múltiples subprocesos: operaciones atómicas y cercas.

Operaciones atómicas

Ahora es posible leer y escribir en una ubicación de memoria determinada mediante el uso de *carga atómica* y operaciones de *almacenamiento atómico* . Para mayor comodidad, estos se incluyen en la clase de plantilla `std::atomic<t>` . Esta clase ajusta un valor de tipo `t` pero esta vez se *carga* y se *almacena* en el objeto atómico.

La plantilla no está disponible para todos los tipos. Los tipos disponibles son específicos de la implementación, pero esto generalmente incluye la mayoría (o todos) los tipos integrales disponibles, así como los tipos de punteros. Así que `std::atomic<unsigned>` y `std::atomic<std::vector<foo> *>` deberían estar disponibles, mientras que `std::atomic<std::pair<bool,char>>` probablemente no estará disponible.

Las operaciones atómicas tienen las siguientes propiedades:

- Todas las operaciones atómicas se pueden realizar simultáneamente desde varios subprocesos sin causar un comportamiento indefinido.
- Una *carga atómica* verá el valor inicial con el que se construyó el objeto atómico o el valor que se le escribió a través de alguna operación de *almacenamiento atómico* .
- *Las tiendas atómicas* para el mismo objeto atómico se ordenan de la misma manera en todos los hilos. Si una hebra ya ha visto el valor de alguna operación de *almacenamiento atómico* , las operaciones de *carga atómica* subsiguientes verán el mismo valor o el valor almacenado por *la* operación de *almacenamiento atómico* subsiguiente.
- *Las* operaciones *atómicas de lectura-modificación-escritura* permiten que *la carga atómica* y *el almacenamiento atómico* ocurran sin otro *almacenamiento atómico* en el medio. Por ejemplo, uno puede incrementar atómicamente un contador desde varios subprocesos, y no se perderá ningún incremento, independientemente de la contención entre los subprocesos.
- Las operaciones atómicas reciben un parámetro opcional `std::memory_order` que define qué propiedades adicionales tiene la operación con respecto a otras ubicaciones de memoria.

std :: memory_order	Sentido
`std::memory_order_relaxed`	sin restricciones adicionales
`std::memory_order_release` → `std::memory_order_acquire`	si `load-acquire` ve el valor almacenado por `store-release` entonces los almacenes *secuenciados antes* `store-release` ocurra el `store-release` antes de las cargas secuenciadas después de la *adquisición de* la *carga*
`std::memory_order_consume`	como `memory_order_acquire` pero solo para cargas dependientes
`std::memory_order_acq_rel`	Combina `load-acquire` y `store-release`
`std::memory_order_seq_cst`	consistencia secuencial

Estas etiquetas de orden de memoria permiten tres disciplinas de ordenación de memoria diferentes: *coherencia secuencial* , *relajada* y *adquisición de versión* con su *versión de consumo de* hermanos.

Consistencia secuencial

Si no se especifica ningún orden de memoria para una operación atómica, el orden se establece de manera predeterminada en *consistencia secuencial* . Este modo también se puede seleccionar explícitamente etiquetando la operación con `std::memory_order_seq_cst` .

Con este orden, ninguna operación de memoria puede cruzar la operación atómica. Todas las operaciones de memoria secuenciadas antes de la operación atómica suceden antes de la operación atómica y la operación atómica ocurre antes de todas las operaciones de memoria que se secuencian después de esta. Este modo es probablemente el más fácil de razonar, pero también conduce a la mayor penalización para el rendimiento. También evita todas las optimizaciones del compilador que de otro modo podrían intentar reordenar las operaciones después de la operación atómica.

Pedidos relajados

Lo opuesto a la *consistencia secuencial* es el ordenamiento de memoria *relajado* . Se selecciona con la etiqueta `std::memory_order_relaxed` . La operación atómica relajada no impondrá restricciones en otras operaciones de memoria. El único efecto que queda, es que la operación sigue siendo atómica.

Liberar-Adquirir pedidos

Una operación de *almacenamiento atómico* se puede etiquetar con `std::memory_order_release` y una operación de *carga atómica* se puede etiquetar con `std::memory_order_acquire` . La primera operación se llama *liberación atómica (store-release),* mientras que la segunda se llama *adquisición (carga atómica)* .

Cuando *la adquisición de carga* ve el valor escrito por un *lanzamiento de tienda,* sucede lo siguiente: todas las operaciones de almacenamiento secuenciadas antes de la *liberación de almacenamiento se* vuelven visibles (*suceden antes*) las operaciones de carga que se secuencian después de la *adquisición de carga* .

Las operaciones atómicas de lectura-modificación-escritura también pueden recibir la etiqueta acumulativa `std::memory_order_acq_rel` . Esto hace que la porción de *carga atómica* de la operación sea una *carga atómica adquirida,* mientras que la porción de *almacenamiento atómico* se convierte *en liberación de almacenamiento atómico* .

Al compilador no se le permite mover las operaciones de la tienda después de una operación de *liberación atómica de la tienda* . Tampoco está permitido mover las operaciones de carga antes de *la carga atómica de adquisición* (o *carga de consumo*).

También tenga en cuenta que no hay *liberación de carga atómica* o *adquisición de almacén atómica* . Intentar crear tales operaciones hace que sean operaciones *relajadas* .

Orden de liberación de consumo

Esta combinación es similar a *la adquisición por versión* , pero esta vez la *carga atómica* se etiqueta con `std::memory_order_consume` y se convierte en una `std::memory_order_consume` *(atómica) de consumo de carga* . Este modo es el mismo que el *de adquisición de versión,* con la única diferencia de que entre las operaciones de carga secuenciadas después de la *carga y el consumo,* solo en función del valor cargado por la *carga y el consumo* se ordenan.

Vallas

Las cercas también permiten que las operaciones de memoria se ordenen entre hilos. Una valla es una valla de liberación o adquirir una valla.

Si una valla de liberación ocurre antes de una cerca de adquisición, entonces las tiendas secuenciadas antes de la cerca de liberación son visibles para las cargas secuenciadas después de la cerca de adquisición. Para garantizar que el cerco de liberación ocurra antes del cercado de adquisición, se pueden usar otras primitivas de sincronización, incluidas las operaciones atómicas relajadas.

Examples

Necesidad de modelo de memoria

```
int x, y;
bool ready = false;

void init()
{
  x = 2;
  y = 3;
```

```
  ready = true;
}
void use()
{
  if (ready)
    std::cout << x + y;
}
```

Un subproceso llama a la función `init()` mientras que otro subproceso (o controlador de señales) llama a la función `use()` . Uno podría esperar que la función `use()` imprima `5` o no haga nada. Esto puede no ser siempre el caso por varias razones:

- La CPU puede reordenar las escrituras que ocurren en `init()` para que el código que realmente se ejecuta pueda tener el siguiente aspecto:

  ```
  void init()
  {
    ready = true;
    x = 2;
    y = 3;
  }
  ```

- La CPU puede reordenar las lecturas que ocurren en `use()` para que el código realmente ejecutado se convierta en:

  ```
  void use()
  {
    int local_x = x;
    int local_y = y;
    if (ready)
      std::cout << local_x + local_y;
  }
  ```

- Un compilador optimizado de C ++ puede decidir reordenar el programa de manera similar.

Tal reordenación no puede cambiar el comportamiento de un programa que se ejecuta en un solo hilo porque un hilo no puede intercalar las llamadas a `init()` y `use()` . Por otro lado, en una configuración de subprocesos múltiples, un subproceso puede ver parte de las escrituras realizadas por el otro subproceso en las que puede suceder que `use()` vea `ready==true` y basura en `x` o `y` o ambos.

El modelo de memoria C ++ permite que el programador especifique qué operaciones de reordenación están permitidas y cuáles no, de modo que un programa de subprocesos múltiples también podría comportarse como se espera. El ejemplo anterior se puede reescribir de forma segura para subprocesos como este:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
```

```
  ready.store(true, std::memory_order_release);
}
void use()
{
  if (ready.load(std::memory_order_acquire))
    std::cout << x + y;
}
```

Aquí `init()` realiza *una* operación de *liberación de tienda atómica* . Esto no solo almacena el valor `true` en `ready` , sino que también le dice al compilador que no puede mover esta operación antes de las operaciones de escritura que se *secuencian antes* .

La función `use()` realiza una operación de *adquisición de carga atómica* . Lee el valor actual de `ready` y también prohíbe al compilador colocar las operaciones de lectura que se *secuencian después de* que *suceda antes de* la *carga atómica* .

Estas operaciones atómicas también hacen que el compilador ponga todas las instrucciones de hardware necesarias para informar a la CPU que se abstenga de realizar reordenamientos no deseados.

Debido a que el *lanzamiento de la tienda atómica se* encuentra en la misma ubicación de memoria que la *adquisición de carga atómica* , el modelo de memoria estipula que si la operación de *adquisición de* la *carga* ve el valor escrito por la operación de *lanzamiento de* la *tienda* , entonces todas las escrituras realizadas por `init()` ' El subproceso s antes de ese *lanzamiento de tienda* será visible para las cargas que `use()` el subproceso de `use()` ejecuta después de su *adquisición de carga* . Es decir, si `use()` ve `ready==true` , entonces se garantiza que veamos `x==2` y `y==3` .

Tenga en cuenta que el compilador y la CPU todavía pueden escribir en `y` antes de escribir en `x` , y de manera similar, las lecturas de estas variables en `use()` pueden suceder en cualquier orden.

Ejemplo de valla

El ejemplo anterior también se puede implementar con cercas y operaciones atómicas relajadas:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  atomic_thread_fence(std::memory_order_release);
  ready.store(true, std::memory_order_relaxed);
}
void use()
{
  if (ready.load(std::memory_order_relaxed))
  {
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
  }
}
```

Si la operación de carga atómica ve el valor escrito por el almacén atómico, entonces el almacenamiento ocurre antes de la carga, y también lo hacen las cercas: la cerca de liberación ocurre antes de la cerca de adquisición, haciendo que las escrituras a `x` e `y` que preceden a la cerca de liberación se hagan visibles. a la instrucción `std::cout` que sigue a la valla de adquisición.

Una valla podría ser beneficiosa si puede reducir el número total de operaciones de adquisición, liberación u otras operaciones de sincronización. Por ejemplo:

```
void block_and_use()
{
  while (!ready.load(std::memory_order_relaxed))
    ;
  atomic_thread_fence(std::memory_order_acquire);
  std::cout << x + y;
}
```

La función `block_and_use()` gira hasta que la `block_and_use() ready` se establece con la ayuda de carga atómica relajada. Luego, se utiliza una sola guía de adquisición para proporcionar el orden de memoria necesario.

Lea Modelo de memoria C ++ 11 en línea: https://riptutorial.com/es/cplusplus/topic/7975/modelo-de-memoria-c-plusplus-11

Capítulo 83: Mover la semantica

Examples

Mover la semantica

Mover la semántica es una forma de mover un objeto a otro en C ++. Para ello, vaciamos el objeto antiguo y colocamos todo lo que tenía en el nuevo objeto.

Para esto, debemos entender qué es una referencia rvalue. Una referencia rvalue (`T&&` donde T es el tipo de objeto) no es muy diferente de una referencia normal (`T&` , ahora se llama referencias lvalue). Pero actúan como 2 tipos diferentes, y así, podemos hacer constructores o funciones que toman un tipo u otro, lo que será necesario cuando se trata de la semántica de movimientos.

La razón por la que necesitamos dos tipos diferentes es para especificar dos comportamientos diferentes. Los constructores de referencia de valores están relacionados con la copia, mientras que los constructores de referencia de valores están relacionados con el movimiento.

Para mover un objeto, usaremos `std::move(obj)` . Esta función devuelve una referencia de valor al objeto, por lo que podemos robar los datos de ese objeto en uno nuevo. Hay varias maneras de hacer esto que se discuten a continuación.

Es importante tener en cuenta que el uso de `std::move` crea solo una referencia rvalue. En otras palabras, la declaración `std::move(obj)` no cambia el contenido de obj, mientras que `auto obj2 = std::move(obj)` (posiblemente) sí lo hace.

Mover constructor

Digamos que tenemos este fragmento de código.

```
class A {
public:
    int a;
    int b;

    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

Para crear un constructor de copia, es decir, para hacer una función que copie un objeto y cree uno nuevo, normalmente elegiríamos la sintaxis que se muestra arriba, tendríamos un constructor para A que toma una referencia a otro objeto de tipo A, y copiaríamos el objeto manualmente dentro del método.

Alternativamente, podríamos haber escrito `A(const A &) = default;` que copia automáticamente

sobre todos los miembros, haciendo uso de su copia constructor.

Para crear un constructor de movimientos, sin embargo, tomaremos una referencia rvalue en lugar de una referencia lvalue, como aquí.

```
class Wallet {
public:
    int nrOfDollars;

    Wallet() = default; //default ctor

    Wallet(Wallet &&other) {
        this->nrOfDollars = other.nrOfDollars;
        other.nrOfDollars = 0;
    }
};
```

Tenga en cuenta que establecemos los valores antiguos en `zero` . El constructor de movimiento predeterminado (`Wallet(Wallet&&) = default;`) copia el valor de `nrOfDollars` , ya que es un POD.

Como las semánticas de movimiento están diseñadas para permitir el estado de "robo" de la instancia original, es importante considerar cómo debería verse la instancia original después de este robo. En este caso, si no cambiáramos el valor a cero, habríamos duplicado la cantidad de dólares en juego.

```
Wallet a;
a.nrOfDollars = 1;
Wallet b (std::move(a)); //calling B(B&& other);
std::cout << a.nrOfDollars << std::endl; //0
std::cout << b.nrOfDollars << std::endl; //1
```

Así hemos movido un objeto construido a partir de uno viejo.

Si bien lo anterior es un ejemplo simple, muestra lo que se pretende que haga el constructor de movimientos. Se vuelve más útil en casos más complejos, como cuando se trata de la gestión de recursos.

```
    // Manages operations involving a specified type.
    // Owns a helper on the heap, and one in its memory (presumably on the stack).
    // Both helpers are DefaultConstructible, CopyConstructible, and MoveConstructible.
    template<typename T,
             template<typename> typename HeapHelper,
             template<typename> typename StackHelper>
    class OperationsManager {
        using MyType = OperationsManager<T, HeapHelper, StackHelper>;

        HeapHelper<T>* h_helper;
        StackHelper<T> s_helper;
        // ...

      public:
        // Default constructor & Rule of Five.
        OperationsManager() : h_helper(new HeapHelper<T>) {}
        OperationsManager(const MyType& other)
```

```
          : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}
        MyType& operator=(MyType copy) {
            swap(*this, copy);
            return *this;
        }
        ~OperationsManager() {
            if (h_helper) { delete h_helper; }
        }

        // Move constructor (without swap()).
        // Takes other's HeapHelper<T>*.
        // Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move
constructor.
        // Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
        //  new helper when it's destroyed.
        OperationsManager(MyType&& other) noexcept
          : h_helper(other.h_helper),
            s_helper(std::move(other.s_helper)) {
            other.h_helper = nullptr;
        }

        // Move constructor (with swap()).
        // Places our members in the condition we want other's to be in, then switches members
        //  with other.
        // OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
        //     swap(*this, other);
        // }

        // Copy/move helper.
        friend void swap(MyType& left, MyType& right) noexcept {
            std::swap(left.h_helper, right.h_helper);
            std::swap(left.s_helper, right.s_helper);
        }
    };
```

Mover la tarea

De manera similar a cómo podemos asignar un valor a un objeto con una referencia de valor l, al copiarlo, también podemos mover los valores de un objeto a otro sin construir uno nuevo. Llamamos a esta asignación de movimiento. Movemos los valores de un objeto a otro objeto existente.

Para esto, tendremos que sobrecargar el `operator =` , no para que tome una referencia de lvalor, como en la asignación de copia, sino para que tome una referencia de rvalor.

```
class A {
    int a;
    A& operator= (A&& other) {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};
```

Esta es la sintaxis típica para definir la asignación de movimiento. Sobrecargamos el `operator =` para que podamos darle una referencia de valor y asignarlo a otro objeto.

```
A a;
a.a = 1;
A b;
b = std::move(a); //calling A& operator= (A&& other)
std::cout << a.a << std::endl; //0
std::cout << b.a << std::endl; //1
```

Por lo tanto, podemos mover asignar un objeto a otro.

Usando std :: move para reducir la complejidad de O (n²) a O (n)

C ++ 11 introdujo el lenguaje central y el soporte de biblioteca estándar para **mover** un objeto. La idea es que cuando un objeto *o* es un temporal y quiere una copia lógica, entonces es seguro que sólo los recursos de hurto *o* 's, como un buffer asignado dinámicamente, dejando *o* lógicamente vacíos, pero aún destructible y copiable.

El soporte de lenguaje principal es principalmente

- el **generador de** tipos de **referencia rvalue** `&&` , por ejemplo, `std::string&&` es una referencia rvalue a una `std::string` , lo que indica que el objeto referido es un temporal cuyos recursos solo pueden ser robados (es decir, movidos)
- soporte especial para un **constructor de movimientos** `T( T&& )` , que se supone que mueve recursos de manera eficiente del otro objeto especificado, en lugar de copiar esos recursos, y
- soporte especial para un **operador de** `auto operator=(T&&) -> T&` **asignación operador de movimiento** `auto operator=(T&&) -> T&` , que también se supone que se mueve desde la fuente.

El soporte de biblioteca estándar es principalmente la plantilla de función `std::move` del encabezado `<utility>` . Esta función produce una referencia de valor al objeto especificado, lo que indica que se puede mover desde, como si fuera un temporal.

Para un contenedor, la copia real suele ser de complejidad O (*n*), donde *n* es el número de elementos en el contenedor, mientras que mover es O (1), tiempo constante. Y para un algoritmo que copia ese contenedor lógicamente *n* veces, esto puede reducir la complejidad del O (n^2) generalmente impráctico a solo O (*n*) lineal.

En su artículo "Contenedores que nunca cambian" en el Dr. Dobbs Journal del 19 de septiembre de 2013 , Andrew Koenig presentó un ejemplo interesante de ineficiencia algorítmica al usar un estilo de programación donde las variables son inmutables después de la inicialización. Con este estilo los bucles se expresan generalmente mediante recursión. Y para algunos algoritmos como la generación de una secuencia de Collatz, la recursión requiere copiar lógicamente un contenedor:

```
// Based on an example by Andrew Koenig in his Dr. Dobbs Journal article
// "Containers That Never Change" September 19, 2013, available at
// <url: http://www.drdobbs.com/cpp/containters-that-never-change/240161543>
```

```
// Includes here, e.g. <vector>

namespace my {
    template< class Item >
    using Vector_ = /* E.g. std::vector<Item> */;

    auto concat( Vector_<int> const& v, int const x )
        -> Vector_<int>
    {
        auto result{ v };
        result.push_back( x );
        return result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            return result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            return collatz_aux( n/2, new_result );
        }
        else
        {
            return collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        return collatz_aux( n, Vector_<int>() );
    }
}  // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << '\n';
}
```

Salida:

```
42 21 64 32 16 8 4 2
```

El número de operaciones de copia de elementos debidas a la copia de los vectores es aproximadamente O (n^2), ya que es la suma 1 + 2 + 3 + ... n .

En números concretos, con los compiladores g ++ y Visual C ++, la invocación anterior de `collatz(42)` dio como resultado una secuencia de Collatz de 8 elementos y 36 operaciones de copia de elementos (8 * `collatz(42)` = 28, más algunas) en llamadas de constructor de copia vectorial.

Todas estas operaciones de copia de elementos pueden eliminarse simplemente moviendo vectores cuyos valores ya no son necesarios. Para hacer esto es necesario eliminar `const` y referencia para los argumentos de tipo vector, pasando los vectores *por valor* . La función de devoluciones ya está optimizada automáticamente. Para las llamadas donde se pasan los vectores, y no se usan más adelante en la función, simplemente aplique `std::move` para *mover* esos buffers en lugar de copiarlos:

```
using std::move;

auto concat( Vector_<int> v, int const x )
    -> Vector_<int>
{
    v.push_back( x );
    // warning: moving a local object in a return statement prevents copy elision [-
Wpessimizing-move]
    // See https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector_<int> result )
    -> Vector_<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result;      // Make absolutely sure no use of `result` after this.
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    else
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector_<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector_<int>() );
}
```

Aquí, con los compiladores g ++ y Visual C ++, el número de operaciones de copia de elementos debidas a invocaciones del constructor de copia vectorial fue exactamente 0.

El algoritmo sigue siendo necesariamente O (*n*) en la longitud de la secuencia de Collatz producida, pero esta es una mejora bastante dramática: O (n^2) → O (*n*).

Con algún soporte de lenguaje, quizás se pueda usar mover y aún expresar y aplicar la inmutabilidad de una variable *entre su inicialización y su movimiento final* , después de lo cual cualquier uso de esa variable debería ser un error. Por desgracia, a partir de C ++ 14 C ++ no es compatible con eso. Para el código sin bucle, el movimiento sin uso después de la mudanza se puede aplicar mediante una nueva declaración del nombre relevante como una `struct` incompleta, como con el `struct result;` arriba, pero esto es feo y no es probable que sea entendido por otros programadores; También los diagnósticos pueden ser bastante engañosos.

En resumen, el lenguaje de C ++ y el soporte de la biblioteca para el movimiento permiten mejoras drásticas en la complejidad del algoritmo, pero debido a su carácter incompleto, al costo de renunciar a las garantías de corrección de códigos y la claridad de código que puede proporcionar `const` .

Para completar, la clase vectorial instrumentada utilizada para medir el número de operaciones de copia de elementos debidas a invocaciones de constructor de copia:

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

    vector<Item>    items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

    Copy_tracking_vector(){}

    Copy_tracking_vector( Copy_tracking_vector const& other )
        : items_( other.items_ )
    { n_copy_ops() += items_.size(); }

    Copy_tracking_vector( Copy_tracking_vector&& other )
        : items_( move( other.items_ ) )
    {}
};
```

Uso de semántica de movimiento en contenedores

Puedes mover un contenedor en lugar de copiarlo:

```
void print(const std::vector<int>& vec) {
    for (auto&& val : vec) {
        std::cout << val << ", ";
    }
```

```
    std::cout << std::endl;
}

int main() {
    // initialize vec1 with 1, 2, 3, 4 and vec2 as an empty vector
    std::vector<int> vec1{1, 2, 3, 4};
    std::vector<int> vec2;

    // The following line will print 1, 2, 3, 4
    print(vec1);

    // The following line will print a new line
    print(vec2);

    // The vector vec2 is assigned with move assingment.
    // This will "steal" the value of vec1 without copying it.
    vec2 = std::move(vec1);

    // Here the vec1 object is in an indeterminate state, but still valid.
    // The object vec1 is not destroyed,
    // but there's is no guarantees about what it contains.

    // The following line will print 1, 2, 3, 4
    print(vec2);
}
```

Reutilizar un objeto movido

Puedes reutilizar un objeto movido:

```
void consumingFunction(std::vector<int> vec) {
    // Some operations
}

int main() {
    // initialize vec with 1, 2, 3, 4
    std::vector<int> vec{1, 2, 3, 4};

    // Send the vector by move
    consumingFunction(std::move(vec));

    // Here the vec object is in an indeterminate state.
    // Since the object is not destroyed, we can assign it a new content.
    // We will, in this case, assign an empty value to the vector,
    // making it effectively empty
    vec = {};

    // Since the vector as gained a determinate value, we can use it normally.
    vec.push_back(42);

    // Send the vector by move again.
    consumingFunction(std::move(vec));
}
```

Lea Mover la semantica en línea: https://riptutorial.com/es/cplusplus/topic/2129/mover-la-semantica

Capítulo 84: Mutex recursivo

Examples

std :: recursive_mutex

La exclusión mutua recursiva permite que el mismo hilo bloquee recursivamente un recurso, hasta un límite no especificado.

Hay muy pocas justificaciones de palabras reales para esto. Ciertas implementaciones complejas pueden necesitar llamar a una copia sobrecargada de una función sin liberar el bloqueo.

```
    std::atomic_int temp{0};
    std::recursive_mutex _mutex;

    //launch_deferred launches asynchronous tasks on the same thread id

    auto future1 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;

                    std::this_thread::sleep_for(std::chrono::seconds(3));
                    std::unique_lock<std::recursive_mutex> lock( _mutex);
                    temp=0;

                });

    auto future2 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;
                    while ( true )
                    {
                        std::this_thread::sleep_for(std::chrono::milliseconds(1));
                        std::unique_lock<std::recursive_mutex> lock( _mutex,
std::try_to_lock);
                        if ( temp < INT_MAX )
                             temp++;

                        cout << temp << endl;

                    }
                });
    future1.get();
    future2.get();
```

Lea Mutex recursivo en línea: https://riptutorial.com/es/cplusplus/topic/9929/mutex-recursivo

Capítulo 85: Mutexes

Observaciones

Es mejor usar *std :: shared_mutex* que *std :: shared_timed_mutex* .

La diferencia de rendimiento es más del doble.

Si desea utilizar RWLock, encontrará que hay dos opciones.
Es std :: shared_mutex y shared_timed_mutex.
puedes pensar que std :: shared_timed_mutex es solo la versión 'std :: shared_mutex + time method'.

Pero la implementación es totalmente diferente.

El siguiente código es la implementación de MSVC14.1 de std :: shared_mutex.

```
class shared_mutex
{
public:
typedef _Smtx_t * native_handle_type;

shared_mutex() _NOEXCEPT
    : _Myhandle(0)
    {    // default construct
    }

~shared_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock() _NOEXCEPT
    {    // lock exclusive
    _Smtx_lock_exclusive(&_Myhandle);
    }

bool try_lock() _NOEXCEPT
    {    // try to lock exclusive
    return (_Smtx_try_lock_exclusive(&_Myhandle) != 0);
    }

void unlock() _NOEXCEPT
    {    // unlock exclusive
    _Smtx_unlock_exclusive(&_Myhandle);
    }

void lock_shared() _NOEXCEPT
```

```
    {    // lock non-exclusive
    _Smtx_lock_shared(&_Myhandle);
    }

bool try_lock_shared() _NOEXCEPT
    {    // try to lock non-exclusive
    return (_Smtx_try_lock_shared(&_Myhandle) != 0);
    }

void unlock_shared() _NOEXCEPT
    {    // unlock non-exclusive
    _Smtx_unlock_shared(&_Myhandle);
    }

native_handle_type native_handle() _NOEXCEPT
    {    // get native handle
    return (&_Myhandle);
    }

shared_mutex(const shared_mutex&) = delete;
shared_mutex& operator=(const shared_mutex&) = delete;
private:
    _Smtx_t _Myhandle;
};

void __cdecl _Smtx_lock_exclusive(_Smtx_t * smtx)
{    /* lock shared mutex exclusively */
AcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_lock_shared(_Smtx_t * smtx)
{    /* lock shared mutex non-exclusively */
AcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}

int __cdecl _Smtx_try_lock_exclusive(_Smtx_t * smtx)
{    /* try to lock shared mutex exclusively */
return (TryAcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx)));
}

int __cdecl _Smtx_try_lock_shared(_Smtx_t * smtx)
{    /* try to lock shared mutex non-exclusively */
return (TryAcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx)));
}

void __cdecl _Smtx_unlock_exclusive(_Smtx_t * smtx)
{    /* unlock exclusive shared mutex */
ReleaseSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_unlock_shared(_Smtx_t * smtx)
{    /* unlock non-exclusive shared mutex */
ReleaseSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}
```

Puede ver que std :: shared_mutex está implementado en Windows Slim Reader / Write Locks (https://msdn.microsoft.com/ko-kr/library/windows/desktop/aa904937(v=vs.85).aspx)

Ahora veamos la implementación de std :: shared_timed_mutex.

El siguiente código es la implementación de MSVC14.1 de std :: shared_timed_mutex.

```
class shared_timed_mutex
{
typedef unsigned int _Read_cnt_t;
static constexpr _Read_cnt_t _Max_readers = _Read_cnt_t(-1);
public:
shared_timed_mutex() _NOEXCEPT
    : _Mymtx(), _Read_queue(), _Write_queue(),
        _Readers(0), _Writing(false)
    {    // default construct
    }

~shared_timed_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock()
    {    // lock exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing)
        _Write_queue.wait(_Lock);
    _Writing = true;
    while (0 < _Readers)
        _Read_queue.wait(_Lock);    // wait for writing, no readers
    }

bool try_lock()
    {    // try to lock exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || 0 < _Readers)
        return (false);
    else
        {    // set writing, no readers
        _Writing = true;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock for duration
    return (try_lock_until(chrono::steady_clock::now() + _Rel_time));
    }

template<class _Clock,
    class _Duration>
    bool try_lock_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock until time point
    auto _Not_writing = [this] { return (!_Writing); };
    auto _Zero_readers = [this] { return (_Readers == 0); };
    unique_lock<mutex> _Lock(_Mymtx);

    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Not_writing))
        return (false);
```

```
    _Writing = true;

    if (!_Read_queue.wait_until(_Lock, _Abs_time, _Zero_readers))
        {    // timeout, leave writing state
        _Writing = false;
        _Lock.unlock();    // unlock before notifying, for efficiency
        _Write_queue.notify_all();
        return (false);
        }

    return (true);
    }

void unlock()
    {    // unlock exclusive
        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);

        _Writing = false;
        }

    _Write_queue.notify_all();
    }

void lock_shared()
    {    // lock non-exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing || _Readers == _Max_readers)
        _Write_queue.wait(_Lock);
    ++_Readers;
    }

bool try_lock_shared()
    {    // try to lock non-exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || _Readers == _Max_readers)
        return (false);
    else
        {    // count another reader
        ++_Readers;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_shared_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock non-exclusive for relative time
    return (try_lock_shared_until(_Rel_time
        + chrono::steady_clock::now()));
    }

template<class _Time>
    bool _Try_lock_shared_until(_Time _Abs_time)
    {    // try to lock non-exclusive until absolute time
    auto _Can_acquire = [this] {
        return (!_Writing && _Readers < _Max_readers); };

    unique_lock<mutex> _Lock(_Mymtx);
```

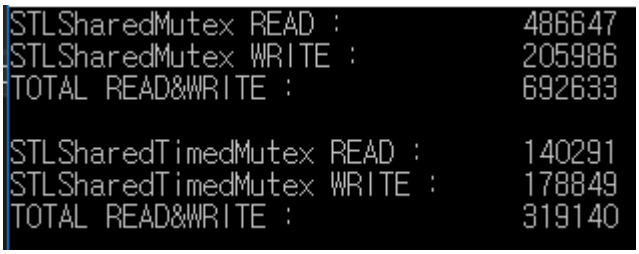
```
    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Can_acquire))
        return (false);

    ++_Readers;
    return (true);
    }

template<class _Clock,
    class _Duration>
    bool try_lock_shared_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

bool try_lock_shared_until(const xtime *_Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

void unlock_shared()
    {    // unlock non-exclusive
    _Read_cnt_t _Local_readers;
    bool _Local_writing;

        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);
        --_Readers;
        _Local_readers = _Readers;
        _Local_writing = _Writing;
        }

    if (_Local_writing && _Local_readers == 0)
        _Read_queue.notify_one();
    else if (!_Local_writing && _Local_readers == _Max_readers - 1)
        _Write_queue.notify_all();
    }

shared_timed_mutex(const shared_timed_mutex&) = delete;
shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;
private:
mutex _Mymtx;
condition_variable _Read_queue, _Write_queue;
_Read_cnt_t _Readers;
bool _Writing;
};

class stl_condition_variable_win7 final : public stl_condition_variable_interface
{
public:
    stl_condition_variable_win7()
    {
        __crtInitializeConditionVariable(&m_condition_variable);
    }

    ~stl_condition_variable_win7() = delete;
    stl_condition_variable_win7(const stl_condition_variable_win7&) = delete;
    stl_condition_variable_win7& operator=(const stl_condition_variable_win7&) = delete;

    virtual void destroy() override {}
```

```
    virtual void wait(stl_critical_section_interface *lock) override
    {
        if (!stl_condition_variable_win7::wait_for(lock, INFINITE))
            std::terminate();
    }

    virtual bool wait_for(stl_critical_section_interface *lock, unsigned int timeout) override
    {
        return __crtSleepConditionVariableSRW(&m_condition_variable,
static_cast<stl_critical_section_win7 *>(lock)->native_handle(), timeout, 0) != 0;
    }

    virtual void notify_one() override
    {
        __crtWakeConditionVariable(&m_condition_variable);
    }

    virtual void notify_all() override
    {
        __crtWakeAllConditionVariable(&m_condition_variable);
    }

private:
    CONDITION_VARIABLE m_condition_variable;
};
```

Puede ver que std :: shared_timed_mutex se implementa en std :: condition_value.

Esta es una gran diferencia.

Así que vamos a comprobar el rendimiento de dos de ellos.



Este es el resultado de la prueba de lectura / escritura de 1000 milisegundos.

std :: shared_mutex procesó lectura / escritura más de 2 veces más que std :: shared_timed_mutex.

En este ejemplo, la relación lectura / escritura es la misma, pero la velocidad de lectura es más frecuente que la velocidad de escritura en real.
Por lo tanto, la diferencia de rendimiento puede ser mayor.

El código a continuación es el código en este ejemplo.

```
void useSTLSharedMutex()
{
    std::shared_mutex shared_mtx_lock;
```

```
    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;

                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;
```

```
    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedMutex READ :           " << readProcessedCnt << std::endl;
    std::cout << "STLSharedMutex WRITE :          " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :              " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}

void useSTLSharedTimedMutex()
{
    std::shared_timed_mutex shared_mtx_lock;

    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;
```

```
                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;

    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedTimedMutex READ :      " << readProcessedCnt << std::endl;
    std::cout << "STLSharedTimedMutex WRITE :     " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :              " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}
```

Examples

std :: unique_lock, std :: shared_lock, std :: lock_guard

Se utiliza para el estilo RAII de adquisición de bloqueos de prueba, bloqueos de prueba cronometrados y bloqueos recursivos.

`std::unique_lock` permite la propiedad exclusiva de mutexes.

`std::shared_lock` permite la propiedad compartida de mutexes. Varios hilos pueden contener `std::shared_locks` en un `std::shared_mutex` . Disponible en C ++ 14.

`std::lock_guard` es una alternativa liviana a `std::unique_lock` y `std::shared_lock` .

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {
        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it =  _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
```

```
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string,std::string>  _phonebook;
};
```

Estrategias para clases de bloqueo: std :: try_to_lock, std :: adopt_lock, std :: defer_lock

Al crear un std :: unique_lock, hay tres estrategias de bloqueo diferentes para elegir: `std::try_to_lock` , `std::defer_lock` y `std::adopt_lock`

1. `std::try_to_lock` permite probar un bloqueo sin bloquear:

```
{
    std::atomic_int temp {0};
    std::mutex _mutex;

    std::thread t( [&](){

        while( temp!= -1){
            std::this_thread::sleep_for(std::chrono::seconds(5));
            std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

            if(lock.owns_lock()){
                //do something
                temp=0;
            }
        }
    });

    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
        if(lock.owns_lock()){
            if (temp < INT_MAX){
                ++temp;
            }
            std::cout << temp << std::endl;
        }
    }
}
```

2. `std::defer_lock` permite crear una estructura de bloqueo sin adquirir el bloqueo. Cuando se bloquea más de un mutex, hay una ventana de oportunidad para un interbloqueo si dos llamadores de funciones intentan adquirir los bloqueos al mismo tiempo:

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
```

```
    std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
    lock1.lock()
    lock2.lock(); // deadlock here
    std::cout << "Locked! << std::endl;
    //...
}
```

Con el siguiente código, pase lo que pase en la función, los bloqueos se adquieren y liberan en el orden apropiado:

```
   {
       std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
       std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
       std::lock(lock1,lock2); // no deadlock possible
       std::cout << "Locked! << std::endl;
       //...

   }
```

3. `std::adopt_lock` no intenta bloquear una segunda vez si el subproceso que realiza la llamada actualmente posee el bloqueo.

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
    std::cout << "Locked! << std::endl;
    //...
}
```

Algo a tener en cuenta es que std :: adopt_lock no es un sustituto para el uso de mutex recursivo. Cuando el bloqueo se sale del ámbito de aplicación, se **libera** el mutex.

std :: mutex

std :: mutex es una estructura de sincronización simple y no recursiva que se utiliza para proteger datos a los que se accede mediante varios subprocesos.

```
    std::atomic_int temp{0};
    std::mutex _mutex;

    std::thread t( [&](){

                      while( temp!= -1){
                          std::this_thread::sleep_for(std::chrono::seconds(5));
                          std::unique_lock<std::mutex> lock( _mutex);

                              temp=0;
                      }
                  });


    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
```

```
        if ( temp < INT_MAX )
            temp++;
        cout << temp << endl;

    }
```

std :: scoped_lock (C ++ 17)

`std::scoped_lock` proporciona una semántica de estilo RAII para poseer una `std::scoped_lock` más, combinada con los algoritmos de evitación de bloqueo utilizados por `std::lock` . Cuando se destruye `std::scoped_lock` , los mutex se liberan en el orden inverso al que fueron adquiridos.

```
{
    std::scoped_lock lock{_mutex1,_mutex2};
    //do something
}
```

Tipos mutex

C ++ 1x ofrece una selección de clases de exclusión mutua:

- std :: mutex - ofrece una funcionalidad de bloqueo simple.
- std :: timed_mutex - ofrece la funcionalidad try_to_lock
- std :: recursive_mutex - permite el bloqueo recursivo por el mismo hilo.
- std :: shared_mutex, std :: shared_timed_mutex - ofrece funcionalidad de bloqueo único y compartido.

std :: bloqueo

`std::lock` utiliza algoritmos de evitación de puntos muertos para bloquear uno o más mutexes. Si se lanza una excepción durante una llamada para bloquear varios objetos, `std::lock` desbloquea los objetos bloqueados con éxito antes de volver a lanzar la excepción.

```
std::lock(_mutex1, _mutex2);
```

Lea Mutexes en línea: https://riptutorial.com/es/cplusplus/topic/9895/mutexes

Capítulo 86: Objetos callables

Introducción

Los objetos recuperables son la colección de todas las estructuras de C ++ que se pueden usar como una función. En la práctica, esto es todo lo que puede pasar a la función C ++ 17 STL invoke () o que se puede usar en el constructor de la función std ::, que incluye: punteros de función, clases con operador (), clases con implícito conversiones, referencias a funciones, punteros a funciones miembros, punteros a datos de miembros, lambdas. Los objetos que se pueden llamar se utilizan en muchos algoritmos STL como predicado.

Observaciones

Una charla muy útil de Stephan T. Lavavej (<functional>: Novedades y uso apropiado) (Diapositivas) nos lleva a la base de esta documentación.

Examples

Punteros a funciones

Los punteros de función son la forma más básica de pasar funciones, que también se pueden utilizar en C. (Consulte la documentación de C para obtener más detalles).

A los efectos de los objetos que se pueden llamar, un puntero de función se puede definir como:

```
typedef returnType(*name)(arguments);                        // All
using name = returnType(*)(arguments);                       // <= C++11
using name = std::add_pointer<returnType(arguments)>::type; // <= C++11
using name = std::add_pointer_t<returnType(arguments)>;     // <= C++14
```

Si usaríamos un puntero de función para escribir nuestro propio ordenamiento vectorial, se vería así:

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // Invoke the function pointer
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}

bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // Passes the pointer to a free function
```

```
struct GreaterThanInt {
   static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // Passes the pointer to a static member
function
```

Alternativamente, podríamos haber invocado el puntero de función de una de las siguientes maneras:

- `(*lessThan)(v.front(), v.back()) // All`
- `std::invoke(lessThan, v.front(), v.back()) // <= C++17`

Clases con operador () (Functors)

Cada clase que sobrecargue al `operator()` puede usarse como un objeto de función. Estas clases pueden ser escritas a mano (a menudo conocidas como funtores) o generadas automáticamente por el compilador escribiendo Lambdas desde C ++ 11 en.

```
struct Person {
    std::string name;
    unsigned int age;
};

// Functor which find a person by name
struct FindPersonByName {
    FindPersonByName(const std::string &name) : _name(name) {}

    // Overloaded method which will get called
    bool operator()(const Person &person) const {
         return person.name == _name;
    }
private:
    std::string _name;
};

std::vector<Person> v; // Assume this contains data
std::vector<Person>::iterator iFind =
    std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...
```

Como los funtores tienen su propia identidad, no se pueden colocar en un typedef y estos deben aceptarse mediante el argumento de la plantilla. La definición de `std::find_if` puede verse como:

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
     for (Iterator i = begin, i != end, ++i)
         if (predicate(*i))
             return i;
     return end;
}
```

A partir de C ++ 17, la llamada del predicado se puede hacer con invocar: `std::invoke(predicate, *i)` .

Lea Objetos callables en línea: https://riptutorial.com/es/cplusplus/topic/6073/objetos-callables

Capítulo 87: Operadores de Bits

Observaciones

Las operaciones de cambio de bits no son portátiles en todas las arquitecturas de procesadores, diferentes procesadores pueden tener diferentes anchos de bits. En otras palabras, si escribiste

```
int a = ~0;
int b = a << 1;
```

Este valor sería diferente en una máquina de 64 bits en comparación con una máquina de 32 bits, o de un procesador basado en x86 a un procesador basado en PIC.

No es necesario tener en cuenta la endiancia para las operaciones de bits en sí mismas, es decir, el desplazamiento a la derecha (`>>`) desplazará los bits hacia el bit menos significativo y un XOR realizará una exclusiva o en los bits. Endian-ness solo se debe tener en cuenta con los datos en sí, es decir, si endian-ness es una preocupación para su aplicación, es una preocupación independientemente de las operaciones de bits.

Examples

& - a nivel de bit y

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Salida

```
a = 6, b = 10, c = 2
```

Por qué

Un poco inteligente `AND` opera en el nivel de bits y utiliza la siguiente tabla de verdad booleana:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

Cuando el valor binario para `a` (`0110`) y el valor binario para `b` (`1010`) son `AND` 'ed juntos, obtenemos el valor binario de `0010` :

```
int a = 0 1 1 0
int b = 1 0 1 0 &
        ---------
int c = 0 0 1 0
```

El bit AND AND no cambia el valor de los valores originales a menos que esté específicamente asignado para usar el operador compuesto de asignación bit a bit `&=` :

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
```

| - en modo bit o

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Salida

```
a = 5, b = 12, c = 13
```

Por qué

Un poco inteligente `OR` opera en el nivel de bits y utiliza la siguiente tabla de verdad booleana:

```
true OR true = true
true OR false = true
false OR false = false
```

Cuando el valor binario para `a` (`0101`) y el valor binario para `b` (`1100`) se unen con `OR` , obtenemos el valor binario de `1101` :

```
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
```

El OR de bits no cambia el valor de los valores originales a menos que se asigne específicamente para usar el operador compuesto de asignación de bits `|=` :

```
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
```

^ - XOR bitwise (OR exclusivo)

```
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Salida

```
a = 5, b = 9, c = 12
```

Por qué

Un poco inteligente `XOR` (exclusivo o) opera en el nivel de bits y utiliza la siguiente tabla de verdad booleana:

```
true OR true = false
true OR false = true
false OR false = false
```

Observe que con una operación XOR `true OR true = false` donde con las operaciones `true AND/OR true = true` , de ahí la naturaleza exclusiva de la operación XOR.

Usando esto, cuando el valor binario para `a` (`0101`) y el valor binario para `b` (`1001`) son `XOR` 'ed juntos obtenemos el valor binario de `1100` :

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
```

El XOR de bits no cambia el valor de los valores originales a menos que se asigne específicamente para usar el operador compuesto de asignación de bits `^=` :

```
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

El XOR de bits se puede utilizar de muchas maneras y se utiliza a menudo en operaciones de máscara de bits para el cifrado y la compresión.

Nota: el siguiente ejemplo se muestra a menudo como un ejemplo de un buen truco. Pero no se debe utilizar en el código de producción (hay mejores formas en que `std::swap()` para lograr el mismo resultado).

También puede utilizar una operación XOR para intercambiar dos variables sin un temporal:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

Para lograr esto, debe agregar un cheque para asegurarse de que pueda usarse.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
```

```
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

Entonces, aunque parece un buen truco por sí solo, no es útil en código real. xor no es una operación lógica básica, sino una combinación de otras: a ^ c = ~ (a & c) & (a | c)

también en el 2015+ las variables de los compiladores se pueden asignar como binarias:

```
int cn=0b0111;
```

~ - bitwise NOT (complemento único)

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)

std::cout << "a = " << static_cast<int>(a) <<
             ", b = " << static_cast<int>(b) << std::endl;
```

Salida

```
a = 234, b = 21
```

Por qué

Un bit sabio `NOT` (complemento único) opera en el nivel de bits y simplemente voltea cada bit. Si es un `1` , se cambia a un `0` , si es un `0` , se cambia a un `1` . El bit NO tiene el mismo efecto que XOR'ing un valor contra el valor máximo para un tipo específico:

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)
unsigned char c = a ^ ~0;
```

El bit bit NOT NO también puede ser una forma conveniente de verificar el valor máximo para un tipo integral específico:

```
unsigned int i = ~0;
unsigned char c = ~0;

std::cout << "max uint = " << i << std::endl <<
             "max uchar = " << static_cast<short>(c) << std::endl;
```

El bit NOT no cambia el valor del valor original y no tiene un operador de asignación compuesto, por lo que no puede hacer `a ~= 10` por ejemplo.

El *bit* NOT NOT (`~`) no debe confundirse con el NOT *lógico* (`!`); donde un bit sabio NO volteará cada bit, un lógico NO usará todo el valor para realizar su operación, en otras palabras `(!1) != (~1)`

<< - desplazamiento a la izquierda

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Salida

```
a = 1, b = 2
```

Por qué

El desplazamiento en el bit a la izquierda desplazará los bits del valor de la mano izquierda (`a`) el número especificado a la derecha (`1`), esencialmente rellenando los bits menos significativos con 0, de modo que se desplaza el valor de `5` (binario `0000 0101`) hacia la izquierda 4 veces (por ejemplo, `5 << 4`) producirá el valor de `80` (binario `0101 0000`). Es posible que tenga en cuenta que desplazar un valor a la izquierda 1 vez también equivale a multiplicar el valor por 2, por ejemplo:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

Pero debe tenerse en cuenta que la operación de desplazamiento a la izquierda desplazará *todos los* bits a la izquierda, incluido el bit de signo, por ejemplo:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Salida posible: `a = 2147483647, b = -2`

Si bien algunos compiladores producirán resultados que parecen esperados, se debe tener en cuenta que si deja un número con signo para cambiar de manera que el bit de signo se vea afectado, el resultado **no** está **definido** . Tampoco está **definido** si el número de bits que desea desplazar es un número negativo o es mayor que el número de bits que puede contener el tipo de la izquierda, por ejemplo:

```
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
```

El desplazamiento a la izquierda de bits no cambia el valor de los valores originales a menos que

se asigne específicamente para usar el operador compuesto de asignación de bits `<<=` :

```
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
```

>> - cambio a la derecha

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Salida

```
a = 2, b = 1
```

Por qué

El cambio de bit a la derecha desplazará los bits del valor de la mano izquierda (`a`) el número especificado a la derecha (`1`); debe tenerse en cuenta que, si bien la operación de un cambio a la derecha es estándar, lo que sucede con los bits de un cambio a la derecha en un número *negativo con signo* está *definido por la implementación* y, por lo tanto, no se puede garantizar que sea portátil, por ejemplo:

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

Tampoco está definido si el número de bits que desea desplazar es un número negativo, por ejemplo:

```
int a = 1;
int b = a >> -1;  // undefined behavior
```

El desplazamiento a la derecha de los bits no cambia el valor de los valores originales a menos que se asigne específicamente el uso del operador compuesto de asignación de bits `>>=` :

```
int a = 2;  // 0010b
a >>= 1;    // a = a >> 1;
```

Lea Operadores de Bits en línea: https://riptutorial.com/es/cplusplus/topic/2572/operadores-de-bits

Capítulo 88: Optimización en C ++

Examples

Optimización de clase base vacía

Un objeto no puede ocupar menos de 1 byte, ya que entonces los miembros de una matriz de este tipo tendrían la misma dirección. Por lo tanto, `sizeof(T)>=1` siempre se cumple. También es cierto que una clase derivada no puede ser más pequeña que *cualquiera de* sus clases base. Sin embargo, cuando la clase base está vacía, su tamaño no necesariamente se agrega a la clase derivada:

```
class Base {};

class Derived : public Base
{
public:
    int i;
};
```

En este caso, no es necesario asignar un byte para `Base` dentro de `Derived` para tener una dirección distinta por tipo por objeto. Si se realiza la optimización de la clase base vacía (y no se requiere relleno), entonces `sizeof(Derived) == sizeof(int)` , es decir, no se realiza ninguna asignación adicional para la base vacía. Esto también es posible con varias clases base (en C ++, las bases múltiples no pueden tener el mismo tipo, por lo que no surgen problemas).

Tenga en cuenta que esto solo se puede realizar si el primer miembro de `Derived` difiere en tipo de cualquiera de las clases base. Esto incluye cualquier base común directa o indirecta. Si es el mismo tipo que una de las bases (o hay una base común), se requiere al menos la asignación de un solo byte para garantizar que no haya dos objetos distintos del mismo tipo que tengan la misma dirección.

Introducción al rendimiento

C y C ++ son conocidos como lenguajes de alto rendimiento, en gran parte debido a la gran cantidad de personalización del código, lo que permite a un usuario especificar el rendimiento mediante la elección de la estructura.

Cuando se optimiza, es importante comparar el código relevante y comprender completamente cómo se utilizará el código.

Los errores comunes de optimización incluyen:

- **Optimización prematura: el** código complejo puede tener *peores resultados* después de la optimización, desperdiciando tiempo y esfuerzo. La primera prioridad debe ser escribir el código *correcto* y *mantenible* , en lugar del código optimizado.
- **Optimización para el caso de uso incorrecto:** agregar una sobrecarga para el 1% podría

no valer la pena para el otro 99%
- **Microoptimización: los** compiladores hacen esto muy eficientemente y la microoptimización puede incluso afectar la capacidad de los compiladores para optimizar aún más el código.

Los objetivos típicos de optimización son:

- Hacer menos trabajo
- Usar algoritmos / estructuras más eficientes.
- Para hacer un mejor uso del hardware.

El código optimizado puede tener efectos secundarios negativos, que incluyen:

- Mayor uso de memoria
- Código complejo: ser difícil de leer o mantener
- API comprometida y diseño de código

Optimizando ejecutando menos código.

El enfoque más sencillo para optimizar es ejecutando menos código. Este enfoque generalmente proporciona una aceleración fija sin cambiar la complejidad de tiempo del código.

A pesar de que este enfoque le proporciona una clara aceleración, esto solo proporcionará mejoras notables cuando el código se llame mucho.

Eliminando código inútil

```
void func(const A *a); // Some random function

// useless memory allocation + deallocation for the instance
auto a1 = std::make_unique<A>();
func(a1.get());

// making use of a stack object prevents
auto a2 = A{};
func(&a2);
```

C ++ 14

Desde C ++ 14, los compiladores pueden optimizar este código para eliminar la asignación y la desasignación correspondiente.

Haciendo código solo una vez

```
std::map<std::string, std::unique_ptr<A>> lookup;
// Slow insertion/lookup
// Within this function, we will traverse twice through the map lookup an element
// and even a thirth time when it wasn't in
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
```

```
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}

// Within this function, we will have the same noticeable effect as the slow variant while
going at double speed as we only traverse once through the code
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

Se puede utilizar un enfoque similar a esta optimización para implementar una versión estable de dispositivos `unique`

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // As insert returns if the insertion was successful, we can deduce if the element was
already in or not
        // This prevents an insertion, which will traverse through the map for every unique
element
        // As a result we can almost gain 50% if v would not contain any duplicates
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Evitar la reasignación inútil y copiar / mover

En el ejemplo anterior, ya evitamos las búsquedas en std :: set, sin embargo, `std::vector` todavía contiene un algoritmo creciente, en el que tendrá que reasignar su almacenamiento. Esto se puede prevenir reservando primero para el tamaño correcto.

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // By reserving 'result', we can ensure that no copying or moving will be done in the
vector
    // as it will have capacity for the maximum number of elements we will be inserting
    // If we make the assumption that no allocation occurs for size zero
    // and allocating a large block of memory takes the same time as a small block of memory
    // this will never slow down the program
    // Side note: Compilers can even predict this and remove the checks the growing from the
generated code
    result.reserve(v.size());
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See example above
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Usando contenedores eficientes

La optimización mediante el uso de las estructuras de datos correctas en el momento adecuado puede cambiar la complejidad del tiempo del código.

```
// This variant of stableUnique contains a complexity of N log(N)
// N > number of elements in v
// log(N) > insert complexity of std::set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Al usar un contenedor que usa una implementación diferente para almacenar sus elementos (hash container en lugar de tree), podemos transformar nuestra implementación en la complejidad N. Como efecto secundario, llamaremos al operador de comparación para std :: string less, ya que solo debe llamarse cuando la cadena insertada debe terminar en el mismo cubo.

```
// This variant of stableUnique contains a complexity of N
// N > number of elements in v
// 1 > insert complexity of std::unordered_set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Optimización de objetos pequeños

La optimización de objetos pequeños es una técnica que se utiliza en estructuras de datos de bajo nivel, por ejemplo, `std::string` (a veces denominada optimización de cadena pequeña / corta). Está destinado a usar el espacio de pila como un búfer en lugar de alguna memoria asignada en caso de que el contenido sea lo suficientemente pequeño como para caber dentro del espacio reservado.

Al agregar una sobrecarga de memoria adicional y cálculos adicionales, intenta evitar una asignación de pila costosa. Los beneficios de esta técnica dependen del uso y pueden incluso afectar el rendimiento si se usan incorrectamente.

Ejemplo

Una forma muy ingenua de implementar una cadena con esta optimización sería lo siguiente:

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};                       ///< Remember if we allocated memory
    char *_buffer{nullptr};                         ///< Pointer to the buffer we are using
    char _smallBuffer[SMALL_BUFFER_SIZE]= {'\0'};   ///< Stack space used for SMALL OBJECT
OPTIMIZATION

public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
       : _isAllocated(rhs._isAllocated)
       , _buffer(rhs._buffer)
       , _smallBuffer(rhs._smallBuffer) //< Not needed if allocated
    {
        if (_isAllocated)
        {
           // Prevent double deletion of the memory
           rhs._buffer = nullptr;
        }
        else
        {
            // Copy over data
            std::strcpy(_smallBuffer, rhs._smallBuffer);
            _buffer = &_smallBuffer[0];
        }
    }
    // Other methods, including other constructors, copy constructor,
    // assignment operators have been omitted for readability
};
```

Como puede ver en el código anterior, se ha agregado cierta complejidad adicional para evitar algunas operaciones `new` y de `delete` . Además de esto, la clase tiene una huella de memoria mayor que no se puede usar, excepto en un par de casos.

A menudo se intenta codificar el valor bool `_isAllocated` , dentro del puntero `_buffer` con la manipulación de bits para reducir el tamaño de una instancia (Intel 64 bit: podría reducir el tamaño

en 8 bytes). Una optimización que solo es posible cuando se conoce cuáles son las reglas de alineación de la plataforma.

¿Cuándo usar?

Como esta optimización agrega mucha complejidad, no se recomienda usar esta optimización en todas las clases. A menudo se encontrará en estructuras de datos de bajo nivel de uso común. En las implementaciones comunes de `standard library` C ++ 11 se pueden encontrar usos en `std::basic_string<>` y `std::function<>` .

Como esta optimización solo evita las asignaciones de memoria cuando los datos almacenados son más pequeños que el búfer, solo dará beneficios si la clase se usa a menudo con datos pequeños.

Un inconveniente final de esta optimización es que se requiere un esfuerzo adicional al mover el búfer, lo que hace que la operación de movimiento sea más costosa que cuando no se usaría el búfer. Esto es especialmente cierto cuando el búfer contiene un tipo que no es POD.

Lea Optimización en C ++ en línea: https://riptutorial.com/es/cplusplus/topic/4474/optimizacion-en-c-plusplus

Capítulo 89: Palabra clave amigo

Introducción

Las clases bien diseñadas encapsulan su funcionalidad, ocultan su implementación y proporcionan una interfaz limpia y documentada. Esto permite un nuevo diseño o cambio siempre que la interfaz no se modifique.

En un escenario más complejo, pueden requerirse múltiples clases que dependen de los detalles de implementación de cada uno. Las clases y funciones de Friend permiten a estos compañeros acceder a los detalles de los demás, sin comprometer la encapsulación y el ocultamiento de la información de la interfaz documentada.

Examples

Función de amigo

Una clase o una estructura puede declarar cualquier función que sea amiga. Si una función es un amigo de una clase, puede acceder a todos sus miembros protegidos y privados:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

Los modificadores de acceso no alteran la semántica de los amigos. Las declaraciones públicas, protegidas y privadas de un amigo son equivalentes.

Las declaraciones de amigos no se heredan. Por ejemplo, si subclase `PrivateHolder` :

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

y tratar de acceder a sus miembros, obtendremos lo siguiente:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
    std::cout << pd.derived_private_value << std::endl;
}
```

Tenga en cuenta que la función miembro `PrivateHolderDerived` no puede acceder a `PrivateHolder::private_value` , mientras que la función friend puede hacerlo.

Método de amigo

Los métodos pueden declararse como amigos, así como funciones:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

Clase de amigo

Una clase entera puede ser declarada como amiga. La declaración de clase de amigo significa que cualquier miembro del amigo puede acceder a miembros privados y protegidos de la clase declarante:

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
};
```

```
class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}
```

La declaración de la clase amiga no es reflexiva. Si las clases necesitan acceso privado en ambas direcciones, ambas necesitan declaraciones de amigos.

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
private:
    int private_value = 0;
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    // Accesser is a friend of PrivateHolder
    friend class Accesser;
    void reverse_accesse() {
        // but PrivateHolder cannot access Accesser's members.
        Accesser a;
        std::cout << a.private_value;
    }
private:
    int private_value;
};
```

Lea Palabra clave amigo en línea: https://riptutorial.com/es/cplusplus/topic/3275/palabra-clave-amigo

Capítulo 90: palabra clave const

Sintaxis

- const Tipo myVariable = initial; // Declara una variable const; no puede ser cambiado
- Tipo const y myReference = myVariable; // Declara una referencia a una variable const.
- Tipo const * myPointer = & myVariable; // Declara un puntero a const. El puntero puede cambiar, pero el miembro de datos subyacente no se puede cambiar a través del puntero
- Escriba * const myPointer = & myVariable; // Declara un puntero const. El puntero no se puede reasignar para apuntar a otra cosa, pero el miembro de datos subyacente se puede cambiar
- Tipo const * const myPointer = & myVariable; // Declara una constante puntero a const.

Observaciones

Una variable marcada como `const` no puede ser cambiada [1]. Intentar llamar a cualquier operación que no sea constante en él resultará en un error del compilador.

1: Bueno, se puede cambiar a través de `const_cast` , pero casi nunca deberías usar eso

Examples

Variables locales const

Declaración y uso.

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;           // Error: can't assign new value to const variable
```

Encuadernación de referencias y punteros.

```
int &b = a;       // Error: can't bind non-const reference to const variable
const int &c = a; // OK; c is a const reference

int *d = &a;      // Error: can't bind pointer-to-non-const to const variable
const int *e = &a // OK; e is a pointer-to-const

int f = 0;
e = &f;           // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*

*e = 1            // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e

int *g = &f;
*g = 1;           // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

Punteros const

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

//Error: Cannot assign to a const reference
*pA = b;

pA = &b;

*pB = b;

//Error: Cannot assign to const pointer
pB = &b;

//Error: Cannot assign to a const reference
*pC = b;

//Error: Cannot assign to const pointer
pC = &b;
```

Funciones de miembro const

Las funciones de miembro de una clase se pueden declarar `const` , que le dice al compilador y a los futuros lectores que esta función no modificará el objeto:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

En una función miembro de `const` , `this` puntero es efectivamente una `const MyClass *` lugar de una `MyClass *` . Esto significa que no puede cambiar ninguna variable miembro dentro de la función; El compilador emitirá una advertencia. Entonces `setMyInt` no pudo ser declarado `const` .

Casi siempre debe marcar las funciones miembro como `const` cuando sea posible. Sólo se pueden llamar a las funciones miembro `const` en una `const MyClass` .

`static` métodos `static` no pueden ser declarados como `const` . Esto se debe a que un método estático pertenece a una clase y no se llama en un objeto; por lo tanto, nunca puede modificar las variables internas del objeto. Así que declarar métodos `static` como `const` sería redundante.

Evitar la duplicación de código en los métodos const y non-const getter.

En C ++, los métodos que difieren solo por el calificador `const` pueden sobrecargarse. A veces es posible que se necesiten dos versiones de getter que devuelvan una referencia a algún miembro.

Deje que `Foo` sea una clase, que tiene dos métodos que realizan operaciones idénticas y devuelve una referencia a un objeto de tipo `Bar` :

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

La única diferencia entre ellos es que un método es non-const y devuelve una referencia non-const (que se puede usar para modificar un objeto) y el segundo es const y devuelve una referencia const.

Para evitar la duplicación de código, existe la tentación de llamar a un método desde otro. Sin embargo, no podemos llamar al método non-const del const. Pero podemos llamar al método const desde el no const. Eso requerirá usar "const_cast" para eliminar el calificador const.

La solucion es:

```
struct Foo
{
    Bar& GetBar(/*arguments*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* some calculations */
        return foo;
    }
};
```

En el código anterior, llamamos a la versión const de `GetBar` desde `GetBar` no const al `GetBar` esto en tipo const: `const_cast<const Foo*>(this)` . Como llamamos al método const desde non-const, el objeto en sí mismo es non-const y se permite desechar la const.

Examina el siguiente ejemplo más completo:

```
#include <iostream>

class Student
{
```

```
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}
```

Lea palabra clave const en línea: https://riptutorial.com/es/cplusplus/topic/2386/palabra-clave-const

Capítulo 91: palabra clave mutable

Examples

modificador de miembro de clase no estático

`mutable` modificador `mutable` en este contexto se usa para indicar que un campo de datos de un objeto const puede modificarse sin afectar el estado visible externamente del objeto.

Si está pensando en almacenar en caché el resultado de un cálculo costoso, probablemente debería usar esta palabra clave.

Si tiene un campo de datos de bloqueo (por ejemplo, `std::unique_lock`) que está bloqueado y desbloqueado dentro de un método const, esta palabra clave también es lo que podría usar.

No debe usar esta palabra clave para romper la constancia lógica de un objeto.

Ejemplo con caché:

```
class pi_calculator {
 public:
     double get_pi() const {
         if (pi_calculated) {
             return pi;
         } else {
             double new_pi = 0;
             for (int i = 0; i < 1000000000; ++i) {
                 // some calculation to refine new_pi
             }
             // note: if pi and pi_calculated were not mutable, we would get an error from a
compiler
             // because in a const method we can not change a non-mutable field
             pi = new_pi;
             pi_calculated = true;
             return pi;
         }
     }
 private:
     mutable bool pi_calculated = false;
     mutable double pi = 0;
};
```

lambdas mutables

Por defecto, el `operator()` implícito `operator()` de un lambda es `const` . Esto no permite realizar operaciones no `const` en la lambda. Para permitir la modificación de miembros, un lambda puede marcarse como `mutable` , lo que hace que el `operator()` implícito `operator()` no sea `const` :

```
int a = 0;

auto bad_counter = [a] {
```

```
    return a++;   // error: operator() is const
                  // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++;  // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

Lea palabra clave mutable en línea: https://riptutorial.com/es/cplusplus/topic/2705/palabra-clave-mutable

Capítulo 92: Palabras clave

Introducción

Las palabras clave tienen un significado fijo definido por el estándar C ++ y no se pueden utilizar como identificadores. Es ilegal redefinir palabras clave utilizando el preprocesador en cualquier unidad de traducción que incluya un encabezado de biblioteca estándar. Sin embargo, las palabras clave pierden su significado especial dentro de los atributos.

Sintaxis

- asm (*cadena-literal*);
- noexcept (*expresión*) // significado 1
- noexcept (*expresión constante*) // significado 2
- noexcept // significado 2
- tamaño *de la expresión unaria*
- sizeof (*type-id*)
- sizeof ... (*identificador*) // desde C ++ 11
- *nombre de* archivo *identificador de nombre anidado identificador* // que significa 1
- *nombre de* archivo plantilla de *especificador de nombre anidado* (*opt*) *simple-template-id* // que significa 1
- *identificador de nombre de tipo* (*opt*) // significado 2
- typename ... *identifier* (*opt*) // significado 2; desde C ++ 11
- *identificador de nombre de tipo* (*opt*) = *ID de tipo* // significado 2
- plantilla < *plantilla-lista-parámetro* > nombre tipográfico ... (*opt*) *identificador* (*opt*) // significado 3
- plantilla < *plantilla-lista-parámetro* > *identificador de nombre de tipo* (*opt*) = *id-expresión* // significado 3

Observaciones

La lista completa de palabras clave es la siguiente:

- `alignas` (desde C ++ 11)
- `alignof` (desde C ++ 11)
- `asm`
- `auto` : desde C ++ 11 , antes de C ++ 11
- `bool`
- `break`
- `case`
- `catch`
- `char`
- `char16_t` (desde C ++ 11)
- `char32_t` (desde C ++ 11)
- `class`

- `const`
- `constexpr` (desde C ++ 11)
- `const_cast`
- `continue`
- `decltype` (desde C ++ 11)
- `default`
- `delete` para administración de memoria , para funciones (desde C ++ 11)
- `do`
- `double`
- `dynamic_cast`
- `else`
- `enum`
- `explicit`
- `export`
- `extern` como especificador de declaración , en especificación de vinculación , para plantillas
- `false`
- `float`
- `for`
- `friend`
- `goto`
- `if`
- `inline` para funciones , para espacios de nombres (desde C ++ 11), para variables (desde C ++ 17)
- `int`
- `long`
- `mutable`
- `namespace`
- `new`
- `noexcept` (desde C ++ 11)
- `nullptr` (desde C ++ 11)
- `operator`
- `private`
- `protected`
- `public`
- `register`
- `reinterpret_cast`
- `return`
- `short`
- `signed`
- `sizeof`
- `static`
- `static_assert` (desde C ++ 11)
- `static_cast`
- `struct`
- `switch`
- `template`
- `this`
- `thread_local` (desde C ++ 11)
- `throw`
- `true`
- `try`
- `typedef`
- `typeid`
- `typename`

- `union`
- `unsigned`
- `using` para redeclar un nombre , para un alias un espacio de nombres , para un alias un tipo
- `virtual` para funciones , para clases base.
- `void`
- `volatile`
- `wchar_t`
- `while`

Los tokens `final` y `override` no son palabras clave. Pueden usarse como identificadores y tienen un significado especial solo en ciertos contextos.

Los tokens `and` , `and_eq` , `bitand` , `bitor` , `compl` , `not` , `not_eq` , `or` , `or_eq` , `xor` y `xor_eq` son ortografías alternativas de `&&` , `&=` , `&` , `|` , `~` `!` , `!=` , `||` , `|=` , `^` , y `^=` , respectivamente. La norma no los trata como palabras clave, pero son palabras clave para todos los propósitos y propósitos, ya que es imposible redefinirlas o usarlas para significar otra cosa que no sean los operadores que representan.

Los siguientes temas contienen explicaciones detalladas de muchas de las palabras clave en C ++, que sirven para propósitos fundamentales como nombrar tipos básicos o controlar el flujo de ejecución.

- Palabras clave de tipo básico
- Control de flujo
- Iteración
- Palabras clave literales
- Escriba palabras clave
- Palabras clave de la declaración variable
- Clases / Estructuras
- Especificadores de clase de almacenamiento

Examples

asm

La palabra clave `asm` toma un solo operando, que debe ser una cadena literal. Tiene un significado definido por la implementación, pero generalmente se pasa al ensamblador de la implementación, con la salida del ensamblador incorporada en la unidad de traducción.

La declaración `asm` es una *definición* , no una *expresión* , por lo que puede aparecer en el ámbito del bloque o en el ámbito del espacio de nombres (incluido el ámbito global). Sin embargo, dado que el ensamblaje en línea no puede estar limitado por las reglas del lenguaje C ++, `asm` puede no aparecer dentro de una función `constexpr` .

Ejemplo:

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

explícito

1. Cuando se aplica a un constructor de un solo argumento, evita que ese constructor se use para realizar conversiones implícitas.

   ```
   class MyVector {
     public:
       explicit MyVector(uint64_t size);
   };
   MyVector v1(100);  // ok
   uint64_t len1 = 100;
   MyVector v2{len1}; // ok, len1 is uint64_t
   int len2 = 100;
   MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
   ```

   Desde que C ++ 11 introdujo las listas de inicializadores, en C ++ 11 y posteriores, `explicit` puede aplicar `explicit` a un constructor con cualquier número de argumentos, con el mismo significado que en el caso de un solo argumento.

   ```
   struct S {
       explicit S(int x, int y);
   };
   S f() {
       return {12, 34};  // ill-formed
       return S{12, 34}; // ok
   }
   ```

C ++ 11

2. Cuando se aplica a una función de conversión, evita que esa función de conversión se utilice para realizar conversiones implícitas.

   ```
   class C {
       const int x;
     public:
       C(int x) : x(x) {}
       explicit operator int() { return x; }
   };
   C c(42);
   int x = c;                    // ill-formed
   int y = static_cast<int>(c); // ok; explicit conversion
   ```

noexcept

C ++ 11

1. Un operador unario que determina si la evaluación de su operando puede propagar una excepción. Tenga en cuenta que los cuerpos de las funciones llamadas no se examinan, por lo que `noexcept` puede producir falsos negativos. El operando no es evaluado.

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n';   // prints 1
}
```

   En este ejemplo, aunque `bar()` nunca puede lanzar una excepción, `noexcept(bar())` sigue siendo falso porque el hecho de que `bar()` no puede propagar una excepción no se ha especificado explícitamente.

2. Al declarar una función, especifica si la función puede o no propagar una excepción. Solo, declara que la función no puede propagar una excepción. Con un argumento entre paréntesis, declara que la función puede o no puede propagar una excepción dependiendo del valor de verdad del argumento.

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

   En este ejemplo, hemos declarado que `f4` , `f5` y `f6` no pueden propagar excepciones. (Aunque se puede lanzar una excepción durante la ejecución de `f6` , se captura y no se permite que se propague fuera de la función). Hemos declarado que `f2` puede propagar una excepción. Cuando se omite el especificador `noexcept` , es equivalente a `noexcept(false)` , por lo que hemos declarado implícitamente que `f1` y `f3` pueden propagar excepciones, incluso aunque las excepciones no se puedan lanzar durante la ejecución de `f3` .

C ++ 17

Si una función es o no una `noexcept` es parte del tipo de función: es decir, en el ejemplo anterior, `f1` , `f2` y `f3` tienen diferentes tipos de `f4` , `f5` y `f6` . Por lo tanto, `noexcept` también es significativo en punteros de función, argumentos de plantilla, etc.

```
void g1() {}
void g2() noexcept {}
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept
```

```
void (*p2)() noexcept = &g2; // ok; types match
void (*p3)() = &g1;          // ok; types match
void (*p4)() = &g2;          // ok; implicit conversion
```

escribe un nombre

1. Cuando le sigue un nombre calificado, `typename` especifica que es el nombre de un tipo. Esto suele ser necesario en las plantillas, en particular, cuando el especificador de nombre anidado es un tipo dependiente distinto de la instanciación actual. En este ejemplo, `std::decay<T>` depende del parámetro de plantilla `T` , por lo tanto, para nombrar el tipo de `type` anidado, debemos prefijar todo el nombre calificado con `typename` . Para más detalles, vea ¿Dónde y por qué tengo que colocar las palabras clave "plantilla" y "nombre de tipo"?

```
template <class T>
auto decay_copy(T&& r) -> typename std::decay<T>::type;
```

2. Introduce un parámetro de tipo en la declaración de una plantilla . En este contexto, es intercambiable con `class` .

```
template <typename T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

C ++ 17

3. `typename` también se puede utilizar al declarar un parámetro de plantilla de plantilla , antes del nombre del parámetro, al igual que la `class` .

```
template <template <class T> typename U>
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

tamaño de

Un operador unario que produce el tamaño en bytes de su operando, que puede ser una expresión o un tipo. Si el operando es una expresión, no se evalúa. El tamaño es una expresión constante de tipo `std::size_t` .

Si el operando es un tipo, debe estar entre paréntesis.

- Es ilegal aplicar `sizeof` a un tipo de función.
- Es ilegal aplicar `sizeof` a un tipo incompleto, incluido el `void` .
- Si se aplica sizeof a un tipo de referencia `T&` o `T&&` , es equivalente a `sizeof(T)` .
- Cuando se aplica `sizeof` a un tipo de clase, produce el número de bytes en un objeto completo de ese tipo, incluidos los bytes de relleno en el medio o al final. Por lo tanto, una expresión `sizeof` nunca puede tener un valor de 0. Consulte el diseño de los tipos de objeto

para obtener más detalles.
- Los tipos `char` , `signed char` y `unsigned char` tienen un tamaño de 1. A la inversa, un byte se define como la cantidad de memoria necesaria para almacenar un objeto `char` . No significa necesariamente 8 bits, ya que algunos sistemas tienen objetos de `char` más de 8 bits.

Si *expr* es una expresión, `sizeof(` *expr* `)` es equivalente a `sizeof(T)` donde `T` es el tipo de *expr.*

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
```

C ++ 11

El operador `sizeof...` produce el número de elementos en un paquete de parámetros.

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Diferentes palabras clave

vacío C ++

1. Cuando se utiliza como un tipo de retorno de función, la palabra clave void especifica que la función no devuelve un valor. Cuando se usa para la lista de parámetros de una función, void especifica que la función no toma parámetros. Cuando se usa en la declaración de un puntero, void especifica que el puntero es "universal".

2. Si el tipo de un puntero es nulo *, el puntero puede apuntar a cualquier variable que no esté declarada con la palabra clave constante o volátil. Un puntero de vacío no se puede anular a menos que se convierta a otro tipo. Un puntero vacío se puede convertir en cualquier otro tipo de puntero de datos.

3. Un puntero de vacío puede apuntar a una función, pero no a un miembro de clase en C ++.

```
void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main() {
pv = &i;
   // Cast optional in C required in C++
pint = (int *)pv;
```

C ++ volátil

1. Un calificador de tipo que puede usar para declarar que un objeto puede ser modificado en el programa por el hardware.

```
volatile declarator ;
```

C ++ virtual

1. La palabra clave virtual declara una función virtual o una clase base virtual.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Parámetros

1. **especificadores de tipo** Especifica el tipo de retorno de la función miembro virtual.
2. **miembro-función-declarador** Declara una función miembro.
3. **especificador de acceso** Define el nivel de acceso a la clase base, público, protegido o privado. Puede aparecer antes o después de la palabra clave virtual.
4. **nombre-clase-base** identifica un tipo de clase previamente declarado

este puntero

1. El puntero de este es un puntero accesible solo dentro de las funciones miembro no estáticas de una clase, estructura o tipo de unión. Apunta al objeto para el que se llama la función miembro. Las funciones miembro estáticas no tienen este puntero.

```
this->member-identifier
```

Un puntero de este objeto no es parte del objeto en sí; no se refleja en el resultado de una sentencia sizeof en el objeto. En cambio, cuando se llama a una función miembro no estática para un objeto, el compilador pasa la dirección del objeto como un argumento oculto a la función. Por ejemplo, la siguiente llamada de función:

```
myDate.setMonth( 3 );

can be interpreted this way:


setMonth( &myDate, 3 );

The object's address is available from within the member function as the this pointer. Most
uses of this are implicit. It is legal, though unnecessary, to explicitly use this when
referring to members of the class. For example:


void Date::setMonth( int mn )
{
   month = mn;             // These three statements
   this->month = mn;       // are equivalent
   (*this).month = mn;
}

The expression *this is commonly used to return the current object from a member function:


return *this;
```

```
The this pointer is also used to guard against self-reference:


if (&Object != this) {
// do not execute in cases of self-reference
```

Intentar, lanzar y atrapar declaraciones (C ++)

1. Para implementar el manejo de excepciones en C ++, use las expresiones try, throw y catch.
2. Primero, use un bloque de prueba para encerrar una o más declaraciones que puedan generar una excepción.
3. Una expresión de lanzamiento indica que se ha producido una condición excepcional (a menudo, un error) en un bloque try. Puede usar un objeto de cualquier tipo como el operando de una expresión de lanzamiento. Normalmente, este objeto se utiliza para comunicar información sobre el error. En la mayoría de los casos, recomendamos que use la clase std :: exception o una de las clases derivadas que se definen en la biblioteca estándar. Si uno de ellos no es apropiado, le recomendamos que obtenga su propia clase de excepción de std :: exception.
4. Para manejar las excepciones que pueden ser lanzadas, implemente uno o más bloques catch inmediatamente después de un bloque try. Cada bloque catch especifica el tipo de excepción que puede manejar.

```
    MyData md;
try {
   // Code that could throw an exception
   md = GetNetworkResource();
}
catch (const networkIOException& e) {
   // Code that executes when an exception of type
   // networkIOException is thrown in the try block
   // ...
   // Log error message in the exception object
   cerr << e.what();
}
catch (const myDataFormatException& e) {
   // Code that handles another exception type
   // ...
   cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
   // ...
   if (IOSuccess == false)
      throw networkIOException("Unable to connect");
   // ...
   if (readError)
      throw myDataFormatException("Format error");
   // ...
}
```

El código después de la cláusula de prueba es la sección protegida del código. La expresión de lanzamiento arroja, es decir, plantea, una excepción. El bloque de código

después de la cláusula catch es el controlador de excepciones. Este es el controlador que captura la excepción que se produce si los tipos en las expresiones de lanzamiento y captura son compatibles.

```
    try {
   throw CSomeOtherException();
}
catch(...) {
   // Catch all exceptions - dangerous!!!
   // Respond (perhaps only partially) to the exception, then
   // re-throw to pass the exception to some other handler
   // ...
   throw;
}
```

amigo (C ++)

1. En algunas circunstancias, es más conveniente otorgar acceso de nivel de miembro a funciones que no son miembros de una clase o a todos los miembros de una clase separada. Sólo el implementador de la clase puede declarar quiénes son sus amigos. Una función o clase no puede declararse como un amigo de ninguna clase. En una definición de clase, use la palabra clave friend y el nombre de una función que no sea miembro u otra clase para otorgarle acceso a los miembros privados y protegidos de su clase. En una definición de plantilla, un parámetro de tipo se puede declarar como amigo.

2. Si declara una función de amigo que no se había declarado previamente, esa función se exporta al ámbito no clasificador adjunto.

```
class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
   friend int ForwardDeclared::IsAFriend();// C2039 error expected
};
```

funciones de amigo

1. Una función amiga es una función que no es miembro de una clase pero tiene acceso a los miembros privados y protegidos de la clase. Las funciones del amigo no se consideran miembros de la clase; Son funciones externas normales que tienen privilegios de acceso especiales.

2. Los amigos no están dentro del alcance de la clase, y no se les llama mediante los operadores de selección de miembros (. Y ->) a menos que sean miembros de otra clase.

3. La clase que otorga acceso declara una función de amigo. La declaración de amigo se puede colocar en cualquier parte de la declaración de clase. No se ve afectado por las palabras clave de control de acceso.

```
#include <iostream>
```

```
using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
    // Output: 0
           1
}
```

Miembros de la clase como amigos

```
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

Lea Palabras clave en línea: https://riptutorial.com/es/cplusplus/topic/4891/palabras-clave

Capítulo 93: Palabras clave de la declaración variable

Examples

const

Un especificador de tipo; cuando se aplica a un tipo, produce la versión const-calificada del tipo. Consulte la palabra clave const para obtener detalles sobre el significado de `const` .

```
const int x = 123;
x = 456;    // error
int& r = x; // error

struct S {
    void f();
    void g() const;
};
const S s;
s.f(); // error
s.g(); // OK
```

decltype

C ++ 11

Cede el tipo de su operando, que no se evalúa.

- Si el operando `e` es un nombre sin paréntesis adicionales, entonces `decltype(e)` es el *tipo declarado* de `e` .

  ```
  int x = 42;
  std::vector<decltype(x)> v(100, x); // v is a vector<int>
  ```

- Si el operando `e` es un acceso de miembro de clase sin paréntesis adicionales, entonces `decltype(e)` es el *tipo declarado* del miembro al que se accedió.

  ```
  struct S {
      int x = 42;
  };
  const S s;
  decltype(s.x) y; // y has type int, even though s.x is const
  ```

- En todos los demás casos, `decltype(e)` produce tanto el tipo como la categoría de valor de la expresión `e` , como sigue:

  - Si `e` es un valor de tipo `T` , entonces `decltype(e)` es `T&` .
  - Si `e` es un xvalor de tipo `T` , entonces `decltype(e)` es `T&&` .

- Si `e` es un prvalor de tipo `T` , entonces `decltype(e)` es `T`

Esto incluye el caso con paréntesis extraños.

```
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a has type int
decltype(g()) b = g(); // b has type int&
decltype((x)) c = x;   // c has type int&, since x is an lvalue
```

C ++ 14

La forma especial `decltype(auto)` deduce el tipo de una variable de su inicializador o el tipo de retorno de una función de las declaraciones de `return` en su definición, utilizando las reglas de deducción de `decltype` de `decltype` lugar de las de `auto` .

```
const int x = 123;
auto y = x;           // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

firmado

Una palabra clave que forma parte de ciertos nombres de tipo entero.

- Cuando se usa solo, `int` está implícito, de modo que `signed` , `signed int` e `int` son del mismo tipo.
- Cuando se combina con `char` , produce el tipo `signed char` , que es un tipo diferente de `char` , incluso si `char` también está firmado. `signed char` tiene un rango que incluye al menos -127 a +127, ambos inclusive.
- Cuando se combina con `short` , `long` o `long long` , es redundante, ya que esos tipos ya están firmados.
- `signed` no se puede combinar con `bool` , `wchar_t` , `char16_t` o `char32_t` .

Ejemplo:

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

no firmado

Un especificador de tipo que solicita la versión sin firmar de un tipo entero.

- Cuando se usa solo, `int` está implícito, por lo que `unsigned` es del mismo tipo que `unsigned int` .
- El tipo `unsigned char` es diferente del tipo `char` , incluso si `char` no está firmado. Puede contener enteros hasta al menos 255.

- `unsigned` también se puede combinar con `short` , `long` o `long long` . No se puede combinar con `bool` , `wchar_t` , `char16_t` o `char32_t` .

Ejemplo:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
    // note: returning invert_case_table[c] directly does the
    // wrong thing on implementations where char is a signed type
}
```

volátil

Un calificador de tipo; cuando se aplica a un tipo, produce la versión calificada volátil del tipo. La calificación volátil desempeña el mismo papel que la calificación `const` en el sistema de tipos, pero la `volatile` no impide que se modifiquen los objetos; en cambio, obliga al compilador a tratar todos los accesos a tales objetos como efectos secundarios.

En el ejemplo a continuación, si `memory_mapped_port` no fuera volátil, el compilador podría optimizar la función para que realice solo la escritura final, lo que sería incorrecto si `sizeof(int)` es mayor que 1. La calificación de `volatile` obliga a tratar todo `sizeof(int)` escribe como diferentes efectos secundarios y, por tanto, realiza todos ellos (en orden).

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Lea Palabras clave de la declaración variable en línea:
https://riptutorial.com/es/cplusplus/topic/7840/palabras-clave-de-la-declaracion-variable

Capítulo 94: Palabras clave de tipo básico

Examples

En t

Indica un tipo entero con signo con "el tamaño natural sugerido por la arquitectura del entorno de ejecución", cuyo rango incluye al menos -32767 a +32767, ambos inclusive.

```
int x = 2;
int y = 3;
int z = x + y;
```

Se puede combinar con `unsigned` , `short` , `long` y `long long` (qv) para obtener otros tipos de enteros.

bool

Un tipo entero cuyo valor puede ser `true` o `false` .

```
bool is_even(int x) {
    return x%2 == 0;
}
const bool b = is_even(47); // false
```

carbonizarse

Un tipo entero que es "lo suficientemente grande como para almacenar cualquier miembro del conjunto de caracteres básico de la implementación". Está definido por la implementación si `char` está firmado (y tiene un rango de al menos -127 a +127, inclusive) o no firmado (y tiene un rango de al menos 0 a 255, inclusive).

```
const char zero = '0';
const char one = zero + 1;
const char newline = '\n';
std::cout << one << newline; // prints 1 followed by a newline
```

char16_t

C ++ 11

Un tipo entero sin signo con el mismo tamaño y alineación que `uint_least16_t` , que por lo tanto es lo suficientemente grande como para contener una unidad de código UTF-16.

```
const char16_t message[] = u"□□□□□\n";            // Chinese for "hello, world\n"
std::cout << sizeof(message)/sizeof(char16_t) << "\n"; // prints 7
```

char32_t

C ++ 11

Un tipo entero sin signo con el mismo tamaño y alineación que `uint_least32_t` , que por lo tanto es lo suficientemente grande como para contener una unidad de código UTF-32.

```
const char32_t full_house[] = U"□□□□□";               // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\n"; // prints 6
```

flotador

Un tipo de punto flotante. Tiene el rango más estrecho de los tres tipos de punto flotante en C ++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

doble

Un tipo de punto flotante. Su gama incluye la de `float` . Cuando se combina con `long` , denota el tipo de punto flotante `long double` , cuyo rango incluye el de `double` .

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

largo

Indica un tipo entero con signo que es al menos tan largo como `int` , y cuyo rango incluye al menos -2147483647 a +2147483647, inclusive (es decir, - (2 ^ 31 - 1) a + (2 ^ 31 - 1)). Este tipo también se puede escribir como `long int` .

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

La combinación `long double` denota un tipo de punto flotante, que tiene el rango más amplio de los tres tipos de punto flotante.

```
long double area(long double radius) {
    const long double pi = 3.14159265358979323851;
    return pi*radius*radius;
}
```

C ++ 11

Cuando el especificador `long` aparece dos veces, como en `long long` , indica un tipo entero con signo que es al menos tan largo como `long` , y cuyo rango incluye al menos -

9223372036854775807 a +9223372036854775807, inclusive (es decir, - (2 ^ 63 - 1) a + (2 ^ 63 - 1)).

```
// support files up to 2 TiB
const long long max_file_size = 2LL << 40;
```

corto

Indica un tipo entero con signo que es al menos tan largo como `char` , y cuyo rango incluye al menos -32767 a +32767, inclusive. Este tipo también se puede escribir como `short int` .

```
// (during the last year)
short hours_worked(short days_worked) {
    return 8*days_worked;
}
```

vacío

Un tipo incompleto; no es posible que un objeto tenga el tipo `void` , ni existen matrices de `void` o referencias a `void` . Se utiliza como el tipo de retorno de funciones que no devuelven nada.

Además, una función se puede declarar de forma redundante con un único parámetro de tipo `void` ; esto es equivalente a declarar una función sin parámetros (por ejemplo, `int main()` y `int main(void)` declaran la misma función). Esta sintaxis está permitida para la compatibilidad con C (donde las declaraciones de funciones tienen un significado diferente al de C ++).

El tipo `void*` ("pointer to `void` ") tiene la propiedad de que cualquier puntero de objeto se puede convertir en él y viceversa y dar como resultado el mismo puntero. Esta función hace que el tipo `void*` adecuado para ciertos tipos de interfaces de borrado de tipo (tipo inseguro), por ejemplo, para contextos genéricos en API de estilo C (por ejemplo, `qsort` , `pthread_create`).

Cualquier expresión se puede convertir en una expresión de tipo `void` ; esto se llama una *expresión de valor descartado* :

```
static_cast<void>(std::printf("Hello, %s!\n", name));  // discard return value
```

Esto puede ser útil para señalar explícitamente que el valor de una expresión no es de interés y que la expresión debe evaluarse solo por sus efectos secundarios.

wchar_t

Un tipo entero lo suficientemente grande como para representar todos los caracteres del conjunto de caracteres extendido soportado más grande, también conocido como el conjunto de caracteres anchos. (No es portátil suponer que `wchar_t` usa alguna codificación en particular, como UTF-16).

Normalmente se usa cuando necesita almacenar caracteres sobre ASCII 255, ya que tiene un tamaño mayor que el del tipo de carácter `char` .

```
const wchar_t message_ahmaric[] = L"ሰላም ልዑል\n"; //Ahmaric for "hello, world\n"
const wchar_t message_chinese[] = L"□□□□□\n";// Chinese for "hello, world\n"
const wchar_t message_hebrew[]  = L"שלום עולם\n"; //Hebrew for "hello, world\n"
const wchar_t message_russian[] = L"Привет мир\n";  //Russian for "hello, world\n"
const wchar_t message_tamil[]   = L"ஹலோ உலகம்\n"; //Tamil for "hello, world\n"
```

Lea Palabras clave de tipo básico en línea:
https://riptutorial.com/es/cplusplus/topic/7839/palabras-clave-de-tipo-basico

Capítulo 95: Paquetes de parametros

Examples

Una plantilla con un paquete de parámetros.

```
template<class ... Types> struct Tuple {};
```

Un paquete de parámetros es un parámetro de plantilla que acepta cero o más argumentos de plantilla. Si una plantilla tiene al menos un paquete de parámetros es una *plantilla variad* .

Expansión de un paquete de parámetros.

El patrón `parameter_pack ...` se expande en una lista de sustituciones separadas por comas de `parameter_pack` con cada uno de sus parámetros.

```
template<class T> // Base of recursion
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) {
  std::cout << first_argument << "\n";
  variadic_printer(other_arguments...); // Parameter pack expansion
}
```

El código anterior invocado con `variadic_printer(1, 2, 3, "hello");` huellas dactilares

```
1
2
3
hello
```

Lea Paquetes de parametros en línea: https://riptutorial.com/es/cplusplus/topic/7668/paquetes-de-parametros

Capítulo 96: Patrón de diseño Singleton

Observaciones

Un **Singleton** está diseñado para garantizar que una clase solo tenga una instancia *y le* brinde un punto de acceso global. Si solo necesita una instancia *o* un punto de acceso global conveniente, pero no ambos, considere otras opciones antes de pasar al singleton.

Las variables globales *pueden* hacer que sea más difícil razonar sobre el código. Por ejemplo, si una de las funciones de llamada no está contenta con los datos que recibe de un Singleton, ahora tiene que rastrear qué es lo que primero está dando los datos erróneos de Singleton en primer lugar.

Los singletons también fomentan el acoplamiento , un término usado para describir dos componentes del código que se unen, reduciendo así la medida de autocontención de cada componente.

Singletons no son compatibles con la concurrencia. Cuando una clase tiene un punto de acceso global, cada subproceso tiene la capacidad de acceder, lo que puede provocar puntos muertos y condiciones de carrera.

Por último, la inicialización perezosa puede causar problemas de rendimiento si se inicializa en el momento incorrecto. Eliminar la inicialización perezosa también elimina algunas de las características que hacen que Singleton sea interesante en primer lugar, como el polimorfismo (ver Subclases).

Fuentes: Patrones de programación de juegos por Robert Nystrom

Examples

Inicialización perezosa

Este ejemplo se ha extraído de la sección de `Q & A` aquí:
http://stackoverflow.com/a/1008289/3807729

Vea este artículo para un diseño simple para un perezoso evaluado con singleton de destrucción garantizada:
¿Puede alguien proporcionarme una muestra de Singleton en c ++?

El clásico perezoso evaluó y destruyó correctamente el singleton.

```
class S
{
    public:
        static S& getInstance()
        {
            static S    instance; // Guaranteed to be destroyed.
```

```
                                // Instantiated on first use.
            return instance;
        }
    private:
        S() {};                 // Constructor? (the {} brackets) are needed here.

        // C++ 03
        // ========
        // Dont forget to declare these two. You want to make sure they
        // are unacceptable otherwise you may accidentally get copies of
        // your singleton appearing.
        S(S const&);            // Don't Implement
        void operator=(S const&); // Don't implement

        // C++ 11
        // =======
        // We can use the better technique of deleting the methods
        // we don't want.
    public:
        S(S const&)             = delete;
        void operator=(S const&)  = delete;

        // Note: Scott Meyers mentions in his Effective Modern
        //       C++ book, that deleted functions should generally
        //       be public as it results in better error messages
        //       due to the compilers behavior to check accessibility
        //       before deleted status
};
```

Consulte este artículo sobre cuándo usar un singleton: (no a menudo)
Singleton: ¿Cómo se debe utilizar?

Vea estos dos artículos sobre el orden de inicialización y cómo hacer frente:
Orden de inicialización de variables estáticas
Encontrar problemas de orden de inicialización estática de C ++

Vea este artículo que describe tiempos de vida:
¿Cuál es el tiempo de vida de una variable estática en una función de C ++?

Vea este artículo que discute algunas implicaciones de subprocesos para singletons:
Instancia Singleton declarada como variable estática del método GetInstance

Vea este artículo que explica por qué el bloqueo de doble comprobación no funcionará en C ++:
¿Cuáles son todas las conductas indefinidas comunes que un programador de C ++ debe conocer?

Subclases

```
class API
{
public:
    static API& instance();

    virtual ~API() {}
```

```
    virtual const char* func1() = 0;
    virtual void func2() = 0;

protected:
    API() {}
    API(const API&) = delete;
    API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1()  override { /* Windows code */ }
    virtual void func2() override { /* Windows code */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux code */ }
    virtual void func2() override { /* Linux code */ }
};

API& API::instance() {
#if PLATFORM == WIN32
    static WindowsAPI instance;
#elif PLATFORM = LINUX
    static LinuxAPI instance;
#endif
    return instance;
}
```

En este ejemplo, un simple conmutador de compilación vincula la clase `API` a la subclase apropiada. De esta manera, se puede acceder a la `API` sin estar acoplada a un código específico de la plataforma.

Hilo seguro Singeton

C ++ 11

Los estándares C ++ 11 garantizan que la inicialización de los objetos de alcance de la función se inicialice de manera sincronizada. Esto se puede utilizar para implementar un singleton seguro para subprocesos con inicialización perezosa .

```
class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
    Foo& operator =(const Foo&) = delete;
};
```

Desinticialización estática segura de singleton.

Hay veces con múltiples objetos estáticos en los que necesita poder garantizar que el *singleton* no se destruirá hasta que todos los objetos estáticos que usan el *singleton* ya no lo necesiten.

En este caso, `std::shared_ptr` puede usarse para mantener el *singleton* activo para todos los usuarios, incluso cuando se llama a los destructores estáticos al final del programa:

```
class Singleton
{
public:
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static std::shared_ptr<Singleton> instance()
    {
        static std::shared_ptr<Singleton> s{new Singleton};
        return s;
    }

private:
    Singleton() {}
};
```

NOTA: Este ejemplo aparece como una respuesta en la sección de preguntas y respuestas aquí.

Lea Patrón de diseño Singleton en línea: https://riptutorial.com/es/cplusplus/topic/2713/patron-de-diseno-singleton

Capítulo 97: Patrón de Plantilla Curiosamente Recurrente (CRTP)

Introducción

Un patrón en el que una clase hereda de una plantilla de clase consigo misma como uno de sus parámetros de plantilla. CRTP se usa generalmente para proporcionar *polimorfismo estático* en C ++.

Examples

El patrón de plantilla curiosamente recurrente (CRTP)

CRTP es una poderosa alternativa estática a las funciones virtuales y la herencia tradicional que se puede usar para dar propiedades de tipos en tiempo de compilación. Funciona al tener una plantilla de clase base que toma, como uno de sus parámetros de plantilla, la clase derivada. Esto permite que se realice legalmente una `static_cast` de su `this` puntero a la clase derivada.

Por supuesto, esto también significa que una clase CRTP *siempre* debe usarse como la clase base de alguna otra clase. Y la clase derivada debe pasar a la clase base.

C ++ 14

Supongamos que tiene un conjunto de contenedores que admiten las funciones `begin()` y `end()` . Los requisitos de la biblioteca estándar para contenedores requieren más funcionalidad. Podemos diseñar una clase base CRTP que proporcione esa funcionalidad, basada únicamente en `begin()` y `end()` :

```
#include <iterator>
template <typename Sub>
class Container {
  private:
    // self() yields a reference to the derived type
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

  public:
    decltype(auto) front() {
      return *self().begin();
    }

    decltype(auto) back() {
      return *std::prev(self().end());
    }

    decltype(auto) size() const {
      return std::distance(self().begin(), self().end());
    }
```

```
    decltype(auto) operator[](std::size_t i) {
      return *std::next(self().begin(), i);
    }
};
```

La clase anterior proporciona las funciones `front()` , `back()` , `size()` y `operator[]` para cualquier subclase que proporcione `begin()` y `end()` . Un ejemplo de subclase es una matriz simple asignada dinámicamente:

```
#include <memory>
// A dynamically allocated array
template <typename T>
class DynArray : public Container<DynArray<T>> {
  public:
    using Base = Container<DynArray<T>>;

    DynArray(std::size_t size)
      : size_{size},
      data_{std::make_unique<T[]>(size_)}
    { }

    T* begin() { return data_.get(); }
    const T* begin() const { return data_.get(); }
    T* end() { return data_.get() + size_; }
    const T* end() const { return data_.get() + size_; }

  private:
    std::size_t size_;
    std::unique_ptr<T[]> data_;
};
```

Los usuarios de la clase `DynArray` pueden usar las interfaces proporcionadas por la clase base CRTP de la siguiente manera:

```
DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);
```

Utilidad: este patrón evita particularmente las llamadas a funciones virtuales en tiempo de ejecución que ocurren para atravesar la jerarquía de herencia y simplemente se basan en conversiones estáticas:

```
DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls
```

La única `Container<DynArray<int>>` estática dentro de la función `begin()` en el `Container<DynArray<int>>` clase base `Container<DynArray<int>>` permite que el compilador optimice drásticamente el código y no se realicen búsquedas de tablas virtuales en el tiempo de ejecución.

Limitaciones: debido a que la clase base tiene una plantilla y es diferente para dos `DynArray` s diferentes, no es posible almacenar los punteros a sus clases base en una matriz de tipo

homogéneo como se podría hacer con la herencia normal, donde la clase base no depende de la derivada tipo:

```
class A {};
class B: public A{};

A* a = new B;
```

CRTP para evitar la duplicación de código

El ejemplo en Visitor Pattern proporciona un caso de uso convincente para CRTP:

```
struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};

struct Circle : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};

struct Square : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};
```

Cada tipo secundario de `IShape` necesita implementar la misma función de la misma manera. Eso es un montón de mecanografía extra. En su lugar, podemos introducir un nuevo tipo en la jerarquía que hace esto por nosotros:

```
template <class Derived>
struct IShapeAcceptor : IShape {
    void accept(IShapeVisitor& visitor) const override {
        // visit with our exact type
        visitor.visit(*static_cast<Derived const*>(this));
    }
};
```

Y ahora, cada forma simplemente necesita heredar del aceptador:

```
struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
    Point center;
    double radius;
};
```

```
struct Square : IShapeAcceptor<Square>
{
    Square(const Point& topLeft, double sideLength) : topLeft(topLeft), sideLength(sideLength)
{}
    Point topLeft;
    double sideLength;
};
```

No es necesario un código duplicado.

Lea Patrón de Plantilla Curiosamente Recurrente (CRTP) en línea:
https://riptutorial.com/es/cplusplus/topic/9269/patron-de-plantilla-curiosamente-recurrente--crtp-

Capítulo 98: Perfilado

Examples

Perfilando con gcc y gprof

El perfilador gprof de GNU, gprof , le permite crear un perfil de su código. Para usarlo, necesitas realizar los siguientes pasos:

1. Construye la aplicación con configuraciones para generar información de perfiles.
2. Generar información de perfil ejecutando la aplicación construida
3. Ver la información de perfil generada con gprof

Para construir la aplicación con configuraciones para generar información de perfiles, agregamos la `-pg` . Así, por ejemplo, podríamos usar

```
$ gcc -pg *.cpp -o app
```

o

```
$ gcc -O2 -pg *.cpp -o app
```

Etcétera.

Una vez que la aplicación, por ejemplo `app` , es construido, ejecutarlo como de costumbre:

```
$ ./app
```

Esto debería producir un archivo llamado `gmon.out` .

Para ver los resultados del perfil, ahora ejecute

```
$ gprof app gmon.out
```

(tenga en cuenta que proporcionamos tanto la aplicación como la salida generada).

Por supuesto, también puede canalizar o redirigir:

```
$ gprof app gmon.out | less
```

Etcétera.

El resultado del último comando debe ser una tabla, cuyas filas son las funciones y cuyas columnas indican el número de llamadas, el tiempo total empleado, el tiempo empleado (es decir,

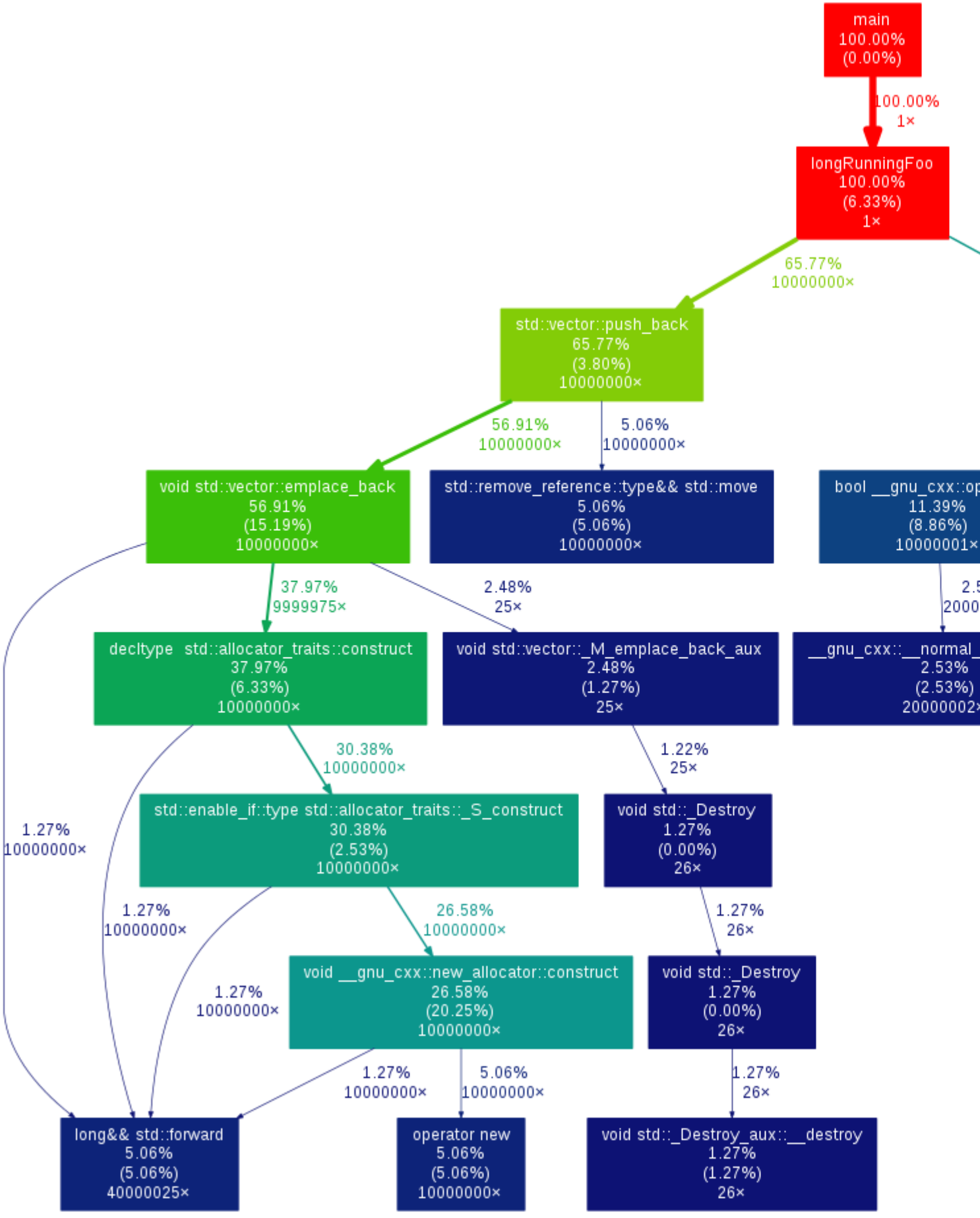
el tiempo empleado en la función, excluyendo las llamadas a los hijos).

Generando diagramas de callgraph con gperf2dot

Para aplicaciones más complejas, los perfiles de ejecución planos pueden ser difíciles de seguir. Esta es la razón por la que muchas herramientas de creación de perfiles también generan algún tipo de información anotada del gráfico de llamadas.

gperf2dot convierte la salida de texto de muchos perfiladores (Linux perf, callgrind, oprofile, etc.) en un diagrama de callgraph. Puede usarlo ejecutando su generador de perfiles (ejemplo para `gprof`):

```
# compile with profiling flags
g++ *.cpp -pg
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```



Perfilando el uso de la CPU con gcc y Google Perf Tools

Google Perf Tools también proporciona un perfilador de CPU, con una interfaz un poco más amigable. Para usarlo:

1. Instalar Google Perf Tools
2. Compila tu código como siempre
3. Agregue la biblioteca de perfiles de `libprofiler` a la ruta de carga de su biblioteca en tiempo de ejecución
4. Utilice `pprof` para generar un perfil de ejecución plano o un diagrama de callgraph

Por ejemplo:

```
# compile code
g++ -O3 -std=c++11 main.cpp -o main

# run with profiler
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000
./main
```

dónde:

- `CPUPROFILE` indica el archivo de salida para el perfil de datos
- `CPUPROFILE_FREQUENCY` indica la frecuencia de muestreo del generador de perfiles;

Utilice `pprof` para postprocesar los datos del perfil.

Puede generar un perfil de llamada plana como texto:

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text --lines ./main main.prof
Using local file ./main.
Using local file main.prof.
Total: 67 samples
      22  32.8%  32.8%       67 100.0% longRunningFoo ??:0
      20  29.9%  62.7%       20  29.9% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
       4   6.0%  68.7%        4   6.0% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
       3   4.5%  73.1%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:388
       3   4.5%  77.6%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:401
       2   3.0%  80.6%        2   3.0% __munmap /build/eglibc-3GlaMS/eglibc-
2.19/misc/../sysdeps/unix/syscall-template.S:81
       2   3.0%  83.6%       12  17.9% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:298
       2   3.0%  86.6%        2   3.0% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:385
       2   3.0%  89.6%        2   3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:26
       1   1.5%  91.0%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
       1   1.5%  92.5%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
       1   1.5%  94.0%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:293
       1   1.5%  95.5%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
```

```
2.19/stdlib/random.c:296
       1   1.5%  97.0%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:371
       1   1.5%  98.5%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:381
       1   1.5% 100.0%        1   1.5% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:28
       0   0.0% 100.0%       67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-
2.19/csu/libc-start.c:287
       0   0.0% 100.0%       67 100.0% _start ??:0
       0   0.0% 100.0%       67 100.0% main ??:0
       0   0.0% 100.0%       14  20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:27
       0   0.0% 100.0%       27  40.3% std::vector::_M_emplace_back_aux ??:0
```

... o puede generar un gráfico de llamadas anotado en un pdf con:

```
pprof --pdf ./main main.prof > out.pdf
```

Lea Perfilado en línea: https://riptutorial.com/es/cplusplus/topic/5347/perfilado

Capítulo 99: Plantillas

Introducción

Las clases, funciones y (desde C ++ 14) las variables pueden tener plantillas. Una plantilla es un fragmento de código con algunos parámetros libres que se convertirán en una clase, función o variable concreta cuando se especifiquen todos los parámetros. Los parámetros pueden ser tipos, valores o plantillas. Una plantilla conocida es `std::vector` , que se convierte en un tipo de contenedor concreto cuando se especifica el tipo de elemento, *por ejemplo,* `std::vector<int>` .

Sintaxis

- *declaración de* plantilla < *plantilla-lista-parámetros* >
- exportar plantilla < *lista de parámetros de plantilla* > *declaración* / * hasta C ++ 11 * /
- plantilla <> *declaración*
- *declaración de* plantilla
- *Declaración de* plantilla externa / * desde C ++ 11 * /
- plantilla < *plantilla-lista-parámetros* > clase ... (*opt*) *identificador* (*opt*)
- plantilla < *plantilla-lista-parámetros* > *identificador de* clase (*opt*) = *id-expresión*
- template < *template-parameters-list* > typename ... (*opt*) *identifier* (*opt*) / * desde C ++ 17 * /
- plantilla < *plantilla-lista-parámetros* > *identificador de nombre de tipo* (*opt*) = *id-expresión* / * desde C ++ 17 * /
- *postfix-expresión* . *expresión-* plantilla de la plantilla
- *postfix-expresión* -> plantilla *id-expresión*
- `template` *especificador de nombre anidado simple-template-id* `::`

Observaciones

La palabra `template` es una palabra clave con cinco significados diferentes en el lenguaje C ++, dependiendo del contexto.

1. Cuando sigue una lista de parámetros de plantilla incluida en `<>` , declara una plantilla como una plantilla de clase , una plantilla de función o una especialización parcial de una plantilla existente.

```
template <class T>
void increment(T& x) { ++x; }
```

2. Cuando es seguido por un *vacío* `<>` , declara una especialización explícita (completa) .

```
template <class T>
void print(T x);

template <> // <-- keyword used in this sense here
```

```
void print(const char* s) {
    // output the content of the string
    printf("%s\n", s);
}
```

3. Cuando sigue una declaración sin `<>` , forma una declaración o definición de instanciación explícita .

```
template <class T>
std::set<T> make_singleton(T x) { return std::set<T>(x); }

template std::set<int> make_singleton(int x); // <-- keyword used in this sense here
```

4. Dentro de una lista de parámetros de plantilla, introduce un parámetro de plantilla de plantilla .

```
template <class T, template <class U> class Alloc>
//                 ^^^^^^^^ keyword used in this sense here
class List {
    struct Node {
        T value;
        Node* next;
    };
    Alloc<Node> allocator;
    Node* allocate_node() {
        return allocator.allocate(sizeof(T));
    }
    // ...
};
```

5. Después del operador de resolución de alcance `::` y los operadores de acceso de miembros de clase `.` y `->` , especifica que el siguiente nombre es una plantilla.

```
struct Allocator {
    template <class T>
    T* allocate();
};

template <class T, class Alloc>
class List {
    struct Node {
        T value;
        Node* next;
    }
    Alloc allocator;
    Node* allocate_node() {
        // return allocator.allocate<Node>();       // error: < and > are interpreted as
                                                    // comparison operators
        return allocator.template allocate<Node>(); // ok; allocate is a template
        //               ^^^^^^^^ keyword used in this sense here
    }
};
```

Antes de C ++ 11, se podría declarar una plantilla con la palabra clave de `export` , convirtiéndola en una plantilla *exportada* . La definición de una plantilla exportada no necesita estar presente en

cada unidad de traducción en la que se crea una instancia de la plantilla. Por ejemplo, se suponía que funcionaba lo siguiente:

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
export template <class T> T identity(T x);
#endif
```

`foo.cpp` :

```
#include "foo.h"
template <class T> T identity(T x) { return x; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    const int x = identity(42); // x is 42
}
```

Debido a la dificultad de implementación, la palabra clave de `export` no era compatible con la mayoría de los compiladores principales. Se eliminó en C ++ 11; ahora, es ilegal utilizar la palabra clave de `export` en absoluto. En su lugar, normalmente es necesario definir plantillas en los encabezados (a diferencia de las funciones que no son de plantilla, que generalmente *no están* definidas en los encabezados). Consulte ¿Por qué las plantillas solo se pueden implementar en el archivo de encabezado?

Examples

Plantillas de funciones

Las plantillas también se pueden aplicar a las funciones (así como a las estructuras más tradicionales) con el mismo efecto.

```
// 'T' stands for the unknown type
// Both of our arguments will be of the same type.
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

Esto puede ser usado de la misma manera que las plantillas de estructura.

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

En ambos casos, el argumento de la plantilla se utiliza para reemplazar los tipos de parámetros;

el resultado funciona igual que una función normal de C ++ (si los parámetros no coinciden con el tipo de plantilla, el compilador aplica las conversiones estándar).

Una propiedad adicional de las funciones de plantilla (a diferencia de las clases de plantilla) es que el compilador puede inferir los parámetros de la plantilla en función de los parámetros pasados a la función.

```
printSum(4, 5);     // Both parameters are int.
                    // This allows the compiler deduce that the type
                    // T is also int.

printSum(5.0, 4);   // In this case the parameters are two different types.
                    // The compiler is unable to deduce the type of T
                    // because there are contradictions. As a result
                    // this is a compile time error.
```

Esta característica nos permite simplificar el código cuando combinamos estructuras de plantillas y funciones. Hay un patrón común en la biblioteca estándar que nos permite crear una `template structure X` mediante la función auxiliar `make_X()` .

```
// The make_X pattern looks like this.
// 1) A template structure with 1 or more template types.
template<typename T1, typename T2>
struct MyPair
{
    T1      first;
    T2      second;
};
// 2) A make function that has a parameter type for
//    each template parameter in the template structure.
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

¿Cómo ayuda esto?

```
auto val1 = MyPair<int, float>{5, 8.7};     // Create object explicitly defining the types
auto val2 = make_MyPair(5, 8.7);            // Create object using the types of the paramters.
                                            // In this code both val1 and val2 are the same
                                            // type.
```

Nota: Esto no está diseñado para acortar el código. Esto está diseñado para hacer que el código sea más robusto. Permite cambiar los tipos cambiando el código en un solo lugar en lugar de en varias ubicaciones.

Reenvío de argumentos

La plantilla puede aceptar tanto las referencias lvalue como rvalue usando la *referencia de reenvío* :

```
template <typename T>
void f(T &&t);
```

En este caso, el tipo real de `t` se deducirá según el contexto:

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

En el primer caso, el tipo `T` se deduce como *referencia a* `X` (`X&`), y el tipo de `t` es la *referencia lvalue a* `X` , mientras que en el segundo caso el tipo de `T` se deduce como `X` y el tipo de `t` como *referencia rvalue a* `X` (`X&&`).

Nota: vale la pena notar que en el primer caso, `decltype(t)` es lo mismo que `T` , pero no en el segundo.

Para reenviar perfectamente `t` a otra función, ya sea una referencia de lvalue o rvalue, se debe usar `std::forward` :

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

Las referencias de reenvío se pueden utilizar con las plantillas variadic:

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Nota: las referencias de reenvío solo se pueden usar para los parámetros de la plantilla, por ejemplo, en el siguiente código, `v` es una referencia de valor, no una referencia de reenvío:

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

Plantilla de clase básica

La idea básica de una plantilla de clase es que el parámetro de plantilla se sustituye por un tipo en tiempo de compilación. El resultado es que la misma clase se puede reutilizar para varios tipos. El usuario especifica qué tipo se utilizará cuando se declara una variable de la clase. Tres ejemplos de esto se muestran en `main()` :

```
#include <iostream>
using std::cout;
```

```
template <typename T>         // A simple class to hold one number of any type
class Number {
public:
    void setNum(T n);         // Sets the class field to the given number
    T plus1() const;          // returns class field's "follower"
private:
    T num;                    // Class field
};

template <typename T>         // Set the class field to the given number
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T>         // returns class field's "follower"
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt;        // Test with an integer (int replaces T in the class)
    anInt.setNum(1);
    cout << "My integer + 1 is " << anInt.plus1() << "\n";     // Prints 2

    Number<double> aDouble;   // Test with a double
    aDouble.setNum(3.1415926535897);
    cout << "My double + 1 is " << aDouble.plus1() << "\n";    // Prints 4.14159

    Number<float> aFloat;     // Test with a float
    aFloat.setNum(1.4);
    cout << "My float + 1 is " << aFloat.plus1() << "\n";      // Prints 2.4

    return 0;  // Successful completion
}
```

Especialización en plantillas

Puede definir la implementación para instancias específicas de una clase de plantilla / método.

Por ejemplo si tienes:

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }
```

A continuación, puede escribir:

```
template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

Luego, un usuario que escribe `sqrt(4.0)` obtendrá la implementación genérica, mientras que `sqrt(4)` obtendrá la implementación especializada.

Especialización en plantillas parciales.

A diferencia de una especialización de plantilla completa, la especialización de plantilla parcial permite introducir una plantilla con algunos de los argumentos de la plantilla existente corregidos. La especialización de plantilla parcial solo está disponible para la clase de plantilla / estructuras:

```
// Common case:
template<typename T, typename U>
struct S {
    T t_val;
    U u_val;
};

// Special case when the first template argument is fixed to int
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) {// Do something}
};
```

Como se muestra arriba, las especializaciones de plantillas parciales pueden introducir conjuntos de datos y funciones completamente diferentes.

Cuando se crea una instancia de una plantilla parcialmente especializada, se selecciona la especialización más adecuada. Por ejemplo, definamos una plantilla y dos especializaciones parciales:

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "General case\n";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int\n";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double\n";
    }
};
```

Ahora las siguientes llamadas:

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

imprimirá

```
General case
```

```
T = int
T = int, U = double
```

Las plantillas de funciones solo pueden ser totalmente especializadas:

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}

// OK.
template<>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // Prints "General case: 1 2.1"
    foo(1,2);    // Prints "Two ints: 1 2"
}

// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

Valor predeterminado del parámetro de la plantilla

Al igual que en el caso de los argumentos de la función, los parámetros de la plantilla pueden tener sus valores predeterminados. Todos los parámetros de plantilla con un valor predeterminado deben declararse al final de la lista de parámetros de plantilla. La idea básica es que los parámetros de la plantilla con el valor predeterminado se pueden omitir mientras se crea una instancia de la plantilla.

Ejemplo simple de uso de valor de parámetro de plantilla predeterminado:

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* Default parameter is ignored, N = 5 */
    my_array<int, 5> a;

    /* Print the length of a.arr: 5 */
    std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

    /* Last parameter is omitted, N = 10 */
    my_array<int> b;

    /* Print the length of a.arr: 10 */
    std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}
```

Plantilla alias

C ++ 11

Ejemplo básico:

```
template<typename T> using pointer = T*;
```

Esta definición hace al `pointer<T>` un alias de `T*` . Por ejemplo:

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

Las plantillas de alias no pueden ser especializadas. Sin embargo, esa funcionalidad se puede obtener indirectamente haciendo que se refieran a un tipo anidado en una estructura:

```
template<typename T>
 struct nonconst_pointer_helper { typedef T* type; };

template<typename T>
 struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;
```

Plantilla plantilla parámetros

A veces nos gustaría pasar a la plantilla un tipo de plantilla sin fijar sus valores. Para esto se crean los parámetros de plantilla de plantilla. Ejemplos de parámetros de plantillas de plantillas muy simples:

```
template <class T>
struct Tag1 { };

template <class T>
struct Tag2 { };

template <template <class> class Tag>
struct IntTag {
   typedef Tag<int> type;
};

int main() {
   IntTag<Tag1>::type t;
}
```

C ++ 11

```
#include <vector>
#include <iostream>

template <class T, template <class...> class C, class U>
C<T> cast_all(const C<U> &c) {
   C<T> result(c.begin(), c.end());
   return result;
```

```
}

int main() {
   std::vector<float> vf = {1.2, 2.6, 3.7};
   auto vi = cast_all<int>(vf);
   for(auto &&i: vi) {
      std::cout << i << std::endl;
   }
}
```

Declaración de argumentos de plantilla no tipo con auto

Antes de C ++ 17, al escribir un parámetro de no tipo de plantilla, primero tenía que especificar su tipo. Así que un patrón común se convirtió en algo como:

```
template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

Pero para expresiones complicadas, usar algo como esto implica tener que escribir `decltype(expr), expr` al crear instancias de plantillas. La solución es simplificar este lenguaje y simplemente permitir el `auto` :

C ++ 17

```
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};

using five = integral_constant<5>;
```

Borrador personalizado vacío para unique_ptr

Un buen ejemplo de motivación puede venir al tratar de combinar la optimización de la base vacía con un eliminador personalizado para `unique_ptr` . Los diferentes borradores de la API de C tienen diferentes tipos de retorno, pero no nos importa, solo queremos que funcione para cualquier función:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) const {
        DeleteFn(ptr);
    }
};

template <T, auto DeleteFn>
```

```
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

Y ahora puede simplemente usar cualquier puntero de función que pueda tomar un argumento de tipo `T` como un parámetro no tipo de plantilla, independientemente del tipo de retorno, y obtener una sobrecarga de tamaño `unique_ptr` fuera de él:

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

Parámetro de plantilla sin tipo

Aparte de los tipos como parámetro de plantilla, podemos declarar valores de expresiones constantes que cumplan uno de los siguientes criterios:

- tipo integral o de enumeración,
- puntero a objeto o puntero a función,
- lvalue referencia a objeto o lvalue referencia a función,
- puntero al miembro,
- `std::nullptr_t` .

Al igual que todos los parámetros de la plantilla, los parámetros de la plantilla que no son de tipo se pueden especificar, predeterminar o derivar de manera explícita mediante la deducción de argumentos de la plantilla.

Ejemplo de uso de parámetros de plantilla no tipo:

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size])  // Pass array by reference. Requires.
{                                        // an exact size. We allow all sizes
    return size;                         // by using a template "size".
}

int main()
{
    char anArrayOfChar[15];
    std::cout << "anArrayOfChar: " << size_of(anArrayOfChar) << "\n";

    int  anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
    std::cout << "anArrayOfData: " << size_of(anArrayOfData) << "\n";
}
```

Ejemplo de especificación explícita de parámetros de tipo y no de tipo de plantilla:

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}
```

Los parámetros de plantilla que no son de tipo son una de las formas de lograr la recurrencia de

la plantilla y permiten realizar la metaprogramación .

Estructuras de datos de plantillas variables

C ++ 14

A menudo es útil definir clases o estructuras que tienen un número variable y un tipo de miembros de datos que se definen en el momento de la compilación. El ejemplo canónico es `std::tuple` , pero a veces es necesario definir sus propias estructuras personalizadas. Aquí hay un ejemplo que define la estructura usando la composición (en lugar de la herencia como con `std::tuple` . Comience con la definición general (vacía), que también sirve como el caso base para la terminación de recrusión en la especialización posterior)

```
template<typename ... T>
struct DataStructure {};
```

Esto ya nos permite definir una estructura vacía, `DataStructure<> data` , aunque aún no es muy útil.

A continuación viene la especialización de casos recursivos:

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

Ahora es suficiente para que `DataStructure<int, float, std::string> data(1, 2.1, "hello")` estructuras de datos arbitrarias, como `DataStructure<int, float, std::string> data(1, 2.1, "hello")` .

Entonces, ¿qué está pasando? Primero, tenga en cuenta que esta es una especialización cuyo requisito es que exista al menos un parámetro de plantilla variable (a saber, `T` arriba), mientras que no se preocupa por la composición específica del paquete `Rest` . Saber que `T` existe permite la definición de su miembro de datos, `first` . El resto de los datos se empaquetan recursivamente como `DataStructure<Rest ... > rest` . El constructor inicia ambos miembros, incluida una llamada de constructor recursiva al miembro `rest` .

Para comprender mejor esto, podemos trabajar con un ejemplo: supongamos que tiene una declaración `DataStructure<int, float> data` . La declaración primero coincide con la especialización, produciendo una estructura con `int first` y `DataStructure<float> rest` de miembros de datos. La definición de `rest` coincide nuevamente con esta especialización, creando `float first` su propia `float first` y los `DataStructure<> rest` . Finalmente, este último `rest` coincide con la definición del caso base, produciendo una estructura vacía.

Puedes visualizar esto de la siguiente manera:

```
DataStructure<int, float>
   -> int first
   -> DataStructure<float> rest
         -> float first
         -> DataStructure<> rest
              -> (empty)
```

Ahora tenemos la estructura de datos, pero no es terriblemente útil todavía, ya que no podemos acceder fácilmente a los elementos de datos individuales (por ejemplo, para acceder al último miembro de `DataStructure<int, float, std::string> data` tendríamos que usar `data.rest.rest.first` , que no es exactamente fácil de usar). Así que le agregamos un método de `get` (solo necesario en la especialización, ya que la estructura del caso base no tiene datos para `get`):

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
    ...
};
```

Como puede ver, esta función de `get` miembros tiene su propia plantilla, esta vez en el índice del miembro que se necesita (de modo que el uso puede ser similar a `data.get<1>()` , similar a `std::tuple`). El trabajo real se realiza mediante una función estática en una clase auxiliar, `GetHelper` . La razón por la que no podemos definir la funcionalidad requerida directamente en `DataStructure` 's `get` es porque (como veremos dentro de poco) que tendría que especializarse en `idx` - pero no es posible especializarse una función miembro de plantilla sin que se especializa la clase que contiene modelo. Tenga en cuenta que el uso de un `auto` estilo C ++ 14 aquí hace que nuestras vidas sean mucho más simples, ya que de lo contrario necesitaríamos una expresión bastante complicada para el tipo de retorno.

Así que a la clase de ayuda. Esta vez necesitaremos una declaración a futuro vacía y dos especializaciones. Primero la declaración:

```
template<size_t idx, typename T>
struct GetHelper;
```

Ahora el caso base (cuando `idx==0`). En este caso acabamos de devolver el `first` miembro:

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
```

```
    }
};
```

En el caso recursivo, decrementamos `idx` e invocamos `GetHelper` para el miembro `rest` :

```
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};
```

Para ver un ejemplo, supongamos que tenemos `DataStructure<int, float> data` y necesitamos `data.get<1>()` . Esto invoca a `GetHelper<1, DataStructure<int, float>>::get(data)` (la segunda especialización), que a su vez invoca a `GetHelper<0, DataStructure<float>>::get(data.rest)` , que finalmente devuelve (por la 1ª especialización como ahora `idx` es 0) `data.rest.first` .

¡Eso es todo! Aquí está el código de funcionamiento completo, con algunos ejemplos de uso en la función `main` :

```
#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
```

```
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;

    return 0;
}
```

Instanciación explícita

Una definición de creación de instancias explícita crea y declara una clase, función o variable concreta de una plantilla, sin usarla todavía. Una instanciación explícita puede ser referenciada desde otras unidades de traducción. Esto se puede usar para evitar definir una plantilla en un archivo de encabezado, si solo se creará una instancia con un conjunto finito de argumentos. Por ejemplo:

```
// print_string.h
template <class T>
void print_string(const T* str);

// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

Debido a que `print_string<char>` y `print_string<wchar_t>` se `print_string<wchar_t>` instancias explícitas en `print_string.cpp` , el vinculador podrá encontrarlos aunque la plantilla `print_string` no esté definida en el encabezado. Si estas declaraciones de creación de instancias explícitas no estuvieran presentes, probablemente se produciría un error de vinculador. Consulte ¿Por qué las plantillas solo se pueden implementar en el archivo de encabezado?

C ++ 11

Si una definición de creación de instancias explícita va precedida por la palabra clave `extern` , se convierte en una *declaración de* creación de instancias explícita. La presencia de una declaración de instanciación explícita para una especialización dada evita la creación de instancias implícita de la especialización dada dentro de la unidad de traducción actual. En cambio, una referencia a esa especialización que de otro modo causaría una creación de instancias implícita puede referirse a una definición de creación de instancias explícita en la misma u otra TU.

foo.h

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // complicated implementation
}
#endif
```

foo.cpp

```
#include "foo.h"
// explicit instantiation definitions for common cases
template void foo(int);
template void foo(double);
```

main.cpp

```
#include "foo.h"
// we already know foo.cpp has explicit instantiation definitions for these
extern template void foo(double);
int main() {
    foo(42);   // instantiates foo<int> here;
               // wasteful since foo.cpp provides an explicit instantiation already!
    foo(3.14); // does not instantiate foo<double> here;
               // uses instantiation of foo<double> in foo.cpp instead
}
```

Lea Plantillas en línea: https://riptutorial.com/es/cplusplus/topic/460/plantillas

Capítulo 100: Plantillas de expresiones

Examples

Plantillas de expresiones básicas en expresiones algebraicas de elementos

---

Introducción y motivación.

Las plantillas de expresión (indicadas como **ETs** a continuación) son una poderosa técnica de metaprogramación de plantilla, utilizada para acelerar los cálculos de expresiones a veces bastante caras. Se usa ampliamente en diferentes dominios, por ejemplo, en la implementación de bibliotecas de álgebra lineal.

Para este ejemplo, considere el contexto de los cálculos algebraicos lineales. Más específicamente, los cálculos que involucran solo **operaciones de elementos** . Este tipo de cálculos son las aplicaciones más básicas de los **ET** y sirven como una buena introducción a cómo funcionan los **ET** internamente.

Veamos un ejemplo motivador. Considere el cálculo de la expresión:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

En aras de la simplicidad, asumiré que la clase `Vector` y operation + (vector plus: element-wise plus operation) y operation * (aquí significa vector producto interno: también element-operation operation) se implementan correctamente, como cómo deberían ser, matemáticamente.

En una implementación convencional sin utilizar **ET** (u otras técnicas similares), se realizan **al menos cinco** construcciones de instancias de `Vector` para obtener el `result` final:

1. Tres instancias correspondientes a `vec_1` , `vec_2` y `vec_3` .
2. Una instancia de `Vector` temporal `_tmp` , que representa el resultado de `_tmp = vec_2*vec_3;` .
3. Finalmente, con el uso adecuado de la **optimización del valor** de **retorno** , la construcción del `result` final en el `result = vec_1 + _tmp;` .

La implementación con **ET** puede **eliminar la creación de** `Vector _tmp` **temporal** en 2, dejando solo **cuatro** construcciones de instancias de `Vector` . Más interesante, considere la siguiente expresión que es más compleja:

```
Vector result = vec_1 + (vec_2*vec3 + vec_1)*(vec_2 + vec_3*vec_1);
```

También habrá cuatro construcciones de instancias de `Vector` en total: `vec_1, vec_2, vec_3` y `result`

. En otras palabras, en este ejemplo, **donde solo están involucradas operaciones de elementos** , se garantiza que **no se crearán objetos temporales a partir de cálculos intermedios** .

¿Cómo funcionan los TE?

Básicamente, las **ET para cualquier cálculo algebraico** consisten en dos bloques de construcción:

1. **Expresiones algebraicas puras** (**PAE**): son proxies / abstracciones de expresiones algebraicas. Un algebraico puro no realiza cálculos reales, son simplemente abstracciones / modelos del flujo de trabajo de cálculo. Un PAE puede ser un modelo de **la entrada o la salida de cualquier expresión algebraica** . Las instancias de **PAE a** menudo se consideran baratas para copiar.
2. **Evaluaciones perezosas** : que son implementaciones de cálculos reales. En el siguiente ejemplo, veremos que para expresiones que solo involucran operaciones de elementos, las evaluaciones perezosas pueden implementar cálculos reales dentro de la operación de acceso indexado en el resultado final, creando así un esquema de evaluación bajo demanda: no se realiza un cálculo Solo si se accede / se solicita el resultado final.

Entonces, específicamente, ¿cómo implementamos **ETs** en este ejemplo? Vamos a caminar a través de él ahora.

Considere siempre el siguiente fragmento de código:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

La expresión para calcular el resultado se puede descomponer en dos subexpresiones:

1. Un vector más expresión (denotado como **plus_expr**)
2. Una expresión de producto interno del vector (denotada como **innerprod_expr**).

Lo que hacen los **ETs** es lo siguiente:

- En lugar de calcular de inmediato cada **subexpresión** , los **ET** primero modelan la expresión completa utilizando una estructura gráfica. Cada nodo en la gráfica representa un **PAE** . La conexión de borde de los nodos representa el flujo de cálculo real. Así que para la expresión anterior, obtenemos el siguiente gráfico:

```
       result = plus_expr( vec_1, innerprod_expr(vec_2, vec_3) )
          /   \
         /     \
        /       \
       /   innerprod_expr( vec_2, vec_3 )
      /         /  \
```

```
     /          /    \
    /          /      \
 vec_1      vec_2    vec_3
```

- El cálculo final se implementa **mirando a través de la jerarquía del gráfico** : ya que aquí se tratan **solo** operaciones de **elementos** , el cálculo de cada valor indexado en el `result` **se puede realizar de forma independiente** : la evaluación final del `result` se puede posponer perezosamente a un **elemento Evaluación sabia** de cada elemento del `result` . En otras palabras, dado que el cálculo de un elemento de `result` , `elem_res` , se puede expresar utilizando los elementos correspondientes en `vec_1` (`elem_1`), `vec_2` (`elem_2`) y `vec_3` (`elem_3`) como:

```
elem_res = elem_1 + elem_2*elem_3;
```

por lo tanto, no es necesario crear un `Vector` temporal para almacenar el resultado del producto interno intermedio: **todo el cómputo de un elemento se puede realizar en conjunto y se puede codificar dentro de la operación de acceso indexado** .

Aquí están los códigos de ejemplo en acción.

Archivo vec.hh: wrapper para std :: vector, utilizado para mostrar el registro cuando se llama a una construcción.

```
#ifndef EXPR_VEC
# define EXPR_VEC

# include <vector>
# include <cassert>
# include <utility>
# include <iostream>
# include <algorithm>
# include <functional>

///
/// This is a wrapper for std::vector. It's only purpose is to print out a log when a
/// vector constructions in called.
/// It wraps the indexed access operator [] and the size() method, which are
/// important for later ETs implementation.
///

// std::vector wrapper.
template<typename ScalarType> class Vector
{
public:
  explicit Vector() { std::cout << "ctor called.\n"; };
  explicit Vector(int size): _vec(size) { std::cout << "ctor called.\n"; };
  explicit Vector(const std::vector<ScalarType> &vec): _vec(vec)
  { std::cout << "ctor called.\n"; };

  Vector(const Vector<ScalarType> & vec): _vec{vec()}
```

```
  { std::cout << "copy ctor called.\n"; };
  Vector(Vector<ScalarType> && vec): _vec(std::move(vec()))
  { std::cout << "move ctor called.\n"; };

  Vector<ScalarType> & operator=(const Vector<ScalarType> &) = default;
  Vector<ScalarType> & operator=(Vector<ScalarType> &&) = default;

  decltype(auto) operator[](int indx) { return _vec[indx]; }
  decltype(auto) operator[](int indx) const { return _vec[indx]; }

  decltype(auto) operator()() & { return (_vec); };
  decltype(auto) operator()() const & { return (_vec); };
  Vector<ScalarType> && operator()() && { return std::move(*this); }

  int size() const { return _vec.size(); }

private:
  std::vector<ScalarType> _vec;
};

///
/// These are conventional overloads of operator + (the vector plus operation)
/// and operator * (the vector inner product operation) without using the expression
/// templates. They are later used for bench-marking purpose.
///

// + (vector plus) operator.
template<typename ScalarType>
auto operator+(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops plus -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::plus<>());
  return Vector<ScalarType>(std::move(_vec));
}

// * (vector inner product) operator.
template<typename ScalarType>
auto operator*(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops multiplies -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::multiplies<>());
  return Vector<ScalarType>(std::move(_vec));
}

#endif //!EXPR_VEC
```

Archivo expr.hh: implementación de plantillas de expresión

para operaciones de elementos (vector plus y vector inner product)

Vamos a dividirlo en secciones.

1. La sección 1 implementa una clase base para todas las expresiones. Emplea el **patrón de plantilla curiosamente recurrente** (CRTP).
2. La Sección 2 implementa el primer **PAE** : un **terminal** , que es solo una envoltura (referencia constante) de una estructura de datos de entrada que contiene un valor de entrada real para el cálculo.
3. La Sección 3 implementa el segundo **PAE** : **binary_operation** , que es una plantilla de clase que se usará más tarde para vector_plus y vector_innerprod. Está parametrizado por el **tipo de operación** , **el PAE del lado izquierdo** y **el PAE del lado derecho** . El cálculo real se codifica en el operador de acceso indexado.
4. La Sección 4 define las operaciones vector_plus y vector_innerprod como una operación de **elementos** . También sobrecarga al operador + y * para **PAE** s: de modo que estas dos operaciones también devuelven **PAE** .

```
#ifndef EXPR_EXPR
# define EXPR_EXPR


/// Fwd declaration.
template<typename> class Vector;

namespace expr
{


/// ----------------------------------------
///
/// Section 1.
///
/// The first section is a base class template for all kinds of expression. It
/// employs the Curiously Recurring Template Pattern, which enables its instantiation
/// to any kind of expression structure inheriting from it.
///
/// ----------------------------------------


  /// Base class for all expressions.
  template<typename Expr> class expr_base
  {
  public:
    const Expr& self() const { return static_cast<const Expr&>(*this); }
    Expr& self() { return static_cast<Expr&>(*this); }

  protected:
    explicit expr_base() {};
    int size() const { return self().size_impl(); }
    auto operator[](int indx) const { return self().at_impl(indx); }
    auto operator()() const { return self()(); };
  };
```

```
/// ----------------------------------------
///
/// The following section 2 & 3 are abstractions of pure algebraic expressions (PAE).
/// Any PAE can be converted to a real object instance using operator(): it is in
/// this conversion process, where the real computations are done.

///
/// Section 2. Terminal
///
/// A terminal is an abstraction wrapping a const reference to the Vector data
/// structure. It inherits from expr_base, therefore providing a unified interface
/// wrapping a Vector into a PAE.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator.
///
/// It might no be necessary for user defined data structures to have a terminal
/// wrapper, since user defined structure can inherit expr_base, therefore eliminates
/// the need to provide such terminal wrapper.
///
/// ----------------------------------------


  /// Generic wrapper for underlying data structure.
  template<typename DataType> class terminal: expr_base<terminal<DataType>>
  {
  public:
    using base_type = expr_base<terminal<DataType>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit terminal(const DataType &val): _val(val) {}
    int size_impl() const { return _val.size(); };
    auto at_impl(int indx) const { return _val[indx]; };
    decltype(auto) operator()() const { return (_val); }

  private:
    const DataType &_val;
  };


/// ----------------------------------------
///
/// Section 3. Binary operation expression.
///
/// This is a PAE abstraction of any binary expression. Similarly it inherits from
/// expr_base.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator. Each call to the at_impl() method is
/// a element wise computation.
///
/// ----------------------------------------


  /// Generic wrapper for binary operations (that are element-wise).
  template<typename Ops, typename lExpr, typename rExpr>
  class binary_ops: public expr_base<binary_ops<Ops,lExpr,rExpr>>
```

```
  {
  public:
    using base_type = expr_base<binary_ops<Ops,lExpr,rExpr>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit binary_ops(const Ops &ops, const lExpr &lxpr, const rExpr &rxpr)
      : _ops(ops), _lxpr(lxpr), _rxpr(rxpr) {};
    int size_impl() const { return _lxpr.size(); };

    /// This does the element-wise computation for index indx.
    auto at_impl(int indx) const { return _ops(_lxpr[indx], _rxpr[indx]); };

    /// Conversion from arbitrary expr to concrete data type. It evaluates
    /// element-wise computations for all indices.
    template<typename DataType> operator DataType()
    {
      DataType _vec(size());
      for(int _ind = 0; _ind < _vec.size(); ++_ind)
        _vec[_ind] = (*this)[_ind];
      return _vec;
    }

  private: /// Ops and expr are assumed cheap to copy.
    Ops   _ops;
    lExpr _lxpr;
    rExpr _rxpr;
  };


/// ----------------------------------------
/// Section 4.
///
/// The following two structs defines algebraic operations on PAEs: here only vector
/// plus and vector inner product are implemented.
///
/// First, some element-wise operations are defined : in other words, vec_plus and
/// vec_prod acts on elements in Vectors, but not whole Vectors.
///
/// Then, operator + & * are overloaded on PAEs, such that: + & * operations on PAEs
/// also return PAEs.
///
/// ----------------------------------------


  /// Element-wise plus operation.
  struct vec_plus_t
  {
    constexpr explicit vec_plus_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs+rhs; }
  };

  /// Element-wise inner product operation.
  struct vec_prod_t
  {
    constexpr explicit vec_prod_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
```

```
    { return lhs*rhs; }
  };

  /// Constant plus and inner product operator objects.
  constexpr vec_plus_t vec_plus{};
  constexpr vec_prod_t vec_prod{};

  /// Plus operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator+(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_plus_t,lExpr,rExpr>(vec_plus,lhs,rhs); }

  /// Inner prod operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator*(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_prod_t,lExpr,rExpr>(vec_prod,lhs,rhs); }

} //!expr


#endif //!EXPR_EXPR
```

Archivo main.cc: test src file

```
# include <chrono>
# include <iomanip>
# include <iostream>
# include "vec.hh"
# include "expr.hh"
# include "boost/core/demangle.hpp"


int main()
{
  using dtype = float;
  constexpr int size = 5e7;

  std::vector<dtype> _vec1(size);
  std::vector<dtype> _vec2(size);
  std::vector<dtype> _vec3(size);

  // ... Initialize vectors' contents.

  Vector<dtype> vec1(std::move(_vec1));
  Vector<dtype> vec2(std::move(_vec2));
  Vector<dtype> vec3(std::move(_vec3));

  unsigned long start_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "\nNo-ETs evaluation starts.\n";

  Vector<dtype> result_no_ets = vec1 + (vec2*vec3);

  unsigned long stop_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
```

```
  std::cout << std::setprecision(6) << std::fixed
            << "No-ETs. Time eclapses: " << (stop_ms_no_ets-start_ms_no_ets)/1000.0
            << " s.\n" << std::endl;

  unsigned long start_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "Evaluation using ETs starts.\n";

  expr::terminal<Vector<dtype>> vec4(vec1);
  expr::terminal<Vector<dtype>> vec5(vec2);
  expr::terminal<Vector<dtype>> vec6(vec3);

  Vector<dtype> result_ets = (vec4 + vec5*vec6);

  unsigned long stop_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << std::setprecision(6) << std::fixed
            << "With ETs. Time eclapses: " << (stop_ms_ets-start_ms_ets)/1000.0
            << " s.\n" << std::endl;

  auto ets_ret_type = (vec4 + vec5*vec6);
  std::cout << "\nETs result's type:\n";
  std::cout << boost::core::demangle( typeid(decltype(ets_ret_type)).name() ) << '\n';

  return 0;
}
```

Aquí hay una salida posible cuando se compila con `-O3 -std=c++14` usando GCC 5.3:

```
ctor called.
ctor called.
ctor called.

No-ETs evaluation starts.
ctor called.
ctor called.
No-ETs. Time eclapses: 0.571000 s.

Evaluation using ETs starts.
ctor called.
With ETs. Time eclapses: 0.164000 s.


ETs result's type:
expr::binary_ops<expr::vec_plus_t, expr::terminal<Vector<float> >,
expr::binary_ops<expr::vec_prod_t, expr::terminal<Vector<float> >,
expr::terminal<Vector<float> > > >
```

Las observaciones son:

- El uso de **ETs** logra un aumento de rendimiento bastante significativo **en este caso** (> 3x).
- Se elimina la creación de objetos vectoriales temporales. Como en el caso de **ETs** , ctor es llamado solo una vez.
- Boost :: demangle se usó para visualizar el tipo de retorno de ET antes de la conversión: claramente construyó exactamente el mismo gráfico de expresión demostrado

anteriormente.

Draw-backs y advertencias

- Una desventaja obvia de las **ET** es la curva de aprendizaje, la complejidad de la implementación y la dificultad de mantenimiento del código. En el ejemplo anterior, donde solo se consideran las operaciones de elementos sabios, la implementación ya contiene una cantidad enorme de placas de preparación, y mucho menos en el mundo real, donde ocurren expresiones algebraicas más complejas en cada cálculo y la independencia de los elementos ya no es válida (por ejemplo, la multiplicación de matrices).), la dificultad será exponencial.

- Otra advertencia de usar **ETs** es que juegan bien con la palabra clave `auto` . Como se mencionó anteriormente, los **PAE** son esencialmente proxies: y los proxies básicamente no funcionan bien con el `auto` . Considere el siguiente ejemplo:

```
auto result = ...;                // Some expensive expression:
                                  // auto returns the expr graph,
                                  // NOT the computed value.
for(auto i = 0; i < 100; ++i)
    ScalrType value = result* ... // Some other expensive computations using result.
```

Aquí, **en cada iteración del bucle for, el resultado se volverá a evaluar** , ya que el gráfico de expresión en lugar del valor calculado se pasa al bucle for.

Bibliotecas existentes implementando **ETs**

- **boost :: proto** es una biblioteca poderosa que te permite definir tus propias reglas y gramáticas para tus propias expresiones y ejecutar usando **ETs** .
- **Eigen** es una biblioteca para álgebra lineal que implementa varios cálculos algebraicos de manera eficiente utilizando **ETs** .

Un ejemplo básico que ilustra plantillas de expresiones.

Una plantilla de expresión es una técnica de optimización en tiempo de compilación utilizada principalmente en computación científica. Su propósito principal es evitar temporarios innecesarios y optimizar los cálculos de bucle usando una sola pasada (normalmente cuando se realizan operaciones en agregados numéricos). Plantillas de expresiones se diseñaron inicialmente con el fin de eludir las ineficiencias de la sobrecarga de operadores ingenuo al implementar numéricos `Array` o `Matrix` tipos. Bjarne Stroustrup introdujo una terminología equivalente para las plantillas de expresión, que las llama "operaciones fusionadas" en la última versión de su libro, "El lenguaje de programación C ++".

Antes de sumergirse realmente en las plantillas de expresión, debe comprender por qué las necesita en primer lugar. Para ilustrar esto, considere la muy simple clase de Matrix que se da a

continuación:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW>
operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

Dada la definición de clase anterior, ahora puede escribir expresiones de matriz como:

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;

Matrix<double, cols, rows> a, b, c;

// initialize a, b & c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
        c(x, y) = 3.0;
    }
}

Matrix<double, cols, rows> d = a + b + c;  // d(x, y) = 6
```

Como se ilustra arriba, ser capaz de sobrecargar el `operator+()` proporciona una notación que imita la notación matemática natural de las matrices.

Desafortunadamente, la implementación anterior también es altamente ineficiente en comparación con una versión equivalente "hecha a mano".

Para entender por qué, debe considerar lo que sucede cuando escribe una expresión como `Matrix d = a + b + c` . De hecho, esto se expande a `((a + b) + c)` u `operator+(operator+(a, b), c)` . En otras palabras, el bucle dentro del `operator+()` se ejecuta dos veces, mientras que podría haberse realizado fácilmente en una sola pasada. Esto también resulta en la creación de 2 temporarios, lo que degrada aún más el rendimiento. En esencia, al agregar la flexibilidad para usar una notación cercana a su contraparte matemática, también ha hecho que la clase `Matrix` altamente ineficiente.

Por ejemplo, sin la sobrecarga del operador, podría implementar una suma de Matrix mucho más eficiente utilizando un solo paso:

```
template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                         const Matrix<T, COL, ROW>& b,
                         const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}
```

Sin embargo, el ejemplo anterior tiene sus propias desventajas porque crea una interfaz mucho más compleja para la clase Matrix (tendrías que considerar métodos como `Matrix::add2()` , `Matrix::AddMultiply()` etc.).

En su lugar, demos un paso atrás y veamos cómo podemos adaptar la sobrecarga del operador para que funcione de una manera más eficiente.

El problema se deriva del hecho de que la expresión `Matrix d = a + b + c` se evalúa demasiado "con entusiasmo" antes de que haya tenido la oportunidad de construir todo el árbol de expresiones. En otras palabras, lo que realmente desea lograr es evaluar `a + b + c` en una sola pasada y solo una vez que realmente necesite asignar la expresión resultante a `d` .

Esta es la idea central detrás de las plantillas de expresión: en lugar de que el `operator+()` evalúe inmediatamente el resultado de agregar dos instancias de Matrix, devolverá una *"plantilla de expresión"* para una futura evaluación una vez que se haya construido todo el árbol de expresiones.

Por ejemplo, aquí hay una posible implementación para una plantilla de expresión correspondiente a la suma de 2 tipos:

```
template <typename LHS, typename RHS>
class MatrixSum
{
public:
    using value_type = typename LHS::value_type;

    MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}
```

```
    value_type operator() (int x, int y) const  {
        return lhs(x, y) + rhs(x, y);
    }
private:
    const LHS& lhs;
    const RHS& rhs;
};
```

Y aquí está la versión actualizada del `operator+()`

```
template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const LHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}
```

Como puede ver, el `operator+()` ya no devuelve una "evaluación impaciente" del resultado de agregar 2 instancias de Matrix (que sería otra instancia de Matrix), sino una plantilla de expresión que representa la operación de adición. El punto más importante a tener en cuenta es que la expresión aún no se ha evaluado. Simplemente contiene referencias a sus operandos.

De hecho, nada le impide crear una instancia de la plantilla de expresión `MatrixSum<>` siguiente manera:

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

Sin embargo, en una etapa posterior, cuando realmente necesite el resultado de la suma, evalúe la expresión `d = a + b` siguiente manera:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}
```

Como se puede ver, otro de los beneficios de utilizar una plantilla de expresión, es que básicamente ha logrado evaluar la suma de `a` y `b` y asignarlo a `d` en una sola pasada.

Además, nada le impide combinar varias plantillas de expresión. Por ejemplo, `a + b + c` daría como resultado la siguiente plantilla de expresión:

```
MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);
```

Y aquí nuevamente puedes evaluar el resultado final usando una sola pasada:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}
```

Finalmente, la última pieza del rompecabezas es en realidad conectar su plantilla de expresión a

la clase `Matrix` . Esto se logra esencialmente al proporcionar una implementación para `Matrix::operator=()` , que toma la plantilla de expresión como un argumento y la evalúa en una sola pasada, como lo hizo "manualmente" antes:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

    template <typename E>
    Matrix<T, COL, ROW>& operator=(const E& expression) {
        for (std::size_t y = 0; y != rows(); ++y) {
            for (std::size_t x = 0; x != cols(); ++x) {
                values[y * COL + x] = expression(x, y);
            }
        }
        return *this;
    }

private:
    std::vector<T> values;
};
```

Lea Plantillas de expresiones en línea: https://riptutorial.com/es/cplusplus/topic/3404/plantillas-de-expresiones

Capítulo 101: Polimorfismo

Examples

Definir clases polimórficas.

El ejemplo típico es una clase de forma abstracta, que luego puede derivarse en cuadrados, círculos y otras formas concretas.

La clase padre:

Empecemos con la clase polimórfica:

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double get_surface() const = 0;
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }

    double get_doubled_surface() const { return 2 * get_surface(); }
};
```

¿Cómo leer esta definición?

- Puede definir el comportamiento polimórfico mediante funciones miembro introducidas con la palabra clave `virtual` . Aquí, `get_surface()` y `describe_object()` obviamente se implementarán de manera diferente para un cuadrado que para un círculo. Cuando se invoca la función en un objeto, la función correspondiente a la clase real del objeto se determinará en el tiempo de ejecución.
- No tiene sentido definir `get_surface()` para una forma abstracta. Es por esto que a la función le sigue `= 0` . Esto significa que la función es *pura función virtual* .
- Una clase polimórfica siempre debe definir un destructor virtual.
- Puede definir funciones miembro no virtuales. Cuando estas funciones se invocarán para un objeto, la función se elegirá en función de la clase utilizada en el momento de la compilación. Aquí `get_double_surface()` se define de esta manera.
- Una clase que contiene al menos una función virtual pura es una clase abstracta. Las clases abstractas no pueden ser instanciadas. Solo puede tener punteros o referencias de un tipo de clase abstracta.

Clases derivadas

Una vez que se define una clase base polimórfica, se puede derivar. Por ejemplo:

```
class Square : public Shape {
    Point top_left;
```

```
    double side_length;
public:
    Square (const Point& top_left, double side)
       : top_left(top_left), side_length(side_length) {}

    double get_surface() override { return side_length * side_length; }
    void describe_object() override {
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y
                  << " with a length of " << side_length << std::endl;
    }
};
```

Algunas explicaciones:

- Puede definir o anular cualquiera de las funciones virtuales de la clase principal. El hecho de que una función fuera virtual en la clase principal la hace virtual en la clase derivada. No hay necesidad de decirle al compilador la palabra clave `virtual` nuevo. Pero se recomienda agregar la `override` la palabra clave al final de la declaración de la función, a fin de evitar errores sutiles causados por variaciones inadvertidas en la firma de la función.
- Si se definen todas las funciones virtuales puras de la clase padre, puede crear una instancia de los objetos para esta clase, de lo contrario, también se convertirá en una clase abstracta.
- No está obligado a anular todas las funciones virtuales. Puede conservar la versión del padre si se adapta a sus necesidades.

Ejemplo de instanciación

```
int main() {

    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also
    square.describe_object();
    std::cout << "Surface: " << square.get_surface() << std::endl;

    Circle circle(Point(0.0, 0.0), 5);

    Shape *ps = nullptr;  // we don't know yet the real type of the object
    ps = &circle;         // it's a circle, but it could as well be a square
    ps->describe_object();
    std::cout << "Surface: " << ps->get_surface() << std::endl;
}
```

Descenso seguro

Supongamos que tiene un puntero a un objeto de una clase polimórfica:

```
Shape *ps;                        // see example on defining a polymorphic class
ps =  get_a_new_random_shape();   // if you don't have such a function yet, you
                                  // could just write ps = new Square(0.0,0.0, 5);
```

una caída hacia abajo sería desde una `Shape` polimórfica general hasta una de sus formas derivadas y más específicas, como `Square` o `Circle` .

¿Por qué abatirse?

La mayoría de las veces, no necesitará saber cuál es el tipo real del objeto, ya que las funciones virtuales le permiten manipular su objeto independientemente de su tipo:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

Si no necesita ningún abatimiento, su diseño sería perfecto.

Sin embargo, es posible que a veces tenga que abatirse. Un ejemplo típico es cuando desea invocar una función no virtual que existe solo para la clase secundaria.

Consideremos por ejemplo los círculos. Sólo los círculos tienen un diámetro. Así que la clase se definiría como:

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class
    Point center;
    double radius;
public:
    Circle (const Point& center, double radius)
       : center(center), radius(radius) {}

    double get_surface() const override { return r * r * M_PI; }

    // this is only for circles. Makes no sense for other shapes
    double get_diameter() const { return 2 * r; }
};
```

La función miembro `get_diameter()` solo existe para círculos. No se definió para un objeto `Shape` :

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

¿Cómo abatir?

Si `static_cast` seguro de que `ps` apunta a un círculo, podría optar por un `static_cast` :

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

Esto hará el truco. Pero es muy arriesgado: si `ps` aparece por algo más que un `Circle` el comportamiento de su código será indefinido.

Así que, en lugar de jugar a la ruleta rusa, debes usar de forma segura un `dynamic_cast` . Esto es específicamente para las clases polimórficas:

```
int main() {
    Circle circle(Point(0.0, 0.0), 10);
    Shape &shape = circle;

    std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

    //shape.get_diameter();   // OUCH !!! Compilation error

    Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
```

```
    if (pc)
        std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
    else
        std::cout << "The shape isn't a circle !" << std::endl;
}
```

Tenga en cuenta que `dynamic_cast` no es posible en una clase que no sea polimórfica. Necesitaría al menos una función virtual en la clase o sus padres para poder usarla.

Polimorfismo y Destructores

Si se pretende que una clase se utilice de forma polimórfica, con las instancias derivadas almacenadas como punteros / referencias base, el destructor de su clase base debe ser `virtual` o estar `protected` . En el primer caso, esto causará que la destrucción del objeto verifique el `vtable` , llamando automáticamente al destructor correcto según el tipo dinámico. En este último caso, la destrucción del objeto a través de un puntero / referencia de clase base está deshabilitada, y el objeto solo se puede eliminar cuando se trata explícitamente como su tipo real.

```
struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};

struct ProtectedDestructor {
  protected:
    ~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~VirtualDestructor() in vtable, sees it's
           // VirtualDerived::~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~ProtectedDestructor() is protected.
delete static_cast<ProtectedDerived*>(pd); // Good.
```

Ambas prácticas garantizan que siempre se llamará al destructor de la clase derivada en las instancias de la clase derivada, lo que evita las fugas de memoria.

Lea Polimorfismo en línea: https://riptutorial.com/es/cplusplus/topic/1717/polimorfismo

Capítulo 102: precedencia del operador

Observaciones

Los operadores se enumeran de arriba a abajo, en precedencia descendente. Los operadores con el mismo número tienen la misma prioridad y la misma asociatividad.

1. `::`
2. Los operadores de postfix: `[] () T(...) . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`
3. Los operadores de prefijo únicos: `++ -- * & + - ! ~ sizeof new delete delete[]` ; la notación de reparto de estilo C, `(T)...` ; (C ++ 11 y superior) `sizeof... alignof noexcept`
4. `.*` y `->*`
5. `*` , `/` , y `%` , operadores aritméticos binarios
6. `+` y `-` , operadores aritméticos binarios
7. `<<` y `>>`
8. `<` , `>` , `<=` , `>=`
9. `==` y `!=`
10. `&` , el operador bit a bit
11. `^`
12. `|`
13. `&&`
14. `||`
15. `?:` (operador condicional ternario)
16. `=` , `*=` , `/=` , `%=` , `+=` , `-=` , `>>=` , `<<=` , `&=` , `^=` , `|=`
17. `throw`
18. `,` (el operador de coma)

La asignación, la asignación compuesta y los operadores condicionales ternarios son asociativos por derecho. Todos los demás operadores binarios son asociativos por la izquierda.

Las reglas para el operador condicional ternario son un poco más complicadas de lo que pueden expresar las reglas de precedencia simples.

- Un operando se enlaza menos fuerte a un `?` a su izquierda o a `:` a su derecha que a cualquier otro operador. Efectivamente, el segundo operando del operador condicional se analiza como si estuviera entre paréntesis. Esto permite una expresión como `a ? b , c : d` para ser sintácticamente válido.
- ¿Un operando se une más fuertemente a un `?` a su derecha que a un operador de asignación o `throw` a su izquierda, entonces `a = b ? c : d` es equivalente a `a = (b ? c : d)` y `throw a ? b : c` es equivalente a `throw (a ? b : c)` .
- Un operando se enlaza más estrechamente con un operador de asignación a su derecha que `:` a su izquierda, entonces `a ? b : c = d` es equivalente a `a ? b : (c = d)` .

Examples

Operadores aritméticos

Los operadores aritméticos en C ++ tienen la misma prioridad que en matemáticas:

La multiplicación y la división han dejado la asociatividad (lo que significa que se evaluarán de izquierda a derecha) y tienen mayor prioridad que la suma y la resta, que también tienen la asociatividad izquierda.

También podemos forzar la precedencia de la expresión usando paréntesis `()`. De la misma manera que lo harías en las matemáticas normales.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)         result: 4
int b = (3+3)/2;        // equal to: (3+3)/2         result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)     result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9     result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);        // equal to: 3 % 1 = 0  3 - 0 = 3
int i = 3-3/1%3;        // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int l = 3-(3/1)%3;      // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int m = 3-(3/(1%3));    // equal to: 1 % 3 = 1  3 / 1 = 3  3 - 3 = 0
```

Operadores lógicos AND y OR

Estos operadores tienen la precedencia habitual en C ++: Y antes de OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

Este código es equivalente a lo siguiente:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Agregar el paréntesis no cambia el comportamiento, sin embargo, hace que sea más fácil de leer. Al agregar estos paréntesis, no existe confusión sobre la intención del escritor.

Lógica && y || operadores: cortocircuito

&& tiene precedencia sobre ||, esto significa que se colocan paréntesis para evaluar lo que se evaluaría en conjunto.

c ++ utiliza la evaluación de cortocircuito en && y || No hacer ejecuciones innecesarias.
Si el lado izquierdo de || devuelve verdadero el lado derecho no necesita ser evaluado nunca más.

```
#include <iostream>
#include <string>

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}


int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
                // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false



    result =
        True("A") || False("B") && False("C");
                // eq. True("A") || (False("B") && False("C"))
    cout << result << " :====================" << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //        the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
}
```

Operadores Unarios

Los operadores unarios actúan sobre el objeto sobre el que son llamados y tienen una alta prioridad. (Ver las observaciones)

Cuando se usa postfix, la acción ocurre solo después de que se evalúa la operación completa, lo que lleva a algunas aritméticas interesantes:

```
int a = 1;
++a;            // result: 2
a--;            // result: 1
int minusa=-a;  // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;      // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;  // prints 5!
int d = ++a/2;      // equal to: (a+1) == 6 / 2 result: 3

int arr[4] =  {1,2,3,4};

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2

int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

Lea precedencia del operador en línea:
https://riptutorial.com/es/cplusplus/topic/3895/precedencia-del-operador

Capítulo 103: Preprocesador

Introducción

El preprocesador de C es un simple analizador / sustituto de texto que se ejecuta antes de la compilación real del código. Usado para extender y facilitar el uso del lenguaje C (y posterior C ++), puede usarse para:

a. **Incluyendo otros archivos** usando `#include`

segundo. **Defina una macro de reemplazo de texto** usando `#define`

do. **Compilación condicional** usando `#if #ifdef`

re. **Lógica específica de la plataforma / compilador** (como una extensión de la compilación condicional)

Observaciones

Las instrucciones del preprocesador se ejecutan antes de que los archivos de origen se entreguen al compilador. Son capaces de lógica condicional de muy bajo nivel. Dado que las construcciones del preprocesador (p. Ej., Macros similares a objetos) no se escriben como las funciones normales (el paso de preprocesamiento ocurre antes de la compilación), el compilador no puede imponer verificaciones de tipos, por lo tanto, deben usarse con cuidado.

Examples

Incluir guardias

Un archivo de encabezado puede ser incluido por otros archivos de encabezado. Por lo tanto, un archivo fuente (unidad de compilación) que incluye múltiples encabezados puede, indirectamente, incluir algunos encabezados más de una vez. Si tal archivo de encabezado que se incluye más de una vez contiene definiciones, el compilador (después del preprocesamiento) detecta una violación de la Regla de una definición (por ejemplo, §3.2 del estándar C ++ 2003) y, por lo tanto, emite un diagnóstico y falla la compilación.

La inclusión múltiple se evita utilizando "incluir guardas", que a veces también se conocen como guardas de encabezado o guardas de macros. Estos se implementan utilizando las directivas `#define` , `#ifndef` , `#endif` del preprocesador.

p.ej

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED
```

```
class Foo    //  a class definition
{
};

#endif
```

La ventaja clave del uso de incluir guardias es que funcionarán con todos los compiladores y preprocesadores que cumplen con los estándares.

Sin embargo, incluir protecciones también causa algunos problemas para los desarrolladores, ya que es necesario asegurarse de que las macros sean únicas en todos los encabezados utilizados en un proyecto. Específicamente, si dos (o más) encabezados usan `FOO_H_INCLUDED` como su protección de inclusión, el primero de esos encabezados incluidos en una unidad de compilación evitará efectivamente que se incluyan los demás. Se presentan desafíos particulares si un proyecto usa una cantidad de bibliotecas de terceros con archivos de encabezado que se usan para incluir guardas en común.

También es necesario asegurarse de que las macros utilizadas en las guardas de inclusión no entren en conflicto con ninguna otra macros definida en los archivos de encabezado.

La mayoría de las implementaciones de C ++ también admiten la directiva `#pragma once` , que garantiza que el archivo solo se incluya una vez en una única compilación. Esta es una directiva estándar *de facto* , pero no es parte de ningún estándar ISO C ++. Por ejemplo:

```
// Foo.h
#pragma once

class Foo
{
};
```

Si bien `#pragma once` evita algunos problemas asociados con la inclusión de guardias, un `#pragma` , por definición en los estándares, es inherentemente un gancho específico del compilador y será ignorado silenciosamente por los compiladores que no lo admiten. Los proyectos que usan `#pragma once` son más difíciles de trasladar a compiladores que no lo admiten.

Una serie de pautas de codificación y estándares de seguridad para C ++ desalientan específicamente cualquier uso del preprocesador que no sea para `#include` los archivos de encabezado o con el propósito de incluir guardas en los encabezados.

Lógica condicional y manejo multiplataforma.

En pocas palabras, la lógica de preprocesamiento condicional consiste en hacer que la lógica de código esté disponible o no esté disponible para la compilación utilizando definiciones de macros.

Tres casos de uso prominentes son:

- diferentes **perfiles de aplicación** (por ejemplo, depuración, lanzamiento, prueba, optimizado) que pueden ser candidatos de la misma aplicación (por ejemplo, con registro adicional).

- **multiplataforma compila** - sola base de código, compilación de múltiples plataformas.
- utilizando una base de código común para múltiples **versiones de aplicaciones** (por ejemplo, **versiones** Basic, Premium y Pro de un software), con características ligeramente diferentes.

Ejemplo a: un enfoque multiplataforma para eliminar archivos (ilustrativo):

```
#ifdef _WIN32
#include <windows.h> // and other windows system files
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
  return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
  return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
  //TODO: check if NSAPI has a more specific function with permission dialog
  return (0 == remove(path.c_str()));
#else
#error "This platform is not supported"
#endif
}
```

Las macros como `_WIN32` , `__APPLE__` o `__unix__` normalmente están predefinidas por las implementaciones correspondientes.

Ejemplo b: habilitar el registro adicional para una compilación de depuración:

```
void s_PrintAppStateOnUserPrompt()
{
    std::cout << "--------BEGIN-DUMP---------------\n"
              << AppState::Instance()->Settings().ToString() << "\n"
#if ( 1 == TESTING_MODE ) //privacy: we want user details only when testing
              << ListToString(AppState::UndoStack()->GetActionNames())
              << AppState::Instance()->CrntDocument().Name()
              << AppState::Instance()->CrntDocument().SignatureSHA() << "\n"
#endif
              << "--------END-DUMP---------------\n"
}
```

Ejemplo c: habilite una función premium en una compilación de producto separada (nota: esto es ilustrativo. A menudo es una mejor idea permitir que una función se desbloquee sin la necesidad de reinstalar una aplicación)

```
void MainWindow::OnProcessButtonClick()
{
#ifndef _PREMIUM
    CreatePurchaseDialog("Buy App Premium", "This feature is available for our App Premium
users. Click the Buy button to purchase the Premium version at our website");
    return;
#endif
    //...actual feature logic here
```

```
}
```

Algunos trucos comunes:

Definición de símbolos en el momento de invocación:

Se puede llamar al preprocesador con símbolos predefinidos (con inicialización opcional). Por ejemplo, este comando (`gcc -E` ejecuta solo el preprocesador)

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

procesa Sample.cpp de la misma manera que lo haría si `#define OPTIMISE_FOR_OS_X` y `#define TESTING_MODE 1` se agregaran a la parte superior de Sample.cpp.

Asegurando una macro se define:

Si una macro no está definida y su valor se compara o verifica, el preprocesador casi siempre asume que el valor es `0` . Hay algunas maneras de trabajar con esto. Un enfoque consiste en asumir que la configuración predeterminada se representa como 0, y cualquier cambio (por ejemplo, en el perfil de compilación de la aplicación) debe realizarse explícitamente (por ejemplo, ENABLE_EXTRA_DEBUGGING = 0 por defecto, establecer -DENABLE_EXTRA_DEBUGGING = 1 para anular). Otro enfoque es hacer que todas las definiciones y valores predeterminados sean explícitos. Esto se puede lograr usando una combinación de `#error` `#ifndef` y `#error` :

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// please include DefaultDefines.h if not already included.
#    error "ENABLE_EXTRA_DEBUGGING is not defined"
#else
#    if ( 1 == ENABLE_EXTRA_DEBUGGING )
  //code
#    endif
#endif
```

Macros

Las macros se clasifican en dos grupos principales: macros similares a objetos y macros similares a funciones. Las macros se tratan como una sustitución de token al principio del proceso de compilación. Esto significa que grandes secciones (o repetidas) de código se pueden abstraer en una macro preprocesadora.

```
// This is an object-like macro
#define    PI         3.14159265358979

// This is a function-like macro.
// Note that we can use previously defined macros
// in other macro definitions (object-like or function-like)
// But watch out, its quite useful if you know what you're doing, but the
// Compiler doesnt know which type to handle, so using inline functions instead
// is quite recommended (But e.g. for Minimum/Maximum functions it is quite useful)
#define    AREA(r)    (PI*(r)*(r))
```

```
// They can be used like this:
double pi_macro   = PI;
double area_macro = AREA(4.6);
```

La biblioteca Qt utiliza esta técnica para crear un sistema de metaobjetos al hacer que el usuario declare la macro Q_OBJECT al frente de la clase definida por el usuario que extiende QObject.

Los nombres de macro generalmente se escriben en mayúsculas, para que sean más fáciles de diferenciar del código normal. Esto no es un requisito, pero muchos programadores lo consideran un buen estilo.

Cuando se encuentra una macro similar a un objeto, se expande como una simple operación de copiar y pegar, con el nombre de la macro reemplazado con su definición. Cuando se encuentra una macro similar a una función, tanto su nombre como sus parámetros se expanden.

```
double pi_squared = PI * PI;
// Compiler sees:
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
// Compiler sees:
double area = (3.14159265358979*(5)*(5))
```

Debido a esto, los parámetros de macros similares a funciones a menudo se incluyen entre paréntesis, como en `AREA()` anterior. Esto es para evitar cualquier error que pueda ocurrir durante la expansión de la macro, específicamente los errores causados por un solo parámetro de macro compuesto por múltiples valores reales.

```
#define BAD_AREA(r) PI * r * r

double bad_area = BAD_AREA(5 + 1.6);
// Compiler sees:
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;

double good_area = AREA(5 + 1.6);
// Compiler sees:
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

También tenga en cuenta que debido a esta simple expansión, se debe tener cuidado con los parámetros pasados a las macros, para evitar efectos secundarios inesperados. Si el parámetro se modifica durante la evaluación, se modificará cada vez que se use en la macro expandida, que generalmente no es lo que queremos. Esto es cierto incluso si la macro incluye los parámetros entre paréntesis para evitar que la expansión rompa algo.

```
int oops = 5;
double incremental_damage = AREA(oops++);
// Compiler sees:
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

Además, las macros no proporcionan seguridad de tipos, lo que lleva a errores difíciles de entender sobre la falta de coincidencia de tipos.

Como los programadores normalmente terminan las líneas con un punto y coma, las macros que están diseñadas para usarse como líneas independientes a menudo están diseñadas para "tragar" un punto y coma; esto evita que cualquier error involuntario sea causado por un punto y coma adicional.

```
#define IF_BREAKER(Func) Func();

if (some_condition)
    // Oops.
    IF_BREAKER(some_func);
else
    std::cout << "I am accidentally an orphan." << std::endl;
```

En este ejemplo, el punto y coma doble inadvertido rompe el bloque `if...else` , impidiendo que el compilador haga coincidir el `else` con el `if` . Para evitar esto, el punto y coma se omite en la definición de macro, lo que provocará que se "trague" el punto y coma inmediatamente después de su uso.

```
#define IF_FIXER(Func) Func()

if (some_condition)
    IF_FIXER(some_func);
else
    std::cout << "Hooray!  I work again!" << std::endl;
```

Dejar el punto y coma final también permite que la macro se use sin terminar la declaración actual, lo que puede ser beneficioso.

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)

// ...

some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

Normalmente, una definición de macro termina al final de la línea. Sin embargo, si una macro necesita cubrir varias líneas, se puede usar una barra invertida al final de una línea para indicar esto. Esta barra invertida debe ser el último carácter de la línea, lo que indica al preprocesador que la siguiente línea debe estar concatenada en la línea actual, tratándolas como una sola línea. Esto se puede utilizar varias veces seguidas.

```
#define TEXT "I \
am \
many \
lines."

// ...

std::cout << TEXT << std::endl; // Output:   I am many lines.
```

Esto es especialmente útil en macros complejas similares a funciones, que pueden necesitar cubrir múltiples líneas.

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("There's no real need for this to use 'new'.")
```

En el caso de macros similares a funciones más complejas, puede ser útil darles su propio alcance para evitar posibles colisiones de nombres o para destruir objetos al final de la macro, similar a una función real. Un lenguaje común para esto es *do mientras 0* , donde la macro está encerrada en un bloque *do-while* . Este bloque generalmente *no* tiene un punto y coma, lo que le permite tragar un punto y coma.

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// Compiler sees:

int x;
do {
    MyClass temp(some_setup_values);
    x = temp.process(41153.7);
} while (0);
```

También hay macros variadic; de manera similar a las funciones variadic, estas toman un número variable de argumentos y luego los expanden todos en lugar de un parámetro especial "Varargs", `__VA_ARGS__` .

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(printf, "%d", 8);
// Compiler sees:
printf("%d", 8);
```

Tenga en cuenta que durante la expansión, `__VA_ARGS__` se puede colocar en cualquier lugar de la definición y se expandirá correctamente.

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)

VARIADIC2(some_func, 3, 8, 6, 9);
// Compiler sees:
some_func(8, 6, 9, 3);
```

En el caso de un parámetro variadic de cero argumentos, diferentes compiladores manejarán la coma final de manera diferente. Algunos compiladores, como Visual Studio, tragarán

silenciosamente la coma sin ninguna sintaxis especial. Otros compiladores, como GCC, requieren que coloque `##` inmediatamente antes de `__VA_ARGS__` . Debido a esto, es aconsejable definir condicionalmente macros variables cuando la portabilidad es una preocupación.

```
// In this example, COMPILER is a user-defined macro specifying the compiler being used.

#if       COMPILER == "VS"
    #define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif     COMPILER == "GCC"
    #define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

Mensajes de error del preprocesador

Los errores de compilación se pueden generar utilizando el preprocesador. Esto es útil por varias razones, algunas de las cuales incluyen, notificar a un usuario si están en una plataforma no compatible o en un compilador no compatible.

por ejemplo, Devolver error si la versión de gcc es 3.0.0 o anterior.

```
#if __GNUC__ < 3
#error "This code requires gcc > 3.0.0"
#endif
```

por ejemplo, Devolver error si se compila en una computadora Apple.

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

Macros predefinidas

Las macros predefinidas son aquellas que define el compilador (en contraste con las definidas por el usuario en el archivo fuente). Esas macros no deben ser redefinidas o no definidas por el usuario.

Las siguientes macros están predefinidas por el estándar C ++:

- `__LINE__` contiene el número de línea de la línea en la que se utiliza esta macro, y puede modificarse mediante la directiva `#line` .
- `__FILE__` contiene el nombre de archivo del archivo en el que se usa esta macro y puede modificarse mediante la directiva `#line` .
- `__DATE__` contiene la fecha (en formato `"Mmm dd yyyy"`) de la compilación del archivo, donde *Mmm* se formatea como si se hubiera obtenido mediante una llamada a `std::asctime()` .
- `__TIME__` contiene el tiempo (en formato `"hh:mm:ss"`) de la compilación del archivo.
- `__cplusplus` es definido por compiladores C ++ (conformes) mientras compila archivos C ++. Su valor es la versión estándar con la que el compilador es **totalmente** `199711L` , es decir, `199711L` para C ++ 98 y C ++ 03, `201103L` para C ++ 11 y `201402L` para C ++ 14 estándar.

c ++ 11

- `__STDC_HOSTED__` se define como `1` si la implementación está *alojada* , o `0` si es *independiente* .

c ++ 17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` contiene un literal `size_t` , que es la alineación utilizada para una llamada al `operator new` desconoce la alineación.

Además, las siguientes macros pueden ser predefinidas por las implementaciones y pueden estar o no presentes:

- `__STDC__` tiene un significado que depende de la implementación, y generalmente se define solo cuando se compila un archivo como C, para indicar el cumplimiento del estándar C completo. (O nunca, si el compilador decide no admitir esta macro).

c ++ 11

- `__STDC_VERSION__` tiene un significado que depende de la implementación, y su valor suele ser la versión C, de manera similar a como `__cplusplus` es la versión C ++. (O incluso no está definido, si el compilador decide no admitir esta macro).
- `__STDC_MB_MIGHT_NEQ_WC__` se define como `1` , si los valores de la codificación restringida del conjunto de caracteres básicos podrían no ser iguales a los valores de sus contrapartes amplias (por ejemplo, si `(uintmax_t)'x' != (uintmax_t)L'x'`)
- `__STDC_ISO_10646__` se define si `wchar_t` está codificado como Unicode y se expande a una constante entera en la forma `yyyymmL` , que indica la última revisión de Unicode compatible.
- `__STDCPP_STRICT_POINTER_SAFETY__` se define como `1` , si la implementación tiene *una seguridad de puntero estricta* (de lo contrario tiene *una seguridad de puntero relajada*)
- `__STDCPP_THREADS__` se define como `1` , si el programa puede tener más de un subproceso de ejecución (aplicable a la *implementación independiente; las implementaciones hospedadas* siempre pueden tener más de un subproceso)

También vale la pena mencionar `__func__` , que no es una macro, sino una función predefinida variable local. Contiene el nombre de la función en la que se utiliza, como una matriz de caracteres estáticos en un formato definido por la implementación.

Además de esas macros predefinidas estándar, los compiladores pueden tener su propio conjunto de macros predefinidas. Uno debe referirse a la documentación del compilador para aprenderlos. P.ej:

- gcc
- Microsoft Visual C ++
- sonido metálico
- Compilador Intel C ++

Algunas de las macros son solo para consultar el soporte de alguna característica:

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C"{ // C code has to be decorated
   // C library header declarations here
```

```
}
#endif
```

Otros son muy útiles para la depuración:

c ++ 11

```
bool success = doSomething( /*some arguments*/ );
if( !success ){
    std::cerr << "ERROR: doSomething() failed on line " << __LINE__ - 2
              << " in function " << __func__ << "()"
              << " in file " << __FILE__
              << std::endl;
}
```

Y otros para control de versiones triviales:

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout << "Hello World program\n"
                  << "v 1.1\n" // I have to remember to update this manually
                  << "compiled: " << __DATE__ << ' ' << __TIME__ // this updates automagically
                  << std::endl;
    }
    else{
        std::cout << "Hello World!\n";
    }
}
```

Macros x

Una técnica idiomática para generar estructuras de código repetidas en tiempo de compilación.

Una X-macro consta de dos partes: la lista y la ejecución de la lista.

Ejemplo:

```
#define LIST \
    X(dog)   \
    X(cat)   \
    X(racoon)

// class Animal {
//  public:
//    void say();
// };

#define X(name) Animal name;
LIST
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X
```

```
    return 0;
}
```

el cual es expandido por el preprocesador en lo siguiente:

```
Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}
```

A medida que las listas se vuelven más grandes (digamos, más de 100 elementos), esta técnica ayuda a evitar el pegado excesivo de copias.

Fuente: https://en.wikipedia.org/wiki/X_Macro

Ver también: X-macros

Si definir una `X` irrelevante para la costura antes de usar `LIST` no es de su agrado, también puede pasar un nombre de macro como argumento:

```
#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(racoon)
```

Ahora, especifica explícitamente qué macro se debe usar al expandir la lista, por ejemplo,

```
#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)
```

Si cada invocación de la `MACRO` debe tomar parámetros adicionales - constante con respecto a la lista, se pueden usar macros variadic

```
//a walkaround for Visual studio
#define EXPAND(x) x

#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(racoon, __VA_ARGS__))
```

El primer argumento lo proporciona `LIST` , mientras que el resto lo proporciona el usuario en la invocación `LIST` . Por ejemplo:

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE,Animal,anim_)
LIST(FORWARD_DECLARE,Object,obj_)
```

se expandirá a

```
Animal anim_dog;
Animal anim_cat;
Animal anim_racoon;
Object obj_dog;
Object obj_cat;
Object obj_racoon;
```

#pragma una vez

La mayoría, pero no todas, las implementaciones de C ++ son compatibles con la directiva `#pragma once` , que garantiza que el archivo solo se incluya una vez en una única compilación. No es parte de ninguna norma ISO C ++. Por ejemplo:

```
// Foo.h
#pragma once

class Foo
{
};
```

Si bien `#pragma once` evita algunos problemas asociados con la inclusión de guardias , un `#pragma` , por definición en los estándares, es inherentemente un gancho específico del compilador y será ignorado silenciosamente por los compiladores que no lo admiten. Los proyectos que usan `#pragma once` deben modificarse para cumplir con los estándares.

Con algunos compiladores, particularmente aquellos que emplean encabezados precompilados , `#pragma once` puede resultar en una considerable aceleración del proceso de compilación. De manera similar, algunos preprocesadores logran una aceleración de la compilación al rastrear qué encabezados han empleado incluyen guardias. El beneficio neto, cuando se utilizan tanto `#pragma once` como incluir guardias, depende de la implementación y puede ser un aumento o una disminución de los tiempos de compilación.

`#pragma once` combinado con incluir guardas, fue el diseño recomendado para los archivos de encabezado al escribir aplicaciones basadas en MFC en Windows, y fue generado por la `add class` Visual Studio, el `add dialog` `add windows` asistentes de `add windows` . Por lo tanto, es muy común encontrarlos combinados en los solicitantes de Windows en C ++.

Operadores de preprocesador

`#` operador `#` o el operador de cadena se usa para convertir un parámetro de Macro a un literal de cadena. Solo se puede utilizar con las macros que tienen argumentos.

```
// preprocessor will convert the parameter x to the string literal x
#define PRINT(x) printf(#x "\n")
```

```
PRINT(This line will be converted to string by preprocessor);
// Compiler sees
printf("This line will be converted to string by preprocessor""\n");
```

El compilador concatena dos cadenas y el argumento final `printf()` será una cadena literal con un carácter de nueva línea al final.

El preprocesador ignorará los espacios antes o después del argumento de la macro. Así que debajo de la declaración impresa nos dará el mismo resultado.

```
PRINT(   This line will be converted to string by preprocessor );
```

Si el parámetro de la cadena literal requiere una secuencia de escape como antes de una comilla doble (), el preprocesador lo insertará automáticamente.

```
PRINT(This "line" will be converted to "string" by preprocessor);
// Compiler sees
printf("This \"line\" will be converted to \"string\" by preprocessor""\n");
```

`##` operador `##` o el operador de pegado de token se utiliza para concatenar dos parámetros o tokens de una macro.

```
// preprocessor will combine the variable and the x
#define PRINT(x) printf("variable" #x " = %d", variable##x)

int variableY = 15;
PRINT(Y);
//compiler sees
printf("variable""Y"" = %d", variableY);
```

y la salida final será

```
variableY = 15
```

Lea Preprocesador en línea: https://riptutorial.com/es/cplusplus/topic/1098/preprocesador

Capítulo 104: Pruebas unitarias en C ++

Introducción

La prueba de unidad es un nivel en la prueba de software que valida el comportamiento y la corrección de las unidades de código.

En C ++, las "unidades de código" a menudo se refieren a clases, funciones o grupos de cualquiera de ellas. La prueba de unidad a menudo se realiza utilizando "marcos de prueba" especializados o "bibliotecas de prueba" que a menudo utilizan patrones de uso o sintaxis no triviales.

Este tema revisará diferentes estrategias y unidades de prueba de bibliotecas o marcos.

Examples

Prueba de google

Google Test es un marco de prueba de C ++ mantenido por Google. Requiere la creación de la biblioteca `gtest` y su vinculación a su marco de prueba al crear un archivo de caso de prueba.

Ejemplo mínimo

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google Test test cases are created using a C++ preprocessor macro
// Here, a "test suite" name and a specific "test name" are provided.
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test will also provide macros for assertions.
    ASSERT_EQ(1+1, 2);
}

// Google Test can be run manually from the main() function
// or, it can be linked to the gtest_main library for an already
// set-up main() function primed to accept Google Test test cases.
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

// Build command: g++ main.cpp -lgtest
```

Captura

Catch es una biblioteca de solo encabezado que le permite usar el estilo de prueba de unidad TDD y BDD .

El siguiente fragmento es de la página de documentación de Catch en este enlace :

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
    GIVEN( "A vector with some items" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "the size is reduced" ) {
            v.resize( 0 );

            THEN( "the size changes but not capacity" ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
        WHEN( "more capacity is reserved" ) {
            v.reserve( 10 );

            THEN( "the capacity changes but not the size" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "less capacity is reserved" ) {
            v.reserve( 0 );

            THEN( "neither size nor capacity are changed" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

Convenientemente, estas pruebas se informarán de la siguiente manera cuando se ejecuten:

```
Scenario: vectors can be sized and resized
     Given: A vector with some items
      When: more capacity is reserved
      Then: the capacity changes but not the size
```

Lea Pruebas unitarias en C ++ en línea: https://riptutorial.com/es/cplusplus/topic/9928/pruebas-unitarias-en-c-plusplus

Capítulo 105: Punteros

Introducción

Un puntero es una dirección que se refiere a una ubicación en la memoria. Se utilizan comúnmente para permitir que las funciones o estructuras de datos conozcan y modifiquen la memoria sin tener que copiar la memoria a la que se hace referencia. Los punteros se pueden utilizar con tipos primitivos (integrados) o definidos por el usuario.

Los punteros hacen uso de la "desreferencia" `*` , "dirección de" `&` , y "flecha" `->` operadores. Los operadores '*' y '->' se usan para acceder a la memoria a la que se apunta, y el operador `&` se usa para obtener una dirección en la memoria.

Sintaxis

- <Tipo de datos> * <Nombre de variable>;
- <Tipo de datos> * <Nombre de variable> = & <Nombre de variable del mismo tipo de datos>;
- <Tipo de datos> * <Nombre de variable> = <Valor del mismo tipo de datos>;
- int * foo; // Un puntero que apunta a un valor entero
- int * bar = & myIntVar;
- barra larga * [2];
- long * bar [] = {& myLongVar1, & myLongVar2}; // Igual a: barra larga * [2]

Observaciones

Tenga en cuenta los problemas al declarar múltiples punteros en la misma línea.

```
int* a, b, c; //Only a is a pointer, the others are regular ints.

int* a, *b, *c; //These are three pointers!

int *foo[2]; //Both *foo[0] and *foo[1] are pointers.
```

Examples

Fundamentos de puntero

C ++ 11

Nota: en todo lo siguiente, se supone la existencia de la constante de C ++ 11 `nullptr` . Para versiones anteriores, reemplace `nullptr` con `NULL` , la constante que solía jugar un rol similar.

Creando una variable de puntero

Se puede crear una variable de puntero utilizando la sintaxis específica `*` , por ejemplo, `int *pointer_to_int;` .
Cuando una variable es de *tipo puntero* (`int *`), solo contiene una dirección de memoria. La dirección de la memoria es la ubicación en la que se almacenan los datos del *tipo subyacente* (`int`).

La diferencia es clara cuando se compara el tamaño de una variable con el tamaño de un puntero al mismo tipo:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
    sizeof(bar) = 24
    sizeof(p_bar0) = 8
*/
```

Tomando la dirección de otra variable

Los punteros se pueden asignar entre sí como variables normales; en este caso, es la **dirección de memoria** que se copia de un puntero a otro, **no los datos reales a los** que apunta un puntero.
Además, pueden tomar el valor `nullptr` que representa una ubicación de memoria nula. Un puntero igual a `nullptr` contiene una ubicación de memoria no válida y, por lo tanto, no hace referencia a datos válidos.

Puede obtener la dirección de memoria de una variable de un tipo dado prefijando la variable con la *dirección del* operador `&` . El valor devuelto por `&` es un puntero al tipo subyacente que contiene la dirección de memoria de la variable (que son datos válidos **siempre que la variable no quede fuera del alcance**).

```
// Copy `p_bar0` into `p_bar_1`.
```

```
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar_2`
big_struct *p_bar2 = &bar;

// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr
```

En contraste con las referencias:

- asignar dos punteros no sobrescribe la memoria a la que se refiere el puntero asignado;
- Los punteros pueden ser nulos.
- La *dirección del* operador se requiere explícitamente.

Accediendo al contenido de un puntero.

Como tomar una dirección requiere `&` , también el acceso al contenido requiere el uso del *operador de referencia* `*` , como un prefijo. Cuando se hace referencia a un puntero, se convierte en una variable del tipo subyacente (en realidad, una referencia a él). Luego se puede leer y modificar, si no `const` .

```
(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`. This prints 5.
std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;
```

La combinación de `*` y el operador `.` es abreviado por `->` :

```
std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " <<  p_bar0->foo1  << std::endl; // Prints 5
```

Desreferenciación de punteros inválidos

Al desreferenciar un puntero, debe asegurarse de que apunta a datos válidos. La anulación de la referencia de un puntero no válido (o un puntero nulo) puede provocar una violación de acceso a la memoria, o leer o escribir datos de basura.

```
big_struct *never_do_this() {
   // This is a local variable. Outside `never_do_this` it doesn't exist.
   big_struct retval;
   retval.foo1 = 11;
   // This returns the address of `retval`.
   return &retval;
   // `retval` is destroyed and any code using the value returned
   // by `never_do_this` has a pointer to a memory location that
   // contains garbage data (or is inaccessible).
}
```

En tal escenario, `g++` y `clang++` emiten correctamente las advertencias:

```
(Clang) warning: address of stack memory associated with local variable 'retval' returned [-
Wreturn-stack-address]
(Gcc)   warning: address of local variable ‘retval’ returned [-Wreturn-local-addr]
```

Por lo tanto, se debe tener cuidado cuando los punteros son argumentos de funciones, ya que podrían ser nulos:

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

Operaciones de puntero

Hay dos operadores para punteros: Dirección de operador (&): devuelve la dirección de memoria de su operando. Operador de contenido de (desreferencia) (*): devuelve el valor de la variable ubicada en la dirección especificada por su operador.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
//Outputs 20(The value of the variable stored in the pointer ptr
```

El asterisco (*) se usa para declarar un puntero con el único propósito de indicar que es un puntero. No confunda esto con el operador de **desreferencia** , que se utiliza para obtener el valor

ubicado en la dirección especificada. Son simplemente dos cosas diferentes representadas con el mismo signo.

Aritmética de puntero

Incremento / Decremento

Un puntero puede ser incrementado o decrementado (prefijo y postfix). Incrementar un puntero avanza el valor del puntero al elemento en la matriz, un elemento más allá del elemento apuntado actualmente. Disminuir un puntero lo mueve al elemento anterior en la matriz.

La aritmética de punteros no está permitida si el tipo al que apunta el puntero no está completo. `void` es siempre un tipo incompleto.

```
char* str = new char[10]; // str = 0x010
++str;                    // str = 0x011  in this case sizeof(char) = 1 byte

int* arr = new int[10];   // arr = 0x00100
++arr;                    // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr;    // void is incomplete.
```

Si se incrementa un puntero al elemento final, entonces el puntero apunta a un elemento más allá del final de la matriz. Tal puntero no puede ser referenciado, pero puede ser disminuido.

Incrementar un puntero al elemento uno más allá del final en la matriz, o disminuir un puntero al primer elemento en una matriz produce un comportamiento indefinido.

Un puntero a un objeto no de matriz puede tratarse, a los efectos de la aritmética de punteros, como si fuera una matriz de tamaño 1.

Suma resta

Se pueden agregar valores enteros a los punteros; actúan como incrementos, pero por un número específico en lugar de por 1. Los valores enteros también se pueden sustraer de los punteros, actuando como disminución del puntero. Al igual que con el incremento / decremento, el puntero debe apuntar a un tipo completo.

```
char* str = new char[10];  // str = 0x010
str += 2;                  // str = 0x010 + 2 * sizeof(char) = 0x012

int* arr = new int[10];    // arr = 0x100
arr += 2;                  // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) ==
4.
```

Diferencia de puntero

La diferencia entre dos punteros del mismo tipo se puede calcular. Los dos punteros deben estar dentro del mismo objeto de matriz; De lo contrario, el comportamiento no definido resulta.

Dados dos punteros `P` y `Q` en la misma matriz, si `P` es el elemento `i` th en la matriz, y `Q` es el elemento `j` th, entonces `P - Q` será `i - j` . El tipo del resultado es `std::ptrdiff_t` , desde `<cstddef>` .

```
char* start = new char[10];  // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; //Equal to 5.
std::ptrdiff_t diff = start - test; //Equal to -5; ptrdiff_t is signed.
```

Lea Punteros en línea: https://riptutorial.com/es/cplusplus/topic/3056/punteros

Capítulo 106: Punteros a los miembros

Sintaxis

- Suponiendo una clase llamada Clase ...
    - type * ptr = & Class :: member; // Solo para miembros estáticos
    - tipo Class :: * ptr = & Class :: member; // Apunta a miembros de la clase no estáticos
- Para los punteros a miembros de clase no estáticos, dadas las siguientes dos definiciones:
    - Instancia de clase;
    - Clase * p = & instancia;
- Punteros a las variables miembro de la clase
    - ptr = & Class :: i; // Punto a la variable i dentro de cada clase
    - instancia. * ptr = 1; // Acceso a la instancia de i
    - p -> * ptr = 1; // acceso p's i
- Punteros a funciones de miembro de clase
    - ptr = & Class :: F; // Punto a la función 'F' dentro de cada clase
    - (instancia. * ptr) (5); // Llamar a la instancia de F
    - (p -> * ptr) (6); // Llamar p's F

Examples

Punteros a funciones miembro estáticas

Una función miembro `static` es como una función C / C ++ ordinaria, excepto con alcance:

- Está dentro de una `class` , por lo que necesita su nombre decorado con el nombre de la clase;
- Tiene accesibilidad, con `public` , `protected` o `private` .

Por lo tanto, si tiene acceso a la función miembro `static` y la decora correctamente, puede señalar la función como cualquier función normal fuera de una `class` :

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // Note that Static() is of type 'Fn'
    static int Static(int i) { return 3*i; }
```

```
}; // Class

int main() {
    Fn *fn;    // fn is a pointer to a type-of Fn

    fn = &MyFn;          // Point to one function
    fn(3);               // Call it
    fn = &Class::Static; // Point to the other function
    fn(4);               // Call it
 } // main()
```

Punteros a funciones miembro

Para acceder a una función miembro de una clase, debe tener un "identificador" para la instancia en particular, ya sea como la instancia en sí, o un puntero o una referencia a ella. Dada una instancia de clase, puedes apuntar a varios de sus miembros con un puntero a miembro, ¡SI obtienes la sintaxis correcta! Por supuesto, el puntero debe ser declarado del mismo tipo que lo que está apuntando a ...

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c;          // Need a Class instance to play with
    Class *p = &c;    // Need a Class pointer to play with

    Fn Class::*fn;    // fn is a pointer to a type-of Fn within Class

    fn = &Class::A;   // fn now points to A within any Class
    (c.*fn)(5);       // Pass 5 to c's function A (via fn)
    fn = &Class::B;   // fn now points to B within any Class
    (p->*fn)(6);      // Pass 6 to c's (via p) function B (via fn)
} // main()
```

A diferencia de los punteros a las variables miembro (en el ejemplo anterior), la asociación entre la instancia de la clase y el puntero del miembro debe vincularse estrechamente entre paréntesis, lo que parece un poco extraño (como si `.*` Y `->*` no fueran extraños ¡suficiente!)

Punteros a variables miembro

Para acceder a un miembro de una `class` , debe tener un "identificador" para la instancia en particular, ya sea como la instancia en sí, o como un puntero o una referencia a ella. Dada una instancia de `class` , puedes apuntar a varios de sus miembros con un puntero a miembro, ¡SI obtienes la sintaxis correcta! Por supuesto, el puntero debe ser declarado del mismo tipo que lo que está apuntando a ...

```
class Class {
public:
    int x, y, z;
    char m, n, o;
}; // Class

int x;  // Global variable

int main() {
    Class c;        // Need a Class instance to play with
    Class *p = &c;  // Need a Class pointer to play with

    int *p_i;       // Pointer to an int

    p_i = &x;       // Now pointing to x
    p_i = &c.x;     // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i;  // Use p_c_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i;      // Use p_c_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()
```

La sintaxis de puntero a miembro requiere algunos elementos sintácticos adicionales:

- Para definir el tipo de puntero, debe mencionar el tipo base, así como el hecho de que está dentro de una clase: `int Class::*ptr;` .
- Si usted tiene una clase o de referencia y desea utilizarlo con un puntero a miembro-, es necesario utilizar el `.*` Operador (afín al `.` Operador).
- Si tiene un puntero a una clase y desea usarlo con un puntero a miembro, debe usar el operador `->*` (similar al operador `->`).

Punteros a variables miembro estáticas

Una variable miembro `static` es como una variable C / C ++ ordinaria, excepto con alcance:

- Está dentro de una `class` , por lo que necesita su nombre decorado con el nombre de la clase;
- Tiene accesibilidad, con `public` , `protected` o `private` .

Por lo tanto, si tiene acceso a la variable miembro `static` y la decora correctamente, entonces puede señalar la variable como cualquier variable normal fuera de una `class` :

```
class Class {
public:
    static int i;
}; // Class

int Class::i = 1; // Define the value of i (and where it's stored!)
```

```
int j = 2;   // Just another global variable

int main() {
    int k = 3; // Local variable

    int *p;

    p = &k;   // Point to k
    *p = 2;   // Modify it
    p = &j;   // Point to j
    *p = 3;   // Modify it
    p = &Class::i; // Point to Class::i
    *p = 4;   // Modify it
} // main()
```

Lea Punteros a los miembros en línea: https://riptutorial.com/es/cplusplus/topic/2130/punteros-a-los-miembros

Capítulo 107: Punteros inteligentes

Sintaxis

- `std::shared_ptr<ClassType> variableName = std::make_shared<ClassType>(arg1, arg2, ...);`
- `std::shared_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`
- `std::unique_ptr<ClassType> variableName = std::make_unique<ClassType>(arg1, arg2, ...);` // C ++ 14
- `std::unique_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`

Observaciones

C ++ no es un lenguaje gestionado por la memoria. La memoria asignada dinámicamente (es decir, los objetos creados con `new`) se "filtrará" si no se desasigna explícitamente (con `delete`). Es responsabilidad del programador asegurarse de que la memoria asignada dinámicamente se libere antes de descartar el último puntero a ese objeto.

Los punteros inteligentes se pueden usar para administrar automáticamente el alcance de la memoria asignada dinámicamente (es decir, cuando la última referencia del puntero se sale del alcance, se elimina).

Los punteros inteligentes se prefieren a los punteros "sin procesar" en la mayoría de los casos. Hacen explícita la semántica de propiedad de la memoria asignada dinámicamente, comunicando en sus nombres si se pretende que un objeto sea compartido o de propiedad única.

Use `#include <memory>` para poder usar punteros inteligentes.

Examples

Compartir propiedad (std :: shared_ptr)

La plantilla de clase `std::shared_ptr` define un puntero compartido que puede compartir la propiedad de un objeto con otros punteros compartidos. Esto contrasta con `std::unique_ptr` que representa propiedad exclusiva.

El comportamiento de compartir se implementa a través de una técnica conocida como conteo de referencias, donde el número de punteros compartidos que apuntan al objeto se almacena a su lado. Cuando este conteo llega a cero, ya sea a través de la destrucción o la reasignación de la última instancia de `std::shared_ptr` , el objeto se destruye automáticamente.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

Para crear varios punteros inteligentes que compartan el mismo objeto, necesitamos crear otro `shared_ptr` que `shared_ptr` alias al primer puntero compartido. Aquí hay 2 formas de hacerlo:

```
std::shared_ptr<Foo> secondShared(firstShared);  // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared;                      // 2nd way: Assigning
```

Cualquiera de las formas anteriores convierte a `secondShared` un puntero compartido que comparte la propiedad de nuestra instancia de `Foo` con `firstShared` .

El puntero inteligente funciona como un puntero en bruto. Esto significa que puedes usar `*` para desreferenciarlos. El operador regular `->` funciona:

```
secondShared->test(); // Calls Foo::test()
```

Finalmente, cuando el último alias `shared_ptr` sale del ámbito, se llama al destructor de nuestra instancia de `Foo` .

Advertencia: la construcción de `shared_ptr` puede `shared_ptr` una excepción `bad_alloc` cuando se deben asignar datos adicionales para la semántica de propiedad compartida. Si al constructor se le pasa un puntero normal, se supone que posee el objeto apuntado y llama al eliminador si se produce una excepción. Esto significa que `shared_ptr<T>(new T(args))` no perderá un objeto `T` si `shared_ptr<T>` asignación de `shared_ptr<T>` . Sin embargo, es recomendable utilizar `make_shared<T>(args)` o `allocate_shared<T>(alloc, args)` , que permiten a la implementación optimizar la asignación de memoria.

Asignación de matrices ([]) utilizando shared_ptr

C ++ 11 C ++ 17

Desafortunadamente, no hay una forma directa de asignar Arrays usando `make_shared<>` .

Es posible crear matrices para `shared_ptr<>` usando `new` y `std::default_delete` .

Por ejemplo, para asignar una matriz de 10 enteros, podemos escribir el código como

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

Aquí es obligatorio especificar `std::default_delete` para asegurarse de que la memoria asignada se limpie correctamente utilizando `delete[]` .

Si conocemos el tamaño en tiempo de compilación, podemos hacerlo de esta manera:

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
  std::shared_ptr<T> operator()const{
    auto r = std::make_shared<std::array<T,N>>();
    if (!r) return {};
    return {r.data(), r};
  }
```

```
};
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }
```

luego `make_shared_array<int[10]>` devuelve un `shared_ptr<int>` apunta a 10 ints, todo construido por defecto.

C ++ 17

Con C ++ 17, `shared_ptr` obtuvo soporte especial para tipos de arreglos. Ya no es necesario especificar el eliminador de matriz de forma explícita, y el puntero compartido se puede anular mediante el operador de índice de matriz `[]` :

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

Los punteros compartidos pueden apuntar a un subobjeto del objeto que posee:

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

Tanto `p2` como `p1` poseen el objeto de tipo `Foo` , pero `p2` apunta a su miembro `int x` . Esto significa que si `p1` queda fuera del alcance o se reasigna, el objeto `Foo` subyacente seguirá vivo, asegurándose de que `p2` no cuelgue.

Importante: un `shared_ptr` solo se conoce a sí mismo y a todos los otros `shared_ptr` que se crearon con el constructor de alias. No conoce ningún otro puntero, incluidos todos los otros `shared_ptr` s creados con una referencia a la misma instancia de `Foo` :

```
Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                 // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been
                 // deleted already!!
```

Propiedad Transferencia de shared_ptr

De forma predeterminada, `shared_ptr` incrementa el recuento de referencia y no transfiere la propiedad. Sin embargo, se puede hacer para transferir la propiedad usando `std::move` :

```
shared_ptr<int> up = make_shared<int>();
// Transferring the ownership
shared_ptr<int> up2 = move(up);
```

```
// At this point, the reference count of up = 0 and the
// ownership of the pointer is solely with up2 with reference count = 1
```

Compartir con propiedad temporal (std :: weak_ptr)

Las instancias de `std::weak_ptr` pueden apuntar a objetos que son propiedad de instancias de `std::shared_ptr` y solo se convierten en propietarios temporales. Esto significa que los punteros débiles no alteran el recuento de referencias del objeto y, por lo tanto, no impiden la eliminación de un objeto si todos los punteros compartidos del objeto se reasignan o destruyen.

En el siguiente ejemplo, se utilizan instancias de `std::weak_ptr` para que la destrucción de un objeto de árbol no se inhiba:

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // Create a TreeNode to serve as the root/parent.
    std::shared_ptr<TreeNode> root(new TreeNode);

    // Give the parent 100 child nodes.
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // Reset the root shared pointer, destroying the root object, and
    // subsequently its child nodes.
    root.reset();
}
```

A medida que los nodos secundarios se agregan a los secundarios del nodo raíz, su `parent` `std::weak_ptr` miembro se establece en el nodo raíz. El `parent` miembro se declara como un puntero débil en lugar de un puntero compartido, de manera que el recuento de referencia del nodo raíz no se incrementa. Cuando el nodo raíz se reinicia al final de `main()` , la raíz se destruye. Dado que el único resto de `std::shared_ptr` referencias a los nodos secundarios estaban contenidos en la colección de la raíz `children` , todos los nodos hijos son posteriormente destruidos también.

Debido a los detalles de la implementación del bloque de control, la memoria asignada shared_ptr puede no liberarse hasta que el `shared_ptr` referencia `weak_ptr` y el `weak_ptr` referencia `weak_ptr` lleguen a cero.

```
#include <memory>
int main()
```

```
{
    {
         std::weak_ptr<int> wk;
         {
             // std::make_shared is optimized by allocating only once
             // while std::shared_ptr<int>(new int(42)) allocates twice.
             // Drawback of std::make_shared is that control block is tied to our integer
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // sh memory should be released at this point...
         }
         // ... but wk is still alive and needs access to control block
     }
     // now memory is released (sh and wk)
}
```

Dado que `std::weak_ptr` no mantiene vivo su objeto referenciado, no es posible el acceso directo de datos a través de `std::weak_ptr` . En su lugar, proporciona una función miembro `lock()` que intenta recuperar un `std::shared_ptr` para el objeto al que se hace referencia:

```
#include <cassert>
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         std::shared_ptr<int> sp;
         {
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // calling lock will create a shared_ptr to the object referenced by wk
             sp = wk.lock();
             // sh will be destroyed after this point, but sp is still alive
         }
         // sp still keeps the data alive.
         // At this point we could even call lock() again
         // to retrieve another shared_ptr to the same data from wk
         assert(*sp == 42);
         assert(!wk.expired());
         // resetting sp will delete the data,
         // as it is currently the last shared_ptr with ownership
         sp.reset();
         // attempting to lock wk now will return an empty shared_ptr,
         // as the data has already been deleted
         sp = wk.lock();
         assert(!sp);
         assert(wk.expired());
     }
}
```

Propiedad única (std :: unique_ptr)

C ++ 11

A `std::unique_ptr` es una plantilla de clase que administra la vida útil de un objeto almacenado dinámicamente. A diferencia de `std::shared_ptr` , el objeto dinámico es propiedad de solo *una*

instancia de `std::unique_ptr` en cualquier momento,

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Nota: `std::unique_ptr` está disponible desde C ++ 11 y `std::make_unique` desde C ++ 14).

Sólo la variable `ptr` mantiene un puntero a un `int` asignado dinámicamente. Cuando un puntero único que posee un objeto queda fuera del alcance, el objeto propio se elimina, es decir, se llama a su destructor si el objeto es de clase y se libera la memoria para ese objeto.

Para usar `std::unique_ptr` y `std::make_unique` con tipos de matriz, use sus especializaciones de matriz:

```
// Creates a unique_ptr to an int with value 59
std::unique_ptr<int> ptr = std::make_unique<int>(59);

// Creates a unique_ptr to an array of 15 ints
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

Puede acceder a `std::unique_ptr` como un puntero en bruto, ya que sobrecarga a esos operadores.

Puede transferir la propiedad del contenido de un puntero inteligente a otro puntero utilizando `std::move` , lo que hará que el puntero inteligente original apunte a `nullptr` .

```
// 1. std::unique_ptr
std::unique_ptr<int> ptr = std::make_unique<int>();

// Change value to 1
*ptr = 1;

// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)
std::unique_ptr<int> ptr2 = std::move(ptr);

int a = *ptr2; // 'a' is 1
int b = *ptr;  // undefined behavior! 'ptr' is 'nullptr'
               // (because of the move command above)
```

Pasando `unique_ptr` a funciones como parámetro:

```
void foo(std::unique_ptr<int> ptr)
{
    // Your code goes here
}

std::unique_ptr<int> ptr = std::make_unique<int>(59);
foo(std::move(ptr))
```

Devolviendo `unique_ptr` desde funciones. Esta es la forma preferida de C ++ 11 de escribir

funciones de fábrica, ya que transmite claramente la semántica de propiedad de la devolución: la persona que llama posee el `unique_ptr` resultante y es responsable de ello.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Compara esto con:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                      // it's not readily apparent what the answer is.
```

C ++ 14

La plantilla de clase `make_unique` se proporciona desde C ++ 14. Es fácil agregarlo manualmente al código C ++ 11:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]()); }
```

C ++ 11

A diferencia del puntero inteligente *tonto* (`std::auto_ptr`), `unique_ptr` también se puede crear una instancia con asignación de vectores (*no* `std::vector`). Los ejemplos anteriores fueron para asignaciones *escalares* . Por ejemplo, para tener una matriz de enteros asignada dinámicamente para 10 elementos, debe especificar `int[]` como tipo de plantilla (y no solo `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Que se puede simplificar con:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Ahora, usas `arr_ptr` como si fuera una matriz:

```
arr_ptr[2] =  10; // Modify third element
```

No tiene que preocuparse por la desasignación. Esta plantilla especializada de la versión llama a

los constructores y destructores adecuadamente. El uso de la versión vectorizada de `unique_ptr` o un `vector` sí mismo es una opción personal.

En versiones anteriores a C ++ 11, `std::auto_ptr` estaba disponible. A diferencia de `unique_ptr` , se permite copiar `auto_ptr` s, sobre el cual la fuente `ptr` perderá la propiedad del puntero contenido y el destino lo recibirá.

Uso de eliminaciones personalizadas para crear una envoltura para una interfaz C

Muchas interfaces C, como SDL2, tienen sus propias funciones de eliminación. Esto significa que no puede utilizar punteros inteligentes directamente:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

En su lugar, necesita definir su propio deleter. Los ejemplos aquí utilizan la estructura `SDL_Surface` que debería liberarse utilizando la función `SDL_FreeSurface()` , pero deberían ser adaptables a muchas otras interfaces en C.

El eliminador debe ser invocable con un argumento de puntero, y por lo tanto puede ser, por ejemplo, un puntero de función simple:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Cualquier otro objeto llamable también funcionará, por ejemplo, una clase con un `operator()` :

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};

std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above
                                                         // as the deleter is value-
initialized
```

Esto no solo le proporciona una administración de memoria automática segura y sin sobrecarga (si utiliza `unique_ptr`), `unique_ptr` que también obtiene una seguridad excepcional.

Tenga en cuenta que el borrado es parte del tipo para `unique_ptr` , y la implementación puede usar la optimización de la base vacía para evitar cualquier cambio en el tamaño de los borradores personalizados vacíos. Entonces, mientras `std::unique_ptr<SDL_Surface, SurfaceDeleter>` y `std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)>` resuelven el mismo problema de una manera similar, el primer tipo sigue siendo solo del tamaño de un puntero mientras que este último tipo debe contener *dos* punteros: tanto la `SDL_Surface*` como la función de puntero! Cuando se tienen eliminaciones personalizadas de función libre, es preferible envolver la función en un tipo vacío.

En los casos en que el conteo de referencias es importante, se podría usar un `shared_ptr` lugar de

un `unique_ptr` . `shared_ptr` siempre almacena un eliminador, esto borra el tipo del eliminador, lo que podría ser útil en las API. Las desventajas de usar `shared_ptr` sobre `unique_ptr` incluyen un mayor costo de memoria para almacenar el eliminador y un costo de rendimiento para mantener el recuento de referencia.

```
// deleter required at construction time and is part of the type
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);

// deleter is only required at construction time, not part of the type
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

C ++ 17

Con la `template auto` , podemos hacer que sea aún más fácil envolver nuestros borrados personalizados:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) {
        DeleteFn(ptr);
    }
};

template <class T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

Con lo que el ejemplo anterior es simplemente:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Aquí, el propósito de `auto` es manejar todas las funciones libres, ya sea que devuelvan `void` (por ejemplo, `SDL_FreeSurface`) o no (por ejemplo, `fclose`).

Propiedad única sin semántica de movimiento (auto_ptr)

C ++ 11

NOTA: `std::auto_ptr` ha quedado en desuso en C ++ 11 y se eliminará en C ++ 17. Solo debe usar esto si está obligado a usar C ++ 03 o anterior y está dispuesto a ser cuidadoso. Se recomienda pasar a unique_ptr en combinación con `std::move` para reemplazar el comportamiento de `std::auto_ptr` .

Antes de tener `std::unique_ptr` , antes de mover la semántica, teníamos `std::auto_ptr` . `std::auto_ptr` proporciona propiedad única pero transfiere la propiedad en la copia.

Al igual que con todos los punteros inteligentes, `std::auto_ptr` limpia automáticamente los recursos (ver RAII):

```
{
    std::auto_ptr<int> p(new int(42));
```

```
    std::cout << *p;
} // p is deleted here, no memory leaked
```

pero solo permite un dueño:

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
  // px is now empty
```

Esto permite usar std :: auto_ptr para mantener la propiedad explícita y única ante el peligro de perder la propiedad sin querer:

```
void f(std::auto_ptr<X> ) {
    // assumes ownership of X
    // deletes it at end of scope
};

std::auto_ptr<X> px = ...;
f(px); // f acquires ownership of underlying X
       // px is now empty
px->foo(); // NPE!
// px.~auto_ptr() does NOT delete
```

La transferencia de propiedad ocurrió en el constructor "copia". El constructor de copia y el operador de asignación de copia de `auto_ptr` toman sus operandos por referencia no `const` para que puedan ser modificados. Una implementación de ejemplo podría ser:

```
template <typename T>
class auto_ptr {
    T* ptr;
public:
    auto_ptr(auto_ptr& rhs)
    : ptr(rhs.release())
    { }

    auto_ptr& operator=(auto_ptr& rhs) {
        reset(rhs.release());
        return *this;
    }

    T* release() {
        T* tmp = ptr;
        ptr = nullptr;
        return tmp;
    }

    void reset(T* tmp = nullptr) {
        if (ptr != tmp) {
            delete ptr;
            ptr = tmp;
        }
    }

    /* other functions ... */
};
```

Esto rompe la semántica de la copia, que requiere que copiar un objeto te deje con dos versiones equivalentes. Para cualquier tipo de copia, `T` , debería poder escribir:

```
T a = ...;
T b(a);
assert(b == a);
```

Pero para `auto_ptr` , este no es el caso. Como resultado, no es seguro poner `auto_ptr` s en contenedores.

Consiguiendo un share_ptr refiriéndose a esto

`enable_shared_from_this` permite obtener una instancia de `shared_ptr` válida para `this` .

Al derivar su clase de la plantilla de clase `enable_shared_from_this` , usted hereda un método `shared_from_this` que devuelve una instancia de `shared_ptr` a `this` .

Tenga en cuenta que el objeto debe crearse como `shared_ptr` en primer lugar:

```
#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: pointing to the same object
```

Nota (2) no puede llamar a `enable_shared_from_this` dentro del constructor.

```
#include <memory> // enable_shared_from_this

class Widget : public std::enable_shared_from_this< Widget >
{
public:
    void DoSomething()
    {
        std::shared_ptr< Widget > self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared< Widget >();
    w -> DoSomething();
    ...
}
```

Si usa `shared_from_this()` en un objeto que no es propiedad de `shared_ptr` , como un objeto automático local o un objeto global, el comportamiento no está definido. Desde C ++ 17 lanza

`std::bad_alloc` en `std::bad_alloc` lugar.

El uso de `shared_from_this()` de un constructor es equivalente a usarlo en un objeto que no pertenece a `shared_ptr` , porque los objetos son poseídos por `shared_ptr` después de que el constructor regrese.

Casting std :: shared_ptr pointers

No es posible usar directamente `static_cast` , `const_cast` , `dynamic_cast` y `reinterpret_cast` en `std::shared_ptr` para recuperar un puntero que comparte la propiedad con el puntero que se pasa como argumento. En su lugar, deben utilizarse las funciones `std::static_pointer_cast` , `std::const_pointer_cast` , `std::dynamic_pointer_cast` y `std::reinterpret_pointer_cast` :

```
struct Base { virtual ~Base() noexcept {}; };
struct Derived: Base {};
auto derivedPtr(std::make_shared<Derived>());
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Tenga en cuenta que `std::reinterpret_pointer_cast` no está disponible en C ++ 11 y C ++ 14, ya que solo fue propuesto por N3920 y adoptado en Library Fundamentals TS en febrero de 2014 . Sin embargo, se puede implementar de la siguiente manera:

```
template <typename To, typename From>
inline std::shared_ptr<To> reinterpret_pointer_cast(
    std::shared_ptr<From> const & ptr) noexcept
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Escribiendo un puntero inteligente: value_ptr

Un `value_ptr` es un puntero inteligente que se comporta como un valor. Cuando se copia, copia su contenido. Cuando se crea, crea su contenido.

```
// Like std::default_delete:
template<class T>
struct default_copier {
  // a copier must handle a null T const* in and return null:
  T* operator()(T const* tin)const {
    if (!tin) return nullptr;
    return new T(*tin);
  }
  void operator()(void* dest, T const* tin)const {
    if (!tin) return;
    return new(dest) T(*tin);
  }
};
// tag class to handle empty case:
struct empty_ptr_t {};
constexpr empty_ptr_t empty_ptr{};
// the value pointer type itself:
template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
  class Base=std::unique_ptr<T, Deleter>
```

```
>
struct value_ptr:Base, private Copier {
  using copier_type=Copier;
  // also typedefs from unique_ptr

  using Base::Base;

  value_ptr( T const& t ):
    Base( std::make_unique<T>(t) ),
    Copier()
  {}
  value_ptr( T && t ):
    Base( std::make_unique<T>(std::move(t)) ),
    Copier()
  {}
  // almost-never-empty:
      value_ptr():
    Base( std::make_unique<T>() ),
    Copier()
  {}
  value_ptr( empty_ptr_t ) {}

  value_ptr( Base b, Copier c={} ):
    Base(std::move(b)),
    Copier(std::move(c))
  {}

  Copier const& get_copier() const {
    return *this;
  }

  value_ptr clone() const {
    return {
      Base(
        get_copier()(this->get()),
        this->get_deleter()
      ),
      get_copier()
    };
  }
  value_ptr(value_ptr&&)=default;
  value_ptr& operator=(value_ptr&&)=default;

  value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
  value_ptr& operator=(value_ptr const&o) {
    if (o && *this) {
      // if we are both non-null, assign contents:
      **this = *o;
    } else {
      // otherwise, assign a clone (which could itself be null):
      *this = o.clone();
    }
    return *this;
  }
  value_ptr& operator=( T const& t ) {
    if (*this) {
      **this = t;
    } else {
      *this = value_ptr(t);
    }
    return *this;
```

```
  }
  value_ptr& operator=( T && t ) {
    if (*this) {
      **this = std::move(t);
    } else {
      *this = value_ptr(std::move(t));
    }
    return *this;
  }
  T& get() { return **this; }
  T const& get() const { return **this; }
  T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  // operator-> from unique_ptr
};
template<class T, class...Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
  return {std::make_unique<T>(std::forward<Args>(args)...)};
}
```

Este valor_ptr en particular solo está vacío si lo construyes con `empty_ptr_t` o si te mueves de él. Expone el hecho de que es un `unique_ptr` , por `explicit operator bool() const` trabaja en él. `.get()` se ha cambiado para devolver una referencia (ya que casi nunca está vacía), y `.get_pointer()` devuelve un puntero.

Este puntero inteligente puede ser útil para los casos `pImpl` , donde queremos valores semánticos, pero tampoco queremos exponer los contenidos de `pImpl` fuera del archivo de implementación.

Con una `Copier` no predeterminada, incluso puede manejar clases de base virtual que saben cómo producir instancias de sus derivadas y convertirlas en tipos de valor.

Lea Punteros inteligentes en línea: https://riptutorial.com/es/cplusplus/topic/509/punteros-inteligentes

Capítulo 108: RAII: la adquisición de recursos es la inicialización

Observaciones

RAII significa **R** isource **A** cquisition **I** s **I** nitialization. RAII es una expresión idiomática utilizada para vincular los recursos con la vida útil de los objetos. También se conoce como SBRM (Administración de recursos basada en el alcance) o RRID (La publicación del recurso es destrucción). En C ++, el destructor para un objeto siempre se ejecuta cuando un objeto se sale del alcance, podemos aprovechar eso para vincular la limpieza de recursos con la destrucción de objetos.

Cada vez que necesite adquirir algún recurso (por ejemplo, un bloqueo, un identificador de archivo, un búfer asignado) que eventualmente necesitará liberar, debe considerar usar un objeto para manejar esa administración de recursos por usted. El desenrollado de la pila ocurrirá independientemente de la excepción o la salida temprana del alcance, por lo que el objeto del controlador de recursos limpiará el recurso por usted sin que tenga que considerar cuidadosamente todas las rutas de código actuales y futuras posibles.

Vale la pena señalar que RAII no libera completamente al desarrollador de pensar en la vida útil de los recursos. Un caso es, obviamente, una llamada crash o exit (), que evitará que se llame a los destructores. Como el sistema operativo limpiará los recursos locales del proceso, como la memoria, después de que finalice el proceso, esto no es un problema en la mayoría de los casos. Sin embargo, con los recursos del sistema (es decir, canalizaciones con nombre, archivos de bloqueo, memoria compartida), todavía se necesitan recursos para tratar el caso en el que un proceso no se limpió después de sí mismo, es decir, en la prueba de inicio si el archivo de bloqueo está ahí, si es así, Verifique que el proceso con el pid realmente existe, luego actúe en consecuencia.

Otra situación es cuando un proceso de Unix llama a una función de la familia exec, es decir, después de un fork-exec para crear un nuevo proceso. Aquí, el proceso hijo tendrá una copia completa de la memoria de los padres (incluidos los objetos RAII), pero una vez que se llamó a exec, no se llamará a ninguno de los destructores en ese proceso. Por otro lado, si un proceso se bifurca y ninguno de los procesos llama exec, todos los recursos se limpian en ambos procesos. Esto es correcto solo para todos los recursos que realmente se duplicaron en la bifurcación, pero con los recursos del sistema, ambos procesos solo tendrán una referencia al recurso (es decir, la ruta de acceso a un archivo de bloqueo) y ambos intentarán liberarlo individualmente, lo que podría causar El otro proceso para fallar.

Examples

Cierre

Bloqueo incorrecto:

```
std::mutex mtx;

void bad_lock_example() {
    mtx.lock();
    try
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock();   // Have to unlock on each exit point.
            return;
        }
        quux();
        mtx.unlock();       // Normal unlock happens here.
    }
    catch(...) {
        mtx.unlock();       // Must also force unlock in the presence of
        throw;              // exceptions and allow the exception to continue.
    }
}
```

Esa es la forma incorrecta de implementar el bloqueo y desbloqueo del mutex. Para garantizar la correcta liberación del mutex con `unlock()` , el programador debe asegurarse de que todos los flujos que resultan en la salida de la función resulten en una llamada a `unlock()` . Como se muestra arriba, este es un proceso frágil ya que requiere que los mantenedores continúen siguiendo el patrón manualmente.

El uso de una clase diseñada apropiadamente para implementar RAII, el problema es trivial:

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx);   // constructor locks.
                                           // destructor unlocks. destructor call
                                           // guaranteed by language.
    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

`lock_guard` es una plantilla de clase extremadamente simple que simplemente llama a `lock()` en su argumento en su constructor, mantiene una referencia al argumento y llama a `unlock()` en el argumento en su destructor. Es decir, cuando el `lock_guard` queda fuera del alcance, se garantiza que el `mutex` está desbloqueado. No importa si la razón por la que se salió del alcance es una excepción o una devolución anticipada: todos los casos se manejan; independientemente del flujo de control, garantizamos que desbloquearemos correctamente.

Finalmente / ScopeExit

Para los casos en que no queremos escribir clases especiales para manejar algún recurso, podemos escribir una clase genérica:

```
template<typename Function>
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) See below

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;
private:
    Function f;
};
// Execute the function f when the returned object goes out of scope.
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)};
}
```

Y su uso de ejemplo.

```
void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&](){ v[i] -= 42; });

    // ... code as recursive call `foo(v, i + 1)`
}
```

Nota (1): Debe considerarse alguna discusión sobre la definición del destructor para manejar la excepción:

- `~Finally() noexcept { f(); }` : `std::terminate` se llama en caso de excepción
- `~Finally() noexcept(noexcept(f())) { f(); }` : terminate () se llama solo en caso de excepción durante el desenrollado de la pila.
- `~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */} }` No se llama a `std::terminate` , pero no podemos manejar el error (incluso para el no desenrollado de la pila).

ScopeSuccess (c ++ 17)

C ++ 17

Gracias a `int std::uncaught_exceptions()` , podemos implementar acciones que se ejecutan solo en caso de éxito (no se produce ninguna excepción en el alcance). Anteriormente, `bool std::uncaught_exception()` solo permite detectar si se está ejecutando **algún** desenrollado de pila.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeSuccess
```

```
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() might throw, as it can be caught normally.
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Salida:

```
Success 1
```

ScopeFail (c ++ 17)

C ++ 17

Gracias a `int std::uncaught_exceptions()` , podemos implementar una acción que se ejecuta solo en caso de fallo (se produce una excepción en el alcance). Anteriormente, `bool`

`std::uncaught_exception()` solo permite detectar si se está ejecutando **algún** desenrollado de pila.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() should not throw, else std::terminate is called.
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "Fail 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "Failure 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Salida:

```
Failure 2
```

Lea RAII: la adquisición de recursos es la inicialización en línea:
https://riptutorial.com/es/cplusplus/topic/1320/raii--la-adquisicion-de-recursos-es-la-inicializacion

Capítulo 109: Recursion en C ++

Examples

Uso de la recursión de la cola y la recursión del estilo de Fibonnaci para resolver la secuencia de Fibonnaci

La forma más simple y más obvia de usar la recursión para obtener el enésimo término de la secuencia de Fibonnaci es esta

```
int get_term_fib(int n)
{
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return get_term_fib(n - 1) + get_term_fib(n - 2);
}
```

Sin embargo, este algoritmo no se escala para términos más altos: para `n` más grande y más grande, la cantidad de llamadas a funciones que necesita realizar crece exponencialmente. Esto puede ser reemplazado con una simple recursión de cola.

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
  if (n == 0)
    return prev;
  if (n == 1)
    return curr;
  return get_term_fib(n - 1, curr, prev + curr);
}
```

Cada llamada a la función ahora calcula inmediatamente el siguiente término en la secuencia de Fibonnaci, por lo que el número de llamadas a función se escala linealmente con `n` .

Recursion con memoizacion.

Las funciones recursivas pueden ser bastante caras. Si son funciones puras (funciones que siempre devuelven el mismo valor cuando se las llama con los mismos argumentos, y que no dependen ni modifican el estado externo), pueden hacerse considerablemente más rápido a expensas de la memoria almacenando los valores ya calculados.

La siguiente es una implementación de la secuencia de Fibonacci con memoización:

```
#include <map>

int fibonacci(int n)
{
  static std::map<int, int> values;
```

```
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Tenga en cuenta que a pesar de usar la fórmula de recursión simple, en la primera llamada esta función es $ O (n) $. En llamadas subsiguientes con el mismo valor, es por supuesto $ O (1) $.

Tenga en cuenta, sin embargo, que esta implementación no es reentrante. Además, no permite deshacerse de los valores almacenados. Una implementación alternativa sería permitir que el mapa se pase como argumento adicional:

```
#include <map>

int fibonacci(int n, std::map<int, int> values)
{
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Para esta versión, se requiere que la persona que llama mantenga el mapa con los valores almacenados. Esto tiene la ventaja de que la función ahora está reentrada y que la persona que llama puede eliminar valores que ya no son necesarios, lo que ahorra memoria. Tiene la desventaja de que rompe la encapsulación; la persona que llama puede cambiar la salida al rellenar el mapa con valores incorrectos.

Lea Recursion en C ++ en línea: https://riptutorial.com/es/cplusplus/topic/5693/recursion-en-c-plusplus

Capítulo 110: Reenvío perfecto

Observaciones

El reenvío perfecto requiere *reenviar las referencias* para preservar los ref-calificadores de los argumentos. Tales referencias aparecen solo en un *contexto deducido* . Es decir:

```
template<class T>
void f(T&& x) // x is a forwarding reference, because T is deduced from a call to f()
{
    g(std::forward<T>(x)); // g() will receive an lvalue or an rvalue, depending on x
}
```

Lo siguiente no implica un reenvío perfecto, porque `T` no se deduce de la llamada del constructor:

```
template<class T>
struct a
{
    a(T&& x); // x is a rvalue reference, not a forwarding reference
};
```

C ++ 17

C ++ 17 permitirá la deducción de argumentos de plantilla de clase. El constructor de "a" en el ejemplo anterior se convertirá en un usuario de una referencia de reenvío

```
a example1(1);
  // same as a<int> example1(1);

int x = 1;
a example2(x);
  // same as a<int&> example2(x);
```

Examples

Funciones de fábrica

Supongamos que queremos escribir una función de fábrica que acepte una lista arbitraria de argumentos y pase esos argumentos sin modificar a otra función. Un ejemplo de una función de este tipo es `make_unique` , que se utiliza para construir de forma segura una nueva instancia de `T` y devolver un `unique_ptr<T>` que posee la instancia.

Las reglas de lenguaje con respecto a las plantillas variadas y las referencias de valores nos permiten escribir dicha función.

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
{
```

```
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

El uso de puntos suspensivos `...` indica un paquete de parámetros, que representa un número arbitrario de tipos. El compilador expandirá este paquete de parámetros al número correcto de argumentos en el sitio de la llamada. Estos argumentos luego se pasan al constructor de `T` usando `std::forward` . Esta función es necesaria para conservar los ref-calificadores de los argumentos.

```
struct foo
{
    foo() {}
    foo(const foo&) {}                    // copy constructor
    foo(foo&&) {}                         // copy constructor
    foo(int, int, int) {}
};

foo f;
auto p1 = make_unique<foo>(f);            // calls foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // calls foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

Lea Reenvío perfecto en línea: https://riptutorial.com/es/cplusplus/topic/1750/reenvio-perfecto

Capítulo 111: Referencias

Examples

Definiendo una referencia

Las referencias se comportan de manera similar, pero no del todo como punteros const. Una referencia se define mediante el sufijo de un signo `&` a un nombre de tipo.

```
int i = 10;
int &refi = i;
```

Aquí, `refi` es una referencia vinculada a `i` .
Referencias resume la semántica de los punteros, actuando como un alias para el objeto subyacente:

```
refi = 20; // i = 20;
```

También puede definir múltiples referencias en una sola definición:

```
int i = 10, j = 20;
int &refi = i, &refj = j;

// Common pitfall :
// int& refi = i, k = j;
// refi will be of type int&.
// though, k will be of type int, not int&!
```

Las referencias **deben** inicializarse correctamente en el momento de la definición y no pueden modificarse posteriormente. La siguiente pieza de códigos provoca un error de compilación:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

Tampoco puede enlazar directamente una referencia a `nullptr` , a diferencia de los punteros:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a
temporary of type 'nullptr_t'
```

Las referencias de C ++ son alias de variables existentes

Una referencia en C ++ es solo un `Alias` u otro nombre de una variable. Al igual que la mayoría de nosotros, podemos referirnos usando nuestro nombre de pasaporte y nuestro nombre de usuario.

Las referencias no existen literalmente y no ocupan ningún recuerdo. Si imprimimos la dirección de la variable de referencia, se imprimirá la misma dirección que la de la variable a la que hace referencia.

```
int main() {
    int i = 10;
    int &j = i;

    cout<<&i<<endl;
    cout<<&b<<endl;
    return 0;
}
```

En el ejemplo anterior, ambos `cout` imprimirán la misma dirección. La situación será la misma si tomamos una variable como referencia en una función.

```
void func (int &fParam ) {
   cout<<"Address inside function => "<<fParam<<endl;
}

int main() {
    int i = 10;
    cout<<"Address inside Main => "<<&i<<endl;

    func(i);

    return 0;
}
```

También en este ejemplo, ambos `cout` imprimirán la misma dirección.

Como sabemos ahora que las `C++ References` son solo un alias, y para que se cree un alias, necesitamos tener algo a lo que el Alias pueda referirse.

Esa es la razón precisa por la que la declaración como esta arrojará un error de compilación

```
int &i;
```

Porque, el alias no se refiere a nada.

Lea Referencias en línea: https://riptutorial.com/es/cplusplus/topic/1548/referencias

Capítulo 112: Regla de una definición (ODR)

Examples

Función multiplicada definida

La consecuencia más importante de la regla de definición única es que las funciones no en línea con enlace externo solo deben definirse una vez en un programa, aunque se pueden declarar varias veces. Por lo tanto, tales funciones no deben definirse en encabezados, ya que un encabezado puede incluirse varias veces desde diferentes unidades de traducción.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() { std:: cout << "bar"; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

En este programa, la función `foo` se define en el encabezado `foo.h` , que se incluye dos veces: una vez desde `foo.cpp` y una vez desde `main.cpp` . Por lo tanto, cada unidad de traducción contiene su propia definición de `foo` . Tenga en cuenta que los guardias de `foo.h` en `foo.h` no evitan que esto suceda, ya que tanto `foo.cpp` como `main.cpp` incluyen `foo.h` *separado* . El resultado más probable de intentar construir este programa es un error de tiempo de enlace que identifica a `foo` como si se hubiera definido de forma múltiple.

Para evitar tales errores, se deben *declarar* funciones en los encabezados y *definirlas* en los archivos `.cpp` correspondientes, con algunas excepciones (ver otros ejemplos).

Funciones en linea

Una función declarada en `inline` se puede definir en múltiples unidades de traducción, siempre que todas las definiciones sean idénticas. También debe definirse en cada unidad de traducción

en la que se utiliza. Por lo tanto, las funciones en línea *deben* definirse en los encabezados y no es necesario mencionarlas en el archivo de implementación.

El programa se comportará como si hubiera una sola definición de la función.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() {
    // more complicated definition
}
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

En este ejemplo, la función más simple `foo` se define en línea en el archivo de encabezado, mientras que la `bar` funciones más complicada no está en línea y se define en el archivo de implementación. Las unidades de traducción `foo.cpp` y `main.cpp` contienen definiciones de `foo` , pero este programa está bien formado ya que `foo` está en línea.

Una función definida dentro de una definición de clase (que puede ser una función miembro o una función amiga) está *implícitamente en* línea. Por lo tanto, si una clase se define en un encabezado, las funciones miembro de la clase se pueden definir dentro de la definición de la clase, aunque las definiciones se pueden incluir en varias unidades de traducción:

```
// in foo.h
class Foo {
    void bar() { std::cout << "bar"; }
    void baz();
};

// in foo.cpp
void Foo::baz() {
   // definition
}
```

La función `Foo::baz` se define fuera de línea, por lo que *no* es una función en línea y no debe definirse en el encabezado.

Violación ODR a través de la resolución de sobrecarga

Incluso con tokens idénticos para funciones en línea, se puede violar la ODR si la búsqueda de nombres no se refiere a la misma entidad. consideremos la `func` en lo siguiente:

- header.h

```
void overloaded(int);
inline void func() { overloaded('*'); }
```

- foo.cpp

```
#include "header.h"

void foo()
{
    func(); // `overloaded` refers to `void overloaded(int)`
}
```

- bar.cpp

```
void overloaded(char); // can come from other include
#include "header.h"

void bar()
{
    func(); // `overloaded` refers to `void overloaded(char)`
}
```

Tenemos una violación de ODR ya que `overloaded` refiere a diferentes entidades dependiendo de la unidad de traducción.

Lea Regla de una definición (ODR) en línea: https://riptutorial.com/es/cplusplus/topic/4907/regla-de-una-definicion--odr-

Capítulo 113: Resolución de sobrecarga

Observaciones

La resolución de sobrecarga ocurre en varias situaciones diferentes

- Llamadas a funciones sobrecargadas con nombre. Los candidatos son todas las funciones encontradas por búsqueda de nombre.
- Llamadas a objeto de clase. Los candidatos suelen ser todos los operadores de llamada de función sobrecargados de la clase.
- Uso de un operador. Los candidatos son las funciones de operador sobrecargadas en el ámbito del espacio de nombres, las funciones de operador sobrecargadas en el objeto de la clase izquierda (si existe) y los operadores integrados.
- Resolución de sobrecarga para encontrar la función de operador de conversión correcta o el constructor a invocar para una inicialización
    - Para la inicialización directa no de lista (`Class c(value)`), los candidatos son constructores de `Class` .
    - Para la inicialización de copia no de lista (`Class c = value`) y para encontrar la función de conversión definida por el usuario para invocar en una secuencia de conversión definida por el usuario. Los candidatos son los constructores de `Class` y si la fuente es un objeto de clase, su operador de conversión funciona.
    - Para la inicialización de una no clase desde un objeto de clase (`Nonclass c = classObject`). Los candidatos son las funciones del operador de conversión del objeto inicializador.
    - Para inicializar una referencia con un objeto de clase (`R &r = classObject`), cuando la clase tiene funciones de operador de conversión que producen valores que pueden vincularse directamente a `r` . Los candidatos son tales funciones de operador de conversión.
    - Para la inicialización de lista de un objeto de clase no agregado (`Class c{1, 2, 3}`), los candidatos son los constructores de la lista de inicializadores para una primera pasada a través de la resolución de sobrecarga. Si esto no encuentra un candidato viable, se realiza un segundo paso a través de la resolución de sobrecarga, con los constructores de `Class` como candidatos.

Examples

Coincidencia exacta

Una sobrecarga sin conversiones necesarias para tipos de parámetros o solo conversiones necesarias entre tipos que aún se consideran coincidencias exactas es preferible a una sobrecarga que requiere otras conversiones para poder llamar.

```
void f(int x);
void f(double x);
f(42); // calls f(int)
```

Cuando un argumento se enlaza con una referencia del mismo tipo, se considera que la coincidencia no requiere una conversión, incluso si la referencia es más calificada como CV.

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // argument type is int; exact match with int&

void g(const int& x);
void g(int x);
g(x); // ambiguous; both overloads give exact match
```

A los efectos de la resolución de sobrecarga, se considera que el tipo "matriz de `T` " coincide exactamente con el tipo "puntero a `T` ", y se considera que la función tipo `T` coincide exactamente con el tipo de indicador de función `T*` , aunque ambos requieren conversiones

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // calls f(int*); exact match with array-to-pointer conversion
g(a); // ambiguous; both overloads give exact match
```

Categorización de argumento a costo de parámetro

La resolución de sobrecarga divide el costo de pasar un argumento a un parámetro en una de cuatro categorías diferentes, llamadas "secuencias". Cada secuencia puede incluir cero, una o varias conversiones.

- Secuencia de conversión estándar

  ```
  void f(int a); f(42);
  ```

- Secuencia de conversión definida por el usuario

  ```
  void f(std::string s); f("hello");
  ```

- Secuencia de conversión elipsis

  ```
  void f(...); f(42);
  ```

- Secuencia de inicialización de lista

  ```
  void f(std::vector<int> v); f({1, 2, 3});
  ```

El principio general es que las secuencias de conversión estándar son las más baratas, seguidas de las secuencias de conversión definidas por el usuario, seguidas de las secuencias de

conversión de puntos suspensivos.

Un caso especial es la secuencia de inicialización de lista, que no constituye una conversión (una lista de inicializador no es una expresión con un tipo). Su costo se determina definiéndolo como equivalente a una de las otras tres secuencias de conversión, según el tipo de parámetro y la forma de la lista de inicializadores.

Búsqueda de nombres y verificación de acceso

La resolución de sobrecarga ocurre *después de la* búsqueda del nombre. Esto significa que no se seleccionará una función de mejor coincidencia por resolución de sobrecarga si pierde búsqueda de nombre:

```
void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // calls S::f because global f is not visible here,
                        // even though it would be a better match
};
```

La resolución de sobrecarga ocurre *antes de la* verificación de acceso. Una resolución inaccesible puede seleccionarse por resolución de sobrecarga si es una mejor coincidencia que una función accesible.

```
class C {
  public:
    static void f(double x);
  private:
    static void f(int x);
};
C::f(42); // Error! Calls private C::f(int) even though public C::f(double) is viable.
```

De manera similar, la resolución de sobrecarga ocurre sin verificar si la llamada resultante está bien formada con respecto a lo `explicit` :

```
struct X {
    explicit X(int );
    X(char );
};

void foo(X );
foo({4}); // X(int) is better much, but expression is
          // ill-formed because selected constructor is explicit
```

Sobrecarga en la referencia de reenvío

Debe tener mucho cuidado al proporcionar una sobrecarga de referencia de reenvío, ya que puede coincidir demasiado bien:

```
struct A {
    A() = default;           // #1
```

```
    A(A const& ) = default;  // #2

    template <class T>
    A(T&& );                 // #3
};
```

La intención aquí era que `A` es copiable, y que tenemos este otro constructor que podría inicializar a otro miembro. Sin embargo:

```
A a;     // calls #1
A b(a);  // calls #3!
```

Hay dos partidos viables para la convocatoria de construcción:

```
A(A const& ); // #2
A(A& );       // #3, with T = A&
```

Ambas son coincidencias exactas, pero `#3` toma una referencia a un objeto calificado menos *cv* que `#2` , por lo que tiene la mejor secuencia de conversión estándar y es la mejor función viable.

La solución aquí es restringir siempre estos constructores (por ejemplo, utilizando SFINAE):

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
    >
A(T&& );
```

El rasgo de tipo aquí es excluir de su consideración cualquier `A` o clase derivada pública y sin ambigüedad de `A` , lo que haría que este constructor no se formara correctamente en el ejemplo descrito anteriormente (y, por lo tanto, eliminado del conjunto de sobrecarga). Como resultado, se invoca el constructor de copia, que es lo que queríamos.

Pasos de resolución de sobrecarga

Los pasos de resolución de sobrecarga son:

1. Encuentra funciones candidatas a través de búsqueda de nombre. Las llamadas no calificadas realizarán una búsqueda regular no calificada, así como una búsqueda dependiente del argumento (si corresponde).

2. Filtrar el conjunto de funciones candidatas a un conjunto de funciones *viables* . Una función viable para la que existe una secuencia de conversión implícita entre los argumentos con los que se llama la función y los parámetros que toma la función.

   ```
   void f(char);          // (1)
   void f(int ) = delete; // (2)
   void f();              // (3)
   void f(int& );         // (4)

   f(4); // 1,2 are viable (even though 2 is deleted!)
   ```

```
      // 3 is not viable because the argument lists don't match
      // 4 is not viable because we cannot bind a temporary to
      //     a non-const lvalue reference
```

3. Elige el mejor candidato viable. Una función viable `F1` es una función mejor que otra función viable `F2` si la secuencia de conversión implícita para cada argumento en `F1` no es peor que la secuencia de conversión implícita correspondiente en `F2` , y ...:

   3.1. Para algunos argumentos, la secuencia de conversión implícita para ese argumento en `F1` es una mejor secuencia de conversión que para ese argumento en `F2` , o

```
void f(int );  // (1)
void f(char ); // (2)

f(4);  // call (1), better conversion sequence
```

   3.2. En una conversión definida por el usuario, la secuencia de conversión estándar desde el retorno de `F1` al tipo de destino es una mejor secuencia de conversión que la del tipo de retorno de `F2` , o

```
struct A
{
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

   3.3. En una vinculación de referencia directa, `F1` tiene el mismo tipo de referencia que `F2` , o

```
struct A
{
    operator X&();  // #1
    operator X&&(); // #2
};
A a;
X& lx = a;  // calls #1
X&& rx = a; // calls #2
```

   3.4. `F1` no es una especialización de plantilla de función, pero `F2` es, o

```
template <class T> void f(T ); // #1
void f(int );                  // #2

f(42); // calls #2, the non-template
```

   3.5. `F1` y `F2` son ambas especializaciones de plantilla de función, pero `F1` es más especializado que `F2` .

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2
```

```
int* p;
f(p); // calls #2, more specialized
```

El ordenamiento aquí es significativo. La mejor verificación de la secuencia de conversión ocurre antes de la verificación de la plantilla frente a la no-plantilla. Esto conduce a un error común con la sobrecarga en la referencia de reenvío:

```
struct A {
    A(A const& ); // #1

    template <class T>
    A(T&& );      // #2, not constrained
};

A a;
A b(a); // calls #2!
        // #1 is not a template but #2 resolves to
        // A(A& ), which is a less cv-qualified reference than #1
        // which makes it a better implicit conversion sequence
```

Si no hay un mejor candidato viable al final, la llamada es ambigua:

```
void f(double ) { }
void f(float ) { }

f(42); // error: ambiguous
```

Promociones y conversiones aritméticas.

Convertir un tipo entero en el tipo promovido correspondiente es mejor que convertirlo en otro tipo entero.

```
void f(int x);
void f(short x);
signed char c = 42;
f(c); // calls f(int); promotion to int is better than conversion to short
short s = 42;
f(s); // calls f(short); exact match is better than promotion to int
```

Promover un `float` para `double` es mejor que convertirlo en algún otro tipo de punto flotante.

```
void f(double x);
void f(long double x);
f(3.14f); // calls f(double); promotion to double is better than conversion to long double
```

Las conversiones aritméticas distintas de las promociones no son mejores ni peores que las otras.

```
void f(float x);
void f(long double x);
f(3.14); // ambiguous
```

```
void g(long x);
void g(long double x);
g(42); // ambiguous
g(3.14); // ambiguous
```

Por lo tanto, para garantizar que no haya ambigüedad al llamar a una función `f` con argumentos integrales o de punto flotante de cualquier tipo estándar, se necesitan un total de ocho sobrecargas, por lo que para cada tipo de argumento posible, ya sea una sobrecarga Se seleccionará exactamente o la sobrecarga única con el tipo de argumento promovido.

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

Sobrecarga dentro de una jerarquía de clases

Los siguientes ejemplos utilizarán esta jerarquía de clases:

```
struct A { int m; };
struct B : A {};
struct C : B {};
```

La conversión del tipo de clase derivada al tipo de clase base se prefiere a las conversiones definidas por el usuario. Esto se aplica cuando se pasa por valor o por referencia, así como cuando se convierte puntero a derivado en puntero a base.

```
struct Unrelated {
    Unrelated(B b);
};
void f(A a);
void f(Unrelated u);
B b;
f(b); // calls f(A)
```

Una conversión de puntero de clase derivada a clase base también es mejor que la conversión a `void*`.

```
void f(A* p);
void f(void* p);
B b;
f(&b); // calls f(A*)
```

Si hay múltiples sobrecargas dentro de la misma cadena de herencia, se prefiere la sobrecarga de clase base más derivada. Esto se basa en un principio similar al envío virtual: se elige la implementación "más especializada". Sin embargo, la resolución de sobrecarga siempre se

produce en el momento de la compilación y nunca se convertirá de forma implícita.

```
void f(const A& a);
void f(const B& b);
C c;
f(c); // calls f(const B&)
B b;
A& r = b;
f(r); // calls f(const A&); the f(const B&) overload is not viable
```

Para los punteros a los miembros, que son contravariantes con respecto a la clase, una regla similar se aplica en la dirección opuesta: se prefiere la clase derivada menos derivada.

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

Sobrecarga en constness y volatilidad.

Pasar un argumento de puntero a un parámetro `T*` , si es posible, es mejor que pasarlo a un parámetro `const T*` .

```
struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) is better than f(const Base*)
Derived d;
f(&d); // f(const Derived*) is better than f(Base*) though;
       // constness is only a "tie-breaker" rule
```

Del mismo modo, pasar un argumento a un parámetro `T&` , si es posible, es mejor que pasarlo a un parámetro `const T&` , incluso si ambos tienen rango de coincidencia exacta.

```
void f(int& r);
void f(const int& r);
int x;
f(x); // both overloads match exactly, but f(int&) is still better
const int y = 42;
f(y); // only f(const int&) is viable
```

Esta regla se aplica también a las funciones miembro calificadas por const, donde es importante para permitir el acceso mutable a objetos no constantes y el acceso inmutable a objetos const.

```
class IntVector {
  public:
    // ...
    int* data() { return m_data; }
```

```
    const int* data() const { return m_data; }
  private:
    // ...
    int* m_data;
};
IntVector v1;
int* data1 = v1.data();       // Vector::data() is better than Vector::data() const;
                              // data1 can be used to modify the vector's data
const IntVector v2;
const int* data2 = v2.data(); // only Vector::data() const is viable;
                              // data2 can't be used to modify the vector's data
```

De la misma manera, una sobrecarga volátil será menos preferida que una sobrecarga no volátil.

```
class AtomicInt {
  public:
    // ...
    int load();
    int load() volatile;
  private:
    // ...
};
AtomicInt a1;
a1.load(); // non-volatile overload preferred; no side effect
volatile AtomicInt a2;
a2.load(); // only volatile overload is viable; side effect
static_cast<volatile AtomicInt&>(a1).load(); // force volatile semantics for a1
```

Lea Resolución de sobrecarga en línea: https://riptutorial.com/es/cplusplus/topic/2021/resolucion-de-sobrecarga

Capítulo 114: RTTI: Información de tipo de tiempo de ejecución

Examples

Nombre de un tipo

Puede recuperar el nombre definido por la implementación de un tipo en tiempo de ejecución utilizando la función miembro `.name()` del objeto `std::type_info` devuelto por `typeid` .

```
#include <iostream>
#include <typeinfo>

int main()
{
    int speed = 110;

    std::cout << typeid(speed).name() << '\n';
}
```

Salida (definida por la implementación):

```
int
```

dynamic_cast

Utilice `dynamic_cast<>()` como una función, que le ayuda a reducir a través de una jerarquía de herencia (descripción principal).

Si debe realizar algún trabajo no polimórfico en algunas clases derivadas `B` y `C` , pero recibió la `class A` base `class A` , escriba de esta forma:

```
class A { public: virtual ~A(){} };

class B: public A
{ public: void work4B(){} };

class C: public A
{ public: void work4C(){} };

void non_polymorphic_work(A* ap)
{
  if (B* bp =dynamic_cast<B*>(ap))
    bp->work4B();
  if (C* cp =dynamic_cast<C*>(ap))
    cp->work4C();
}
```

La palabra clave typeid

La palabra clave `typeid` es un operador unario que proporciona información de tipo de tiempo de ejecución sobre su operando si el tipo de operando es un tipo de clase polimórfica. Devuelve un lvalue de tipo `const std::type_info` . Se ignora la calificación cv de nivel superior.

```
struct Base {
    virtual ~Base() = default;
};
struct Derived : Base {};
Base* b = new Derived;
assert(typeid(*b) == typeid(Derived{})); // OK
```

`typeid` también se puede aplicar a un tipo directamente. En este caso, las primeras referencias de nivel superior se eliminan, luego se ignora la calificación cv de nivel superior. Por lo tanto, el ejemplo anterior podría haberse escrito con `typeid(Derived)` lugar de `typeid(Derived{})` :

```
assert(typeid(*b) == typeid(Derived{})); // OK
```

Si se aplica `typeid` a cualquier expresión que *no sea* del tipo de clase polimórfica, el operando no se evalúa, y la información de tipo devuelta es para el tipo estático.

```
struct Base {
    // note: no virtual destructor
};
struct Derived : Base {};
Derived d;
Base& b = d;
assert(typeid(b) == typeid(Base)); // not Derived
assert(typeid(std::declval<Base>()) == typeid(Base)); // OK because unevaluated
```

Cuándo usar el que está en c ++

Utilice **dynamic_cast** para convertir punteros / referencias dentro de una jerarquía de herencia.

Utilice **static_cast** para conversiones de tipo ordinario.

Utilice **reinterpret_cast** para la **reinterpretación** de bajo nivel de patrones de bits. Utilizar con extrema precaución.

Use **const_cast** para desechar const / volatile. Evita esto a menos que estés atascado usando una API const-incorrect.

Lea RTTI: Información de tipo de tiempo de ejecución en línea:
https://riptutorial.com/es/cplusplus/topic/3129/rtti--informacion-de-tipo-de-tiempo-de-ejecucion

Capítulo 115: Semáforo

Introducción

Los semáforos no están disponibles en C ++ a partir de ahora, pero se pueden implementar fácilmente con un mutex y una variable de condición.

Este ejemplo fue tomado de:

C ++ 0x no tiene semáforos? ¿Cómo sincronizar hilos?

Examples

Semáforo C ++ 11

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
    Semaphore (int count_ = 0)
    : count(count_)
    {
    }

    inline void notify( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        cout << "thread " << tid <<  " notify" << endl;
        //notify the waiting thread
        cv.notify_one();
    }
    inline void wait( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        while(count == 0) {
            cout << "thread " << tid << " wait" << endl;
            //wait on the mutex until notify is called
            cv.wait(lock);
            cout << "thread " << tid << " run" << endl;
        }
        count--;
    }
private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};
```

Clase de semáforo en acción.

La siguiente función agrega cuatro hilos. Tres hilos compiten por el semáforo, que se establece en una cuenta de uno. Un subproceso más lento llama a `notify_one()` , lo que permite que uno de

los subprocesos en espera continúe.

El resultado es que `s1` comienza a girar de inmediato, lo que hace que el `count` uso del Semáforo permanezca por debajo de 1. Los otros subprocesos esperan, a su vez, la variable de condición hasta que se llama a notificar ().

```
int main()
{
    Semaphore sem(1);

    thread s1([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.wait( 1 );
        }
    });
    thread s2([&]() {
        while(true){
            sem.wait( 2 );
        }
    });
    thread s3([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::milliseconds(600));
            sem.wait( 3 );
        }
    });
    thread s4([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.notify( 4 );
        }
    });


    s1.join();
    s2.join();
    s3.join();
    s4.join();

    ...
    }
```

Lea Semáforo en línea: https://riptutorial.com/es/cplusplus/topic/9785/semaforo

Capítulo 116: Separadores de dígitos

Examples

Separador de dígitos

Los literales numéricos de más de unos pocos dígitos son difíciles de leer.

- Pronuncia 7237498123.
- Compare 237498123 con 237499123 para la igualdad.
- Decida si 237499123 o 20249472 es más grande.

`C++14` define la comilla simple `'` como separador de dígitos, en números y literales definidos por el usuario. Esto puede hacer que sea más fácil para los lectores humanos analizar grandes números.

C ++ 14

```
long long decn = 1'000'000'000ll;
long long hexn = 0xFFFF'FFFFll;
long long octn = 00'23'00ll;
long long binn = 0b1010'001111;
```

Las comillas simples se ignoran al determinar su valor.

Ejemplo:

- Los literales `1048576` , `1'048'576` , `0X100000` , `0x10'0000` y `0'004'000'000` tienen el mismo valor.
- Los literales `1.602'176'565e-19` y `1.602176565e-19` tienen el mismo valor.

La posición de las comillas simples es irrelevante. Todos los siguientes son equivalentes:

C ++ 14

```
long long a1 = 12345678911;
long long a2 = 123'456'78911;
long long a3 = 12'34'56'78'911;
long long a4 = 12345'67891ll;
```

También está permitido en literales `user-defined` :

C ++ 14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Lea Separadores de dígitos en línea: https://riptutorial.com/es/cplusplus/topic/10595/separadores-de-digitos

Capítulo 117: SFINAE (el fallo de sustitución no es un error)

Examples

enable_if

`std::enable_if` es una utilidad conveniente para usar condiciones booleanas para activar SFINAE. Se define como:

```
template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};
```

Es decir, `enable_if<true, R>::type` es un alias para `R` , mientras que `enable_if<false, T>::type` está mal formado porque la especialización de `enable_if` no tiene un `type` miembro de tipo.

`std::enable_if` se puede usar para restringir plantillas:

```
int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }
```

Aquí, una llamada a `negate(1)` fallaría debido a la ambigüedad. Pero la segunda sobrecarga no está diseñada para usarse con tipos integrales, por lo que podemos agregar:

```
int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }
```

Ahora, la instanciación de `negate<int>` resultaría en una falla de sustitución ya que el valor `!std::is_arithmetic<int>::value` es `false` . Debido a SFINAE, esto no es un error grave, este candidato simplemente se elimina del conjunto de sobrecarga. Como resultado, `negate(1)` solo tiene un único candidato viable, que luego se llama.

Cuando usarlo

Vale la pena tener en cuenta que `std::enable_if` es un ayudante *además* de SFINAE, pero no es lo que hace que SFINAE funcione en primer lugar. Consideremos estas dos alternativas para implementar una funcionalidad similar a `std::size` , es decir, un `size(arg)` conjunto de sobrecarga

`size(arg)` que produce el tamaño de un contenedor o matriz:

```
// for containers
template<typename Cont>
auto size1(Cont const& cont) -> decltype( cont.size() );

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// implementation omitted
template<typename Cont>
struct is_sizeable;

// for containers
template<typename Cont, std::enable_if_t<std::is_sizeable<Cont>::value, int> = 0>
auto size2(Cont const& cont);

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size2(Elt const(&arr)[Size]);
```

Suponiendo que `is_sizeable` esté escrito correctamente, estas dos declaraciones deberían ser exactamente equivalentes con respecto a SFINAE. ¿Cuál es la más fácil de escribir y la más fácil de revisar y entender de un vistazo?

Ahora, consideremos cómo podríamos implementar ayudantes aritméticos que eviten el desbordamiento de enteros con signo en favor de un comportamiento envolvente o modular. Lo que quiere decir que, por ejemplo, `incr(i, 3)` sería lo mismo que `i += 3` excepto por el hecho de que el resultado siempre se definirá incluso si `i` es un `int` con valor `INT_MAX` . Estas son dos alternativas posibles:

```
// handle signed types
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(-1) < static_cast<Int>(0)]>;

// handle unsigned types by just doing target += amount
// since unsigned arithmetic already behaves as intended
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(0) < static_cast<Int>(-1)]>;

template<typename Int, std::enable_if_t<std::is_signed<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

template<typename Int, std::enable_if_t<std::is_unsigned<Int>::value, int> = 0>
void incr2(Int& target, Int amount);
```

Una vez más, ¿cuál es el más fácil de escribir y cuál es el más fácil de revisar y entender de un vistazo?

Una fortaleza de `std::enable_if` es cómo juega con la refactorización y el diseño de API. Si `is_sizeable<Cont>::value` pretende reflejar si `cont.size()` es válido, entonces usar la expresión como aparece para `size1` puede ser más conciso, aunque eso dependerá de si `is_sizeable` se

usaría en varios lugares o no. . Contraste que con `std::is_signed` que refleja su intención mucho más claramente que cuando su implementación se filtra en la declaración de `incr1` .

void_t

C ++ 11

`void_t` es una meta-función que mapea cualquier tipo (número de) al tipo `void` . El propósito principal de `void_t` es facilitar la escritura de rasgos de tipo.

`std::void_t` será parte de C ++ 17, pero hasta entonces, es muy sencillo de implementar:

```
template <class...> using void_t = void;
```

Algunos compiladores requieren una implementación ligeramente diferente:

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

La aplicación principal de `void_t` es escribir rasgos de tipo que comprueban la validez de una declaración. Por ejemplo, verifiquemos si un tipo tiene una función miembro `foo()` que no tome argumentos:

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T&>().foo())>> : std::true_type {};
```

¿Como funciona esto? Cuando intento crear `has_foo<T>::value` instancia de `has_foo<T>::value` , el compilador intentará buscar la mejor especialización para `has_foo<T, void>` . Tenemos dos opciones: la principal y la secundaria, que implica tener que instanciar esa expresión subyacente:

- Si `T` *tiene* una función miembro `foo()` , entonces cualquier tipo que devuelve se convierte a `void` , y la especialización se prefiere al primario basado en las reglas de ordenación parcial de la plantilla. Entonces `has_foo<T>::value` será `true`
- Si `T` *no* tiene una función miembro de este tipo (o requiere más de un argumento), entonces la sustitución falla para la especialización y solo tenemos la plantilla principal para el respaldo. Por lo tanto, `has_foo<T>::value` es `false` .

Un caso más simple:

```
template<class T, class=void>
struct can_reference : std::false_type {};

template<class T>
struct can_reference<T, std::void_t<T&>> : std::true_type {};
```

esto no usa `std::declval` o `decltype` .

Puedes notar un patrón común de un argumento vacío. Podemos factorizar esto:

```
struct details {
  template<template<class...>class Z, class=void, class...Ts>
  struct can_apply:
    std::false_type
  {};
  template<template<class...>class Z, class...Ts>
  struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...>:
    std::true_type
  {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

que oculta el uso de `std::void_t` y hace que `can_apply` actúe como un indicador si el tipo suministrado como primer argumento de plantilla está bien formado después de sustituir los otros tipos en él. Los ejemplos anteriores pueden ahora reescribirse usando `can_apply` como:

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>;    // Is T& well formed for T?
```

y:

```
template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply< dot_foo_r, T >;    // Is T.foo() well formed for T?
```

Lo que parece más sencillo que las versiones originales.

Hay post-C ++ 17 propuestas para `std` rasgos similares a `can_apply` .

La utilidad de `void_t` fue descubierta por Walter Brown. Dio una maravillosa presentación al respecto en CppCon 2016.

arrastrando decltype en plantillas de funciones

C ++ 11

Una de las funciones de restricción es usar el tipo de `decltype` final para especificar el tipo de retorno:

```
namespace details {
   using std::to_string;
```

```
   // this one is constrained on being able to call to_string(T)
   template <class T>
   auto convert_to_string(T const& val, int )
       -> decltype(to_string(val))
   {
       return to_string(val);
   }

   // this one is unconstrained, but less preferred due to the ellipsis argument
   template <class T>
   std::string convert_to_string(T const& val, ... )
   {
       std::ostringstream oss;
       oss << val;
       return oss.str();
   }
}

template <class T>
std::string convert_to_string(T const& val)
{
    return details::convert_to_string(val, 0);
}
```

Si llamo a `convert_to_string()` con un argumento con el que puedo invocar a `to_string()` , entonces tengo dos funciones viables para `details::convert_to_string()` . Se prefiere el primero ya que la conversión de `0` a `int` es una mejor secuencia de conversión implícita que la conversión de `0` a `...`

Si llamo a `convert_to_string()` con un argumento desde el cual no puedo invocar a `to_string()` , entonces la primera instanciación de la plantilla de función conduce a una falla de sustitución (no hay `decltype(to_string(val))`). Como resultado, ese candidato se elimina del conjunto de sobrecarga. La segunda plantilla de función no está restringida, por lo que está seleccionada y en su lugar, pasamos por el `operator<<(std::ostream&, T)` . Si ese no está definido, entonces tenemos un error de compilación con una pila de plantillas en la línea `oss << val` .

Que es el SFINAE

SFINAE significa **S** ubstitution **F** ailure **I** s **N** ot **A** n **E** rror. El código mal formado que resulta de la sustitución de tipos (o valores) para instanciar una plantilla de función o una plantilla de clase **no** es un error de compilación, solo se trata como un error de deducción.

Los fallos de deducción en las plantillas de función de creación de instancias o las especializaciones de plantillas de clase eliminan a ese candidato del conjunto de consideración, como si ese candidato fallido no existiera para empezar.

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }

int vals[10];
begin(vals); // OK. The first function template substitution fails because
```

```
            // vals.begin() is ill-formed. This is not an error! That function
            // is just removed from consideration as a viable overload candidate,
            // leaving us with the array overload.
```

Solo las fallas de sustitución en el **contexto inmediato** se consideran fallas de deducción, todas las demás se consideran errores graves.

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // ok

std::string msg = "Hello";
add_one(msg); // error. msg += 1 is ill-formed for std::string, but this
              // failure is NOT in the immediate context of substituting T
```

enable_if_all / enable_if_any

C ++ 11

Ejemplo motivacional

Cuando tiene un paquete de plantillas variadas en la lista de parámetros de la plantilla, como en el siguiente fragmento de código:

```
template<typename ...Args> void func(Args &&...args) { //... };
```

La biblioteca estándar (antes de C ++ 17) no ofrece una forma directa de escribir **enable_if** para imponer restricciones SFINAE en ***todos los parámetros*** en `Args` o ***cualquiera de los parámetros*** en `Args` . C ++ 17 ofrece `std::conjunction` y `std::disjunction` que resuelven este problema. Por ejemplo:

```
/// C++17: SFINAE constraints on all of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };

/// C++17: SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };
```

Si no tiene C ++ 17 disponible, existen varias soluciones para lograrlo. Una de ellas es usar una clase de caso base y **especializaciones parciales** , como se demuestra en las respuestas de esta pregunta .

Alternativamente, también se puede implementar a mano el comportamiento de `std::conjunction` y `std::disjunction` de una manera bastante directa. En el siguiente ejemplo, demostraré las implementaciones y las combinaré con `std::enable_if` para producir dos alias: `enable_if_all` y

`enable_if_any` , que hacen exactamente lo que se supone que deben hacer semánticamente. Esto puede proporcionar una solución más escalable.

Implementación de `enable_if_all` y `enable_if_any`

Primero `seq_and` `std::conjunction` y `std::disjunction` usando `seq_and` y `seq_or` respectivamente:

```
/// Helper for prior to C++14.
template<bool B, class T, class F >
using conditional_t = typename std::conditional<B,T,F>::type;

/// Emulate C++17 std::conjunction.
template<bool...> struct seq_or: std::false_type {};
template<bool...> struct seq_and: std::true_type {};

template<bool B1, bool... Bs>
struct seq_or<B1,Bs...>:
  conditional_t<B1,std::true_type,seq_or<Bs...>> {};

template<bool B1, bool... Bs>
struct seq_and<B1,Bs...>:
  conditional_t<B1,seq_and<Bs...>,std::false_type> {};
```

Entonces la implementación es bastante sencilla:

```
template<bool... Bs>
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;

template<bool... Bs>
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

Eventualmente algunos ayudantes:

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

Uso

El uso también es sencillo:

```
    /// SFINAE constraints on all of the parameters in Args.
    template<typename ...Args,
             enable_if_all_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };

    /// SFINAE constraints on any of the parameters in Args.
    template<typename ...Args,
             enable_if_any_t<custom_conditions_v<Args>...>* = nullptr>
```

```
    void func(Args &&...args) { //... };
```

is_detected

Para generalizar la creación de type_trait: en base a SFINAE hay rasgos experimentales `detected_or` o `detected_t` , `is_detected` .

Con los parámetros de la plantilla `typename Default` , `template <typename...> Op` y `typename ... Args` :

- `is_detected` : alias de `std::true_type` o `std::false_type` dependiendo de la validez de `Op<Args...>`
- `detected_t` : alias de `Op<Args...>` o `nonesuch` dependiendo de validez de `Op<Args...>` .
- `detected_or` : alias de una estructura con `value_t` que `is_detected` , y `type` que es `Op<Args...>` o `Default` dependiendo de la validez de `Op<Args...>`

que se puede implementar utilizando `std::void_t` para SFINAE de la siguiente manera:

C ++ 17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
        using value_t = std::true_type;
        using type = Op<Args...>;
    };

} // namespace detail

// special type to indicate detection failure
struct nonesuch {
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;
```

Los rasgos para detectar la presencia del método se pueden implementar simplemente:

```
typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));

struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
              "Unexpected");

static_assert(std::is_same<void, // Default
                           detected_or<void, foo_type, C1, char>>::value,
              "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
              "Unexpected");
```

Resolución de sobrecarga con un gran número de opciones.

Si necesita seleccionar entre varias opciones, habilitar solo una a través de `enable_if<>` puede ser bastante engorroso, ya que varias condiciones también deben ser negadas.

La ordenación entre sobrecargas se puede seleccionar en su lugar utilizando la herencia, es decir, el envío de etiquetas.

En lugar de probar lo que necesita estar bien formado, y también probar la negación de todas las demás condiciones de la versión, en lugar de eso, probamos solo lo que necesitamos, preferiblemente en un tipo de `decltype` en un retorno final.
Esto podría dejar varias opciones bien formadas, diferenciamos entre aquellos que usan 'etiquetas', similares a las etiquetas `random_access_tag` -rasgo (`random_access_tag` et al). Esto funciona porque una coincidencia directa es mejor que una clase base, que es mejor que una clase base de una clase base, etc.

```
#include <algorithm>
#include <iterator>

namespace detail
{
    // this gives us infinite types, that inherit from each other
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};

    // the overload we want to be preferred have a higher N in pick<N>
    // this is the first helper template function
```

```
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // if the container have a member stable_sort, use that
        t.stable_sort();
    }

    // this helper will be second best match
    template<typename T>
    auto stable_sort(T& t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // if the container have a member sort, but no member stable_sort
        // it's customary that the sort member is stable
        t.sort();
    }

    // this helper will be picked last
    template<typename T>
    auto stable_sort(T& t, pick<0>)
        -> decltype( std::stable_sort(std::begin(t), std::end(t)), void() )
    {
        // the container have neither a member sort, nor member stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }

}

// this is the function the user calls. it will dispatch the call
// to the correct implementation with the help of 'tags'.
template<typename T>
void stable_sort(T& t)
{
    // use an N that is higher that any used above.
    // this will pick the highest overload that is well formed.
    detail::stable_sort(t, detail::pick<10>{});
}
```

Existen otros métodos que se usan comúnmente para diferenciar las sobrecargas, como que la coincidencia exacta sea mejor que la conversión, que sea mejor que la elipsis.

Sin embargo, el envío de etiquetas puede extenderse a cualquier número de opciones, y es un poco más claro en la intención.

Lea SFINAE (el fallo de sustitución no es un error) en línea:
https://riptutorial.com/es/cplusplus/topic/1169/sfinae--el-fallo-de-sustitucion-no-es-un-error-

Capítulo 118: Sobrecarga de funciones

Introducción

Véase también el tema separado sobre resolución de sobrecarga

Observaciones

Se pueden producir ambigüedades cuando un tipo puede convertirse implícitamente en más de un tipo y no hay una función coincidente para ese tipo específico.

Por ejemplo:

```
void foo(double, double);
void foo(long, long);

//Call foo with 2 ints
foo(1, 2); //Function call is ambiguous - int can be converted into a double/long at the same
time
```

Examples

¿Qué es la sobrecarga de funciones?

La sobrecarga de funciones es tener múltiples funciones declaradas en el mismo ámbito con el mismo nombre exacto en el mismo lugar (conocido como *alcance*) que difiere solo en su *firma* , es decir, los argumentos que aceptan.

Supongamos que está escribiendo una serie de funciones para las capacidades de impresión generalizadas, comenzando con `std::string` :

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

Esto funciona bien, pero suponga que quiere una función que también acepte un `int` y que también lo imprima. Podrías escribir:

```
void print_int(int num)
{
    std::cout << "This is an int:  " << num << std::endl;
}
```

Pero debido a que las dos funciones aceptan diferentes parámetros, simplemente puede escribir:

```
void print(int num)
```

```
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Ahora tiene 2 funciones, ambas con nombre de `print` , pero con firmas diferentes. Uno acepta `std::string` , el otro es `int` . Ahora puedes llamarlos sin preocuparte por nombres diferentes:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

En lugar de:

```
print("Hello world!");
print_int(1337);
```

Cuando tiene funciones sobrecargadas, el compilador infiere a cuál de las funciones llamar desde los parámetros que le proporciona. Se debe tener cuidado al escribir sobrecargas de funciones. Por ejemplo, con conversiones de tipo implícitas:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Ahora no está claro de inmediato a qué sobrecarga de `print` se llama cuando escribe:

```
print(5);
```

Y es posible que necesites darle algunas pistas a tu compilador, como:

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

También se debe tener cuidado al escribir sobrecargas que aceptan parámetros opcionales:

```
// WRONG CODE
void print(int num1, int num2 = 0)    //num2 defaults to 0 if not included
{
    std::cout << "These are ints: << num1 << " and " << num2 << std::endl;
}
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Debido a que no hay forma de que el compilador sepa si una llamada como `print(17)` está

destinada a la primera o la segunda función debido al segundo parámetro opcional, esto no se compilará.

Tipo de retorno en la sobrecarga de funciones

Tenga en cuenta que no puede sobrecargar una función en función de su tipo de retorno. Por ejemplo:

```
// WRONG CODE
std::string getValue()
{
  return "hello";
}

int getValue()
{
  return 0;
}

int x = getValue();
```

Esto provocará un error de compilación, ya que el compilador no podrá determinar a qué versión de `getValue` llamar, aunque el tipo de retorno esté asignado a un `int` .

Función de miembro cv-qualifier Sobrecarga

Las funciones dentro de una clase pueden sobrecargarse cuando se accede a ellas mediante una referencia calificada por el CV a esa clase; esto es más comúnmente usado para sobrecargar para `const` , pero también puede ser usado para sobrecargar para `volatile` y `const volatile` . Esto se debe a que todas las funciones miembro no estáticas toman `this` como un parámetro oculto, al que se aplican los calificadores cv. Esto se usa más comúnmente para sobrecargar para `const` , pero también se puede usar para `volatile` y `const volatile` .

Esto es necesario porque solo se puede llamar a una función miembro si está al menos tan calificada como CV como la instancia a la que se llama. Mientras que una instancia no `const` puede llamar a miembros `const` y no `const` , una instancia `const` solo puede llamar miembros `const` . Esto permite que una función tenga un comportamiento diferente dependiendo de los calificadores cv de la instancia que llama, y le permite al programador no permitir funciones para un calificador (es) calificador (es) no deseado (s) al no proporcionar una versión con ese calificador (es).

Una clase con algunos básicos `print` método podría ser `const` sobrecargado de este modo:

```
#include <iostream>

class Integer
{
    public:
        Integer(int i_): i{i_}{}

        void print()
        {
```

```
            std::cout << "int: " << i << std::endl;
        }

        void print() const
        {
            std::cout << "const int: " << i << std::endl;
        }

    protected:
        int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}
```

Este es un principio clave de la corrección de `const` : Al marcar las funciones miembro como `const` , se les permite llamar en instancias `const` , lo que a su vez permite que las funciones tomen instancias como punteros / referencias `const` si no necesitan modificarlas. Esto permite que el código especifique si modifica el estado tomando parámetros no modificados como `const` y parámetros modificados sin calificadores cv, lo que hace que el código sea más seguro y más legible.

```
class ConstCorrect
{
  public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." <<
std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Error.  Can only be called from non-const instance.
}

void const_incorrect_func(ConstCorrect& cc)
{
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Good.  Can only be called from non-const instance.
}
```

Un uso común de esto es declarar los accesores como `const` , y los mutadores como no `const` .

Ningún miembro de clase puede ser modificado dentro de una función miembro `const` . Si hay algún miembro que realmente necesita modificar, como bloquear un `std::mutex` , puede declararlo como `mutable` :

```
class Integer
{
    public:
        Integer(int i_): i{i_}{}

        int get() const
        {
            std::lock_guard<std::mutex> lock{mut};
            return i;
        }

        void set(int i_)
        {
            std::lock_guard<std::mutex> lock{mut};
            i = i_;
        }

    protected:
        int i;
        mutable std::mutex mut;
};
```

Lea Sobrecarga de funciones en línea: https://riptutorial.com/es/cplusplus/topic/510/sobrecarga-de-funciones

Capítulo 119: Sobrecarga del operador

Introducción

En C ++, es posible definir operadores como `+` y `->` para tipos definidos por el usuario. Por ejemplo, el encabezado `<string>` define un operador `+` para concatenar cadenas. Esto se hace definiendo una *función de operador* utilizando la palabra clave del `operator` .

Observaciones

Los operadores para los tipos incorporados no se pueden cambiar, los operadores solo se pueden sobrecargar para los tipos definidos por el usuario. Es decir, al menos uno de los operandos debe ser de un tipo definido por el usuario.

Los siguientes operadores *no pueden* ser sobrecargados:

- El miembro de acceso o el operador de "punto" `.`
- El puntero al operador de acceso miembro `.*`
- El operador de resolución de alcance, `::`
- El operador condicional ternario, `?:`
- `dynamic_cast` , `static_cast` , `reinterpret_cast` , `const_cast` , `typeid` , `sizeof` , `alignof` y `noexcept`
- Las directivas de preprocesamiento, `#` y `##` , que se ejecutan antes de que esté disponible cualquier tipo de información.

Hay algunos operadores que **no** debería (99,98% de las veces) de sobrecarga:

- `&&` y `||` (Prefiero, en cambio, usar la conversión implícita a `bool`)
- `,`
- La dirección del operador (unario `&`)

¿Por qué? Porque sobrecargan a los operadores que otro programador nunca esperaría, lo que resulta en un comportamiento diferente al anticipado.

Por ejemplo, el usuario definió `&&` y `||` sobrecargas de estos operadores pierden su evaluación de cortocircuito y pierden sus propiedades de secuenciación especiales (C ++ 17) , la cuestión de la secuencia también se aplica a `,` sobrecargas de operadores.

Examples

Operadores aritméticos

Puedes sobrecargar todos los operadores aritméticos básicos:

- `+` y `+=`
- `-` y `-=`
- `*` y `*=`

- `/` y `/=`
- `&` y `&=`
- `|` y `|=`
- `^` y `^=`
- `>>` y `>>=`
- `<<` y `<<=`

La sobrecarga para todos los operadores es la misma. *Desplácese hacia abajo para la explicación*

Sobrecarga fuera de `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Sobrecarga dentro de `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Nota: el `operator+` debe devolver un valor no constante, ya que devolver una referencia no tendría sentido (devuelve un *nuevo* objeto) ni tampoco devolvería un valor `const` (por lo general, no debería devolver por `const`). El primer argumento se pasa por valor, ¿por qué? Porque

1. No puede modificar el objeto original (`Object foobar = foo + bar;` no debería modificar `foo` después de todo, no tendría sentido)
2. No puede hacer que sea `const` , porque tendrá que poder modificar el objeto (porque el `operator+` se implementa en términos de `operator+=` , que modifica el objeto)

Pasar por `const&` sería una opción, pero luego tendrá que hacer una copia temporal del objeto pasado. Al pasar por valor, el compilador lo hace por ti.

`operator+=` devuelve una referencia al mismo, porque es posible encadenarlos (aunque no use la misma variable, sería un comportamiento indefinido debido a los puntos de secuencia).

El primer argumento es una referencia (queremos modificarlo), pero no `const` , porque entonces no podrías modificarlo. El segundo argumento no debe modificarse, por lo que, por razones de rendimiento, pasa por `const&` (pasar por const es más rápido que por valor).

Operadores unarios

Puede sobrecargar los 2 operadores unarios:

- `++foo` y `foo++`
- `--foo` y `foo--`

La sobrecarga es la misma para ambos tipos (`++` y `--`). *Desplácese hacia abajo para la explicación*

Sobrecarga fuera de `class` / `struct` :

```
//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}
```

Sobrecarga dentro de `class` / `struct` :

```
//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}
```

Nota: El operador de prefijo devuelve una referencia a sí mismo, para que pueda continuar las

operaciones en él. El primer argumento es una referencia, ya que el operador de prefijo cambia el objeto, esa es también la razón por la que no es `const` (de lo contrario no podría modificarlo).

El operador de postfix devuelve por valor un temporal (el valor anterior), por lo que no puede ser una referencia, ya que sería una referencia a un temporal, que sería un valor de basura al final de la función, porque la variable temporal se apaga. de alcance). Tampoco puede ser `const` , porque deberías poder modificarlo directamente.

El primer argumento es una referencia no `const` al objeto "llamante", porque si fuera `const` , no sería capaz de modificarlo, y si no fuera una referencia, no cambiaría el valor original.

Es debido a la copia necesaria en las sobrecargas de operadores de postfix que es mejor hacer que sea un hábito usar el prefijo ++ en lugar de postfix ++ en bucles `for` . Desde la perspectiva del bucle `for` , por lo general son funcionalmente equivalentes, pero puede haber una ligera ventaja de rendimiento al usar el prefijo ++, especialmente con clases "grandes" con muchos miembros para copiar. Ejemplo de uso del prefijo ++ en un bucle for:

```
for (list<string>::const_iterator it = tokens.begin();
     it != tokens.end();
     ++it) { // Don't use it++
    ...
}
```

Operadores de comparación

Puede sobrecargar todos los operadores de comparación:

- `==` y `!=`
- `>` y `<`
- `>=` y `<=`

La forma recomendada de sobrecargar a todos esos operadores es implementando solo 2 operadores (`==` y `<`) y luego usándolos para definir el resto. *Desplácese hacia abajo para la explicación*

Sobrecarga fuera de `class` / `struct` :

```
//Only implement those 2
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Sobrecarga dentro de `class` / `struct` :

```
//Note that the functions are const, because if they are not const, you wouldn't be able
```

```
//to call them if the object is const

//Only implement those 2
bool operator==(const T& rhs) const { /* Compare */ }
bool operator<(const T& rhs) const { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& rhs) const { return !(*this == rhs); }
bool operator>(const T& rhs) const { return rhs < *this; }
bool operator<=(const T& rhs) const { return !(*this > rhs); }
bool operator>=(const T& rhs) const { return !(*this < rhs); }
```

Los operadores obviamente devuelven un `bool` , indicando `true` o `false` para la operación correspondiente.

Todos los operadores toman sus argumentos por `const&` , porque lo único que hacen los operadores es comparar, por lo que no deben modificar los objetos. Pasar por `&` (referencia) es más rápido que por valor, y para asegurarse de que los operadores no lo modifiquen, es una referencia `const` .

Tenga en cuenta que los operadores dentro de la `class` / `struct` se definen como `const` , la razón de esto es que sin las funciones `const` , no sería posible comparar objetos `const` , ya que el compilador no sabe que los operadores no modifican nada.

Operadores de conversión

Puede sobrecargar los operadores de tipo, de modo que su tipo pueda convertirse implícitamente en el tipo especificado.

El operador de conversión **debe** estar definido en una `class` / `struct` :

```
operator T() const { /* return something */ }
```

Nota: el operador es `const` para permitir la conversión de objetos `const` .

Ejemplo:

```
struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }
    // ^^^^^^^^
    // to disable implicit conversion
};

Text t;
t.text = "Hello world!";

//Ok
const char* copyoftext = t;
```

Operador de subíndice de matriz

Incluso puede sobrecargar el operador de subíndice de matriz `[]` .

Siempre debe (99,98% de las veces) implemente 2 versiones, una `const` y un no- `const` versión, ya que si el objeto es `const` , no debería ser capaz de modificar el objeto devuelto por `[]` .

Los argumentos se pasan por `const&` lugar de por valor porque pasar por referencia es más rápido que por valor, y `const` para que el operador no cambie el índice accidentalmente.

Los operadores devuelven por referencia, porque por diseño puede modificar la devolución del objeto `[]` , es decir:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
          //wouldn't be possible if not returned by reference
```

Sólo se puede sobrecargar el interior de una `class` / `struct` :

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Se pueden lograr múltiples operadores de subíndices, `[][]...` , a través de objetos proxy. El siguiente ejemplo de una clase simple de matriz de fila demuestra esto:

```
template<class T>
class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
```

```
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};
```

Operador de llamada de función

Puede sobrecargar la función llamada operador `()` :

La sobrecarga se debe hacer dentro de una `class` / `struct` :

```
//R -> Return type
//Types -> any different type
R operator()(Type name, Type2 name2, ...)
{
    //Do something
    //return something
}

//Use it like this (R is return type, a and b are variables)
R foo = object(a, b, ...);
```

Por ejemplo:

```
struct Sum
{
    int operator()(int a, int b)
    {
        return a + b;
    }
};

//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

Operador de asignación

El operador de asignación es uno de los operadores más importantes porque le permite cambiar el estado de una variable.

Si no sobrecarga el operador de asignación para su `class` / `struct` , el compilador lo genera automáticamente: el operador de asignación generado automáticamente realiza una "asignación de miembros", es decir, invocando operadores de asignación en todos los miembros, para que se copie un objeto al otro, un miembro a la vez. El operador de asignación debe estar sobrecargado cuando la asignación simple de miembros no es adecuada para su `class` / `struct` , por ejemplo, si necesita realizar una **copia profunda** de un objeto.

Sobrecargar el operador de asignación `=` es fácil, pero debe seguir algunos pasos simples.

1. **Prueba de autoasignación.** Esta verificación es importante por dos razones:
   - una autoasignación es una copia innecesaria, por lo que no tiene sentido realizarla;
   - El siguiente paso no funcionará en el caso de una autoasignación.
2. **Limpia los datos antiguos.** Los datos antiguos deben ser reemplazados por otros nuevos. Ahora, puede comprender el segundo motivo del paso anterior: si se destruyó el contenido del objeto, una autoasignación no podrá realizar la copia.
3. **Copia todos los miembros.** Si sobrecarga el operador de asignación para su `class` o su `struct` , el compilador no lo genera automáticamente, por lo que deberá hacerse cargo de copiar todos los miembros del otro objeto.
4. **Devuelve** `*this` . El operador regresa solo por referencia, porque permite el encadenamiento (es decir, `int b = (a = 6) + 4; //b == 10`).

```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Nota: `other` se pasa por `const&` , porque el objeto que se asigna no debe cambiarse, y pasar por referencia es más rápido que por valor, y para asegurarse de que `operator=` no lo modifique accidentalmente, es `const` .

El operador de asignación **solo se** puede sobrecargar en la `class` / `struct` , porque el valor izquierdo de `=` es **siempre** la `class` / `struct` sí. Definirlo como una función gratuita no tiene esta garantía, y por eso no está permitido.

Cuando lo declara en la `class` / `struct` , el valor de la izquierda es implícitamente la `class` / `struct` sí, por lo que no hay problema con eso.

Operador NO bit a bit

Sobrecargar el bit a bit (`~`) es bastante simple. *Desplácese hacia abajo para la explicación*

Sobrecarga fuera de `class` / `struct` :

```
T operator~(T lhs)
{
    //Do operation
    return lhs;
}
```

Sobrecarga dentro de `class` / `struct` :

```
T operator~()
{
    T t(*this);
    //Do operation
    return t;
}
```

Nota: `operator~` devuelve por valor, porque tiene que devolver un nuevo valor (el valor modificado), y no una referencia al valor (sería una referencia al objeto temporal, que tendría un valor de basura en él tan pronto como el operador está hecho). `const` porque el código de llamada debe poder modificarlo posteriormente (es decir, `int a = ~a + 1;` debería ser posible).

Dentro de la `class` / `struct` usted tiene que hacer un objeto temporal, porque no se puede modificar `this` , ya que modificaría el objeto original, que no debería ser el caso.

Operadores de cambio de bit para E / S

Los operadores `<<` y `>>` se utilizan comúnmente como operadores de "escritura" y "lectura":

- `std::ostream` overloads `<<` para escribir variables en el flujo subyacente (ejemplo: `std::cout`)
- `std::istream` overloads `>>` para leer desde el flujo subyacente a una variable (ejemplo: `std::cin`)

La forma en que lo hacen es similar si desea sobrecargarlos "normalmente" fuera de la `class` / `struct` , excepto que la especificación de los argumentos no es del mismo tipo:

- Tipo de retorno es el flujo del que desea sobrecargar (por ejemplo, `std::ostream`) pasado por referencia, para permitir el encadenamiento (Encadenamiento: `std::cout << a << b;`). Ejemplo: `std::ostream&`
- `lhs` sería el mismo que el tipo de retorno
- `rhs` es el tipo desde el que desea permitir la sobrecarga (es decir, `T`), pasado por `const&` lugar de valor por razones de rendimiento (`rhs` no debería cambiarse de todos modos). Ejemplo: `const Vector&` .

Ejemplo:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
    lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
```

```
    return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

Números complejos revisados

El siguiente código implementa un tipo de número complejo muy simple para el cual el campo subyacente se promueve automáticamente, siguiendo las reglas de promoción del tipo de idioma, bajo la aplicación de los cuatro operadores básicos (+, -, * y /) con un miembro de un campo diferente (ya sea otro `complex<T>` o algún tipo escalar).

Se pretende que sea un ejemplo holístico que cubra la sobrecarga de operadores junto con el uso básico de plantillas.

```
#include <type_traits>

namespace not_std{

using std::decay_t;

//---------------------------------------------------------------------
// complex< value_t >
//---------------------------------------------------------------------

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
        this->x -= x;
        return *this;
    }
    complex &operator -= (const complex &other)
    {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }
```

```
    complex &operator *= (const value_t &s)
    {
        this->x *= s;
        this->y *= s;
        return *this;
    }
    complex &operator *= (const complex &other)
    {
        (*this) = (*this) * other;
        return *this;
    }

    complex &operator /= (const value_t &s)
    {
        this->x /= s;
        this->y /= s;
        return *this;
    }
    complex &operator /= (const complex &other)
    {
        (*this) = (*this) / other;
        return *this;
    }

    complex(const value_t &x, const value_t &y)
    : x{x}
    , y{y}
    {}

    template<typename other_value_t>
    explicit complex(const complex<other_value_t> &other)
    : x{static_cast<const value_t &>(other.x)}
    , y{static_cast<const value_t &>(other.y)}
    {}

    complex &operator = (const complex &) = default;
    complex &operator = (complex &&) = default;
    complex(const complex &) = default;
    complex(complex &&) = default;
    complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//---------------------------------------------------------------
// operator - (negation)
//---------------------------------------------------------------

template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//---------------------------------------------------------------
// operator +
//---------------------------------------------------------------

template<typename left_t,typename right_t>
```

```
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }

template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//-------------------------------------------------------------------
// operator -
//-------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//-------------------------------------------------------------------
// operator *
//-------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
        };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }

//-------------------------------------------------------------------
// operator /
//-------------------------------------------------------------------
```

```
template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>
{
    const auto r = absqr(b);
    return {
        ( a.x*b.x + a.y*b.y) / r,
        (-a.x*b.y + a.y*b.x) / r
        };
}

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
         b.x * s,
        -b.y * s
        };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std


int main(int argc, char **argv)
{
    using namespace not_std;

    complex<float> fz{4.0f, 1.0f};

    // makes a complex<double>
    auto dz = fz * 1.0;

    // still a complex<double>
    auto idz = 1.0f/dz;

    // also a complex<double>
    auto one = dz * idz;

    // a complex<double> again
    auto one_again = fz * idz;

    // Operator tests, just to make sure everything compiles.

    complex<float> a{1.0f, -2.0f};
    complex<double> b{3.0, -4.0};

    // All of these are complex<double>
    auto c0 = a + b;
    auto c1 = a - b;
    auto c2 = a * b;
    auto c3 = a / b;

    // All of these are complex<float>
    auto d0 = a + 1;
```

```
    auto d1 = 1 + a;
    auto d2 = a - 1;
    auto d3 = 1 - a;
    auto d4 = a * 1;
    auto d5 = 1 * a;
    auto d6 = a / 1;
    auto d7 = 1 / a;

    // All of these are complex<double>
    auto e0 = b + 1;
    auto e1 = 1 + b;
    auto e2 = b - 1;
    auto e3 = 1 - b;
    auto e4 = b * 1;
    auto e5 = 1 * b;
    auto e6 = b / 1;
    auto e7 = 1 / b;

    return 0;
}
```

Operadores nombrados

Puede extender C ++ con operadores nombrados que están "cotizados" por los operadores estándar de C ++.

Primero comenzamos con una biblioteca de doce líneas:

```
namespace named_operator {
  template<class D>struct make_operator{constexpr make_operator(){}};

  template<class T, char, class O> struct half_apply { T&& lhs; };

  template<class Lhs, class Op>
  half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
    return {std::forward<Lhs>(lhs)};
  }

  template<class Lhs, class Op, class Rhs>
  auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
  -> decltype( named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) ) )
  {
    return named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) );
  }
}
```

Esto no hace nada todavía.

Primero, adjuntando vectores.

```
namespace my_ns {
  struct append_t : named_operator::make_operator<append_t> {};
  constexpr append_t append{};

  template<class T, class A0, class A1>
  std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const&
rhs ) {
```

```
      lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
      return std::move(lhs);
  }
}
using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};

auto c = a *append* b;
```

El núcleo de este caso es que definimos un `append` objeto de tipo `append_t:named_operator::make_operator<append_t>` .

Luego sobrecargamos named_invoke (lhs, append_t, rhs) para los tipos que queremos a la derecha e izquierda.

La biblioteca sobrecarga `lhs*append_t` , devolviendo un objeto `half_apply` temporal. También sobrecarga `half_apply*rhs` para llamar `named_invoke( lhs, append_t, rhs )` .

Simplemente tenemos que crear el token `append_t` adecuado y hacer un `named_invoke` ADL de la firma adecuada, y todo se conecta y funciona.

Para un ejemplo más complejo, suponga que quiere tener una multiplicación de elementos de un std :: array:

```
template<class=void, std::size_t...Is>
auto indexer( std::index_sequence<Is...> ) {
  return [](auto&& f) {
    return f( std::integral_constant<std::size_t, Is>{}... );
  };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
  struct e_times_t : named_operator::make_operator<e_times_t> {};
  constexpr e_times_t e_times{};

  template<class L, class R, std::size_t N,
    class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
  >
  std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N>
const& rhs ) {
    using result_type = std::array<Out, N>;
    auto index_over_N = indexer<N>();
    return index_over_N([&](auto...is)->result_type {
      return {{
        (lhs[is] * rhs[is])...
      }};
    });
  }
}
```

ejemplo vivo .

Este código de matriz de elementos se puede ampliar para trabajar en tuplas o pares o matrices de estilo C, o incluso contenedores de longitud variable si decide qué hacer si las longitudes no coinciden.

También puede `lhs *element_wise<'+'>* rhs` tipo de operador inteligente y obtener `lhs *element_wise<'+'>* rhs` .

Escribir operadores de productos `*dot*` y `*cross*` también son usos obvios.

El uso de `*` se puede extender para admitir otros delimitadores, como `+` . La precisión del delimitador determina la precisión del operador nombrado, lo que puede ser importante al traducir las ecuaciones físicas a C ++ con un uso mínimo de extra `()` s.

Con un ligero cambio en la biblioteca anterior, podemos admitir `->*then*` operadores y extender la `std::function` antes de que se actualice el estándar, o escribir de forma monádica `->*bind*` . También podría tener un operador con nombre de estado, donde pasamos cuidadosamente la `Op` hasta la función de invocación final, permitiendo:

```
named_operator<'*'> append = [](auto lhs, auto&& rhs) {
  using std::begin; using std::end;
  lhs.insert( end(lhs), begin(rhs), end(rhs) );
  return std::move(lhs);
};
```

generando un operador anexado de contenedores nombrado en C ++ 17.

Lea Sobrecarga del operador en línea: https://riptutorial.com/es/cplusplus/topic/562/sobrecarga-del-operador

Capítulo 120: static_assert

Sintaxis

- static_assert (*bool_constexpr* , *mensaje*)
- static_assert (*bool_constexpr*) / * Desde C ++ 17 * /

Parámetros

Parámetro	Detalles
bool_constexpr	Expresión para comprobar
mensaje	Mensaje para imprimir cuando *bool_constexpr* es *falso*

Observaciones

A diferencia de las aserciones en tiempo de ejecución , las aserciones estáticas se verifican en tiempo de compilación y también se aplican cuando se compilan construcciones optimizadas.

Examples

static_assert

Las afirmaciones significan que una condición debe ser verificada y si es falsa, es un error. Para `static_assert()` , esto se hace en tiempo de compilación.

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() only works for integral types" );
    return (t << 3) + (t << 1);
}
```

Un `static_assert()` tiene un primer parámetro obligatorio, la condición, que es un bex constexpr. *Puede* tener un segundo parámetro, el mensaje, que es una cadena literal. Desde C ++ 17, el segundo parámetro es opcional; antes de eso, es obligatorio.

C ++ 17

```
template<typename T>
T mul10(const T t)
{
    static_assert(std::is_integral<T>::value);
    return (t << 3) + (t << 1);
}
```

Se utiliza cuando:

- En general, se requiere una verificación en tiempo de compilación en algún tipo en constexpr value
- Una función de plantilla necesita verificar ciertas propiedades de un tipo que se le pasa
- Uno quiere escribir casos de prueba para:
  - plantilla de metafunciones
  - funciones constexpr
  - macro metaprogramacion
- Se requieren ciertas definiciones (por ejemplo, versión C ++)
- Código legado de portabilidad, afirmaciones sobre `sizeof(T)` (p. Ej., Int. De 32 bits)
- Se requieren ciertas funciones del compilador para que el programa funcione (empaquetamiento, optimización de clase base vacía, etc.)

Tenga en cuenta que `static_assert()` no participa en SFINAE : por lo tanto, cuando son posibles sobrecargas / especializaciones adicionales, no se debe usar en lugar de técnicas de metaprogramación de plantillas (como `std::enable_if<>`). Puede usarse en el código de la plantilla cuando ya se encuentra la sobrecarga / especialización esperada, pero se requieren verificaciones adicionales. En tales casos, podría proporcionar mensajes de error más concretos que confiar en SFINAE para esto.

Lea static_assert en línea: https://riptutorial.com/es/cplusplus/topic/3822/static-assert

Capítulo 121: std :: array

Parámetros

Parámetro	Definición
`class T`	Especifica el tipo de datos de los miembros de la matriz.
`std::size_t N`	Especifica el número de miembros en la matriz

Observaciones

El uso de un `std::array` requiere la inclusión del encabezado `<array>` usando `#include <array>` .

Examples

Inicializando un std :: array

Inicializando `std::array<T, N>` , donde `T` es un tipo escalar y `N` es el número de elementos de tipo `T`

Si `T` es un tipo escalar, `std::array` se puede inicializar de las siguientes maneras:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Inicializando `std::array<T, N>` , donde `T` es un tipo no escalar y `N` es el número de elementos de tipo `T`

Si `T` es un tipo no escalar, `std::array` se puede inicializar de las siguientes maneras:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
```

```
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Acceso a elementos

1. **`at(pos)`**

Devuelve una referencia al elemento en la posición `pos` con comprobación de límites. Si `pos` no está dentro del rango del contenedor, se lanza una excepción de tipo `std::out_of_range` .

La complejidad es constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) **`operator[pos]`**

Devuelve una referencia al elemento en la posición `pos` sin verificación de límites. Si `pos` no está dentro del rango del contenedor, puede ocurrir un error de *violación de segmentación en* tiempo de ejecución. Este método proporciona acceso a elementos equivalente a las matrices clásicas y, por lo tanto, más eficiente que `at(pos)` .

La complejidad es constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) `std::get<pos>`

Esta función que **no es miembro** devuelve una referencia al elemento en la posición **constante en tiempo de compilación** `pos` sin verificación de límites. Si `pos` no está dentro del rango del contenedor, puede ocurrir un error de *violación de segmentación en* tiempo de ejecución.

La complejidad es constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) `front()`

Devuelve una referencia al primer elemento en el contenedor. Llamar a `front()` en un contenedor vacío no está definido.

La complejidad es constante O (1).

Nota: Para un contenedor c, la expresión `c.front()` es equivalente a `*c.begin()` .

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) **`back()`**

Devuelve la referencia al último elemento en el contenedor. La `back()` llamada `back()` en un contenedor vacío no está definido.

La complejidad es constante O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) **`data()`**

Devuelve el puntero a la matriz subyacente que sirve como almacenamiento de elementos. El puntero es tal que el `range [data(); data() + size())` siempre es un rango válido, incluso si el contenedor está vacío (`data()` no se puede descifrar en ese caso).

La complejidad es constante O (1).

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string

    return 0;
}
```

Comprobando el tamaño de la matriz

Una de las principales ventajas de `std::array` en comparación con la matriz de estilo `C` es que podemos verificar el tamaño de la matriz utilizando la función miembro `size()`

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Iterando a través de la matriz

`std::array` es un contenedor STL, puede usar un rango basado en bucle similar a otros contenedores como `vector`

```
int main() {
     std::array<int, 3> arr = { 1, 2, 3 };
     for (auto i : arr)
         cout << i << '\n';
}
```

Cambiando todos los elementos de la matriz a la vez

La función miembro `fill()` se puede usar en `std::array` para cambiar los valores a la vez después de la inicialización

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

Lea std :: array en línea: https://riptutorial.com/es/cplusplus/topic/2712/std----array

Capítulo 122: std :: atómica

Examples

tipos atómicos

Cada creación de instancias y la especialización completa de la plantilla `std::atomic` define un tipo atómico. Si un hilo se escribe en un objeto atómico mientras otro hilo lo lee, el comportamiento está bien definido (consulte el modelo de memoria para obtener detalles sobre las carreras de datos)

Además, los accesos a objetos atómicos pueden establecer la sincronización entre subprocesos y ordenar accesos de memoria no atómica según lo especificado por `std::memory_order` .

std :: atomic se puede crear una instancia con cualquier `TriviallyCopyable type T. std::atomic` no se puede copiar ni mover.

La biblioteca estándar proporciona especializaciones de la plantilla std :: atomic para los siguientes tipos:

1. Se define una especialización completa para el tipo `bool` y su nombre typedef que se trata como un `std::atomic<T>` no especializado, excepto que tiene un diseño estándar, un constructor predeterminado trivial, un destructor trivial y soporta la sintaxis de inicialización agregada:

Nombre de typedef	Especialización completa
std::atomic_bool	std::atomic<bool>

2) Especializaciones completas y typedefs para tipos integrales, como sigue:

Nombre de typedef	Especialización completa
std::atomic_char	std::atomic<char>
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>

Nombre de typedef	Especialización completa
std::atomic_ulong	std::atomic<unsigned long>
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>
std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>

Nombre de typedef	Especialización completa
std::atomic_uintptr_t	std::atomic<std::uintptr_t>
std::atomic_size_t	std::atomic<std::size_t>
std::atomic_ptrdiff_t	std::atomic<std::ptrdiff_t>
std::atomic_intmax_t	std::atomic<std::intmax_t>
std::atomic_uintmax_t	std::atomic<std::uintmax_t>

Ejemplo simple de usar std :: atomic_int

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::memory_order_relaxed
#include <thread>         // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
  foo.store(x,std::memory_order_relaxed);     // set value atomically
}

void print_foo() {
  int x;
  do {
    x = foo.load(std::memory_order_relaxed);  // get value atomically
  } while (x==0);
  std::cout << "foo: " << x << '\n';
}

int main ()
{
  std::thread first (print_foo);
  std::thread second (set_foo,10);
  first.join();
  //second.join();
  return 0;
}
//output: foo: 10
```

Lea std :: atómica en línea: https://riptutorial.com/es/cplusplus/topic/7475/std----atomica

Capítulo 123: std :: cualquiera

Observaciones

La clase `std::any` proporciona un contenedor seguro de tipos en el que podemos poner valores únicos de cualquier tipo.

Examples

Uso básico

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << '\n';
}

try {
  std::any_cast<int>(an_object);
} catch(std::bad_any_cast&) {
    std::cout << "Wrong type\n";
}

std::any_cast<std::string&>(an_object) = "42";
std::cout << std::any_cast<std::string>(an_object) << '\n';
```

Salida

```
hello world
Wrong type
42
```

Lea std :: cualquiera en línea: https://riptutorial.com/es/cplusplus/topic/7894/std----cualquiera

Capítulo 124: std :: forward_list

Introducción

`std::forward_list` es un contenedor que admite la inserción y eliminación rápida de elementos desde cualquier parte del contenedor. El acceso aleatorio rápido no es compatible. Se implementa como una lista enlazada individualmente y, en esencia, no tiene ningún gasto general en comparación con su implementación en C. Comparado con `std::list` este contenedor proporciona un almacenamiento más eficiente en espacio cuando no se necesita la iteración bidireccional.

Observaciones

Agregar, eliminar y mover los elementos dentro de la lista, o en varias listas, no invalida los iteradores que actualmente hacen referencia a otros elementos en la lista. Sin embargo, un iterador o referencia que se refiere a un elemento se invalida cuando el elemento correspondiente se elimina (a través de erase_after) de la lista. std :: forward_list cumple con los requisitos de Container (excepto la función de miembro de tamaño y la complejidad de ese operador == es siempre lineal), AllocatorAwareContainer y SequenceContainer.

Examples

Ejemplo

```
#include <forward_list>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
```

```
    std::forward_list<std::string> words3(words1);
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Salida:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

Métodos

Nombre del método	Definición
`operator=`	asigna valores al contenedor
`assign`	asigna valores al contenedor
`get_allocator`	devuelve el asignador asociado
------	------
Acceso a elementos	
`front`	acceder al primer elemento
------	------
Iteradores	
`before_begi` n	devuelve un iterador al elemento antes de comenzar
`cbefore_begin`	devuelve un iterador constante al elemento antes de comenzar
`begin`	regresa un iterador al principio
`cbegin`	devuelve un iterador de const al principio
`end`	devuelve un iterador al final
`cend`	devuelve un iterador al final
Capacidad	
`empty`	comprueba si el contenedor está vacío
`max_size`	Devuelve el máximo número posible de elementos.

Nombre del método	Definición
Modificadores	
`clear`	borra los contenidos
`insert_after`	inserta elementos después de un elemento
`emplace_after`	construye elementos en el lugar después de un elemento
`erase_after`	borra un elemento después de un elemento
`push_front`	inserta un elemento al principio
`emplace_front`	construye un elemento en el lugar al principio
`pop_front`	elimina el primer elemento
`resize`	Cambia el número de elementos almacenados.
`swap`	intercambia los contenidos
Operaciones	
`merge`	fusiona dos listas ordenadas
`splice_after`	Mueve elementos de otro forward_list
`remove`	Remueve elementos que satisfacen criterios específicos.
`remove_if`	Remueve elementos que satisfacen criterios específicos.
`reverse`	invierte el orden de los elementos
`unique`	elimina elementos duplicados consecutivos
`sort`	ordena los elementos

Lea std :: forward_list en línea: https://riptutorial.com/es/cplusplus/topic/9703/std----forward-list

Capítulo 125: std :: function: Para envolver cualquier elemento que sea llamable

Examples

Uso simple

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

std :: función utilizada con std :: bind

Piense en una situación en la que necesitamos devolver una función con argumentos. `std::function` utilizada con `std::bind` proporciona una construcción de diseño muy potente como se muestra a continuación.

```
class A
{
public:
    std::function<void(int, const std::string&)> m_CbFunc = nullptr;
    void foo()
    {
        if (m_CbFunc)
        {
            m_CbFunc(100, "event fired");
        }
    }

};

class B
{
public:
    B()
    {
        auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
std::placeholders::_2);
        anObjA.m_CbFunc = aFunc;
    }
    void eventHandler(int i, const std::string& s)
    {
        std::cout << s << ": " << i << std::endl;
```

```
    }

    void DoSomethingOnA()
    {
        anObjA.foo();
    }

    A anObjA;
};

int main(int argc, char *argv[])
{
     B anObjB;
     anObjB.DoSomethingOnA();
}
```

std :: función con lambda y std :: bind

```
#include <iostream>
#include <functional>

using std::placeholders::_1; // to be used in std::bind example

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // std::function moo called
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* Function pointers */
    std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // can also be: stdf_foobar(2, foo)

    /* Lambda expressions */
    /* An unnamed closure from a lambda expression can be
     * stored in a std::function object:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                             [capture_value](int param) -> int { return 7 + capture_value *
param; })
              << std::endl;
    // result: 15 ==  value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind expressions */
    /* The result of a std::bind expression can be passed.
     * For example by binding parameters to a function pointer call:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
```

```
    std::cout << c << std::endl;
    // c == 49 == 2 + ( 9*5 + 2 )

    return 0;
}
```

`function` sobrecarga

`std::function` puede causar una sobrecarga significativa. Debido a que `std::function` tiene [valor semántico] [1], debe copiar o mover el llamable dado en sí mismo. Pero como puede tomar callables de un tipo arbitrario, con frecuencia tendrá que asignar memoria dinámicamente para hacer esto.

Algunas implementaciones de `function` se denominan "optimización de objetos pequeños", donde los tipos pequeños (como punteros de función, punteros de miembro o functores con muy poco estado) se almacenarán directamente en el objeto de `function` . Pero incluso esto solo funciona si el tipo es `noexcept` mover constructible. Además, el estándar de C ++ no requiere que todas las implementaciones proporcionen una.

Considera lo siguiente:

```
//Header file
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;

void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//Source file
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

Un parámetro de plantilla sería la solución preferida para `SortMyContainer` , pero supongamos que esto no es posible o deseable por cualquier motivo. `SortMyContainer` no necesita almacenar `pred` más allá de su propia llamada. Y, sin embargo, `pred` puede asignar memoria si el functor que se le asigna es de algún tamaño no trivial.

`function` asigna memoria porque necesita algo para copiar / mover; `function` se apropia de lo callable que se le da. Pero `SortMyContainer` no necesita *poseer* el invocable; es solo referenciarlo Así que usar la `function` aquí es una exageración; Puede ser eficiente, pero puede no serlo.

No hay un tipo de función de biblioteca estándar que simplemente haga referencia a un llamable. Por lo tanto, habrá que encontrar una solución alternativa, o puede elegir vivir con los gastos generales.

Además, la `function` no tiene medios efectivos para controlar de dónde provienen las asignaciones de memoria para el objeto. Sí, tiene constructores que toman un `allocator` , pero [muchas implementaciones no los implementan correctamente ... o incluso *en absoluto*] [2].

C ++ 17

Los constructores de `function` que toman un `allocator` ya no forman parte del tipo. Por lo tanto, no hay manera de gestionar la asignación.

Llamar a una `function` también es más lento que llamar directamente al contenido. Dado que cualquier instancia de `function` podría contener cualquier llamada, la llamada a través de una `function` debe ser indirecta. La sobrecarga de la `function` de llamada está en el orden de una llamada de función virtual.

Enlace std :: función a diferentes tipos de llamada

```
/*
 * This example show some ways of using std::function to call
 *  a) C-like function
 *  b) class-member function
 *  c) operator()
 *  d) lambda function
 *
 * Function call can be made:
 *  a) with right arguments
 *  b) argumens with different order, types and count
 */
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;



// simple function to be called
double foo_fn(int x, float y, double z)
{
  double res = x + y + z;
  std::cout << "foo_fn called with arguments: "
            << x << ", " << y << ", " << z
            << " result is : " << res
            << std::endl;
  return res;
}

// structure with member function to call
struct foo_struct
{
    // member function to call
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
    // this member function has different signature - but it can be used too
    // please not that argument order is changed too
```

```
    double foo_fn_4(int x, double z, float y, long xx)
    {
        double res = x + y + z + xx;
        std::cout << "foo_struct::foo_fn_4 called with arguments: "
                << x << ", " << z << ", " << y << ", " << xx
                << " result is : " << res
                << std::endl;
        return res;
    }
    // overloaded operator() makes whole object to be callable
    double operator()(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::operator() called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
};


int main(void)
{
  // typedefs
  using function_type = std::function<double(int, float, double)>;

  // foo_struct instance
  foo_struct fs;

  // here we will store all binded functions
  std::vector<function_type> bindings;

  // var #1 - you can use simple function
  function_type var1 = foo_fn;
  bindings.push_back(var1);

  // var #2 - you can use member function
  function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
  bindings.push_back(var2);

  // var #3 - you can use member function with different signature
  // foo_fn_4 has different count of arguments and types
  function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, 0l);
  bindings.push_back(var3);

  // var #4 - you can use object with overloaded operator()
  function_type var4 = fs;
  bindings.push_back(var4);

  // var #5 - you can use lambda function
  function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "lambda  called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    };
  bindings.push_back(var5);
```

```
  std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"
            << std::endl;

  for (auto f : bindings)
      f(1, 2, 3);

}
```

Vivir

Salida:

```
Test stored functions with arguments: x = 1, y = 2, z = 3
foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn_4 called with arguments: 1, 3, 2, 0 result is : 6
foo_struct::operator() called with arguments: 1, 2, 3 result is : 6
lambda  called with arguments: 1, 2, 3 result is : 6
```

Almacenando argumentos de funciones en std :: tuple

Algunos programas necesitan almacenar los argumentos para futuras llamadas de alguna función.

Este ejemplo muestra cómo llamar a cualquier función con argumentos almacenados en std :: tuple

```
#include <iostream>
#include <functional>
#include <tuple>
#include <iostream>

// simple function to be called
double foo_fn(int x, float y, double z)
{
   double res =  x + y + z;
   std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z
             << " res=" << res;
   return res;
}

// helpers for tuple unrolling
template<int ...> struct seq {};
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };

// invocation helper
template<typename FN, typename P, int ...S>
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)
{
   return fn(std::get<S>(params) ...);
}
// call function with arguments stored in std::tuple
template<typename Ret, typename ...Args>
Ret call_fn(const std::function<Ret(Args...)>& fn,
```

```
            const std::tuple<Args...>& params)
{
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());
}


int main(void)
{
  // arguments
  std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);
  // function to call
  std::function<double(int, float, double)> fn = foo_fn;

  // invoke a function with stored arguments
  call_fn(fn, t);
}
```

Vivir

Salida:

```
foo_fn called. x = 1 y = 5 z = 10 res=16
```

Lea std :: function: Para envolver cualquier elemento que sea llamable en línea: https://riptutorial.com/es/cplusplus/topic/2294/std----function--para-envolver-cualquier-elemento-que-sea-llamable

Capítulo 126: std :: integer_sequence

Introducción

La plantilla de clase `std::integer_sequence<Type, Values...>` representa una secuencia de valores de tipo `Type` donde `Type` es uno de los tipos de enteros incorporados. Estas secuencias se utilizan al implementar plantillas de clase o función que se benefician del acceso posicional. La biblioteca estándar también contiene tipos "de fábrica" que crean secuencias ascendentes de valores enteros solo a partir del número de elementos.

Examples

Gire un std :: tuple en parámetros de función

Se puede usar un `std::tuple<T...>` para pasar varios valores. Por ejemplo, podría usarse para almacenar una secuencia de parámetros en alguna forma de una cola. Al procesar tal tupla, sus elementos deben convertirse en argumentos de llamada de función:

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// ----------------------------------------------------------------------------
// Example functions to be called:
void f(int i, std::string const& s) {
    std::cout << "f(" << i << ", " << s << ")\n";
}
void f(int i, double d, std::string const& s) {
    std::cout << "f(" << i << ", " << d << ", " << s << ")\n";
}
void f(char c, int i, double d, std::string const& s) {
    std::cout << "f(" << c << ", " << i << ", " << d << ", " << s << ")\n";
}
void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")\n";
}

// ----------------------------------------------------------------------------
// The actual function expanding the tuple:
template <typename Tuple, std::size_t... I>
void process(Tuple const& tuple, std::index_sequence<I...>) {
    f(std::get<I>(tuple)...);
}

// The interface to call. Sadly, it needs to dispatch to another function
// to deduce the sequence of indices created from std::make_index_sequence<N>
template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
}
```

```
// ------------------------------------------------------------------------
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}
```

Siempre que una clase admita `std::get<I>(object)` y `std::tuple_size<T>::value` se puede expandir con la función `process()` . La función en sí es totalmente independiente del número de argumentos.

Crear un paquete de parámetros que consiste en enteros.

`std::integer_sequence` sí mismo consiste en mantener una secuencia de enteros que se pueden convertir en un paquete de parámetros. Su valor principal es la posibilidad de crear plantillas de clase "de fábrica" creando estas secuencias:

```
#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) {
    std::initializer_list<bool>{ bool(std::cout << I << ' ')... };
    std::cout << '\n';
}

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // explicitly specify sequences:
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // generate sequences:
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}
```

La plantilla de la función `print_sequence()` usa un `std::initializer_list<bool>` al expandir la secuencia de enteros para garantizar el orden de evaluación y no crear una variable [array] no utilizada.

Convertir una secuencia de índices en copias de un elemento.

La expansión del paquete de parámetros de índices en una expresión de coma con un valor crea una copia del valor para cada uno de los índices. Lamentablemente, `gcc` y el `clang` pensar en el índice no tiene ningún efecto y advertir de ello (`gcc` se puede silenciar el índice de fundición a `void`

):

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

template <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

template <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
    auto array = make_array<20>(std::string("value"));
    std::copy(array.begin(), array.end(),
              std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << "\n";
}
```

Lea std :: integer_sequence en línea: https://riptutorial.com/es/cplusplus/topic/8315/std----integer-sequence

Capítulo 127: std :: iomanip

Examples

std :: setw

```
int val = 10;
// val will be printed to the extreme left end of the  output console:
std::cout << val << std::endl;
 // val will be printed in an output field of length 10 starting from right end of the field:
std::cout << std::setw(10) << val << std::endl;
```

Esto produce:

```
10
        10
1234567890
```

(donde está la última línea para ayudar a ver las compensaciones del personaje).

A veces necesitamos establecer el ancho del campo de salida, generalmente cuando necesitamos obtener la salida en un diseño estructurado y adecuado. Eso se puede hacer usando **`std::setw`** de **std :: iomanip** .

La sintaxis de `std::setw` es:

```
std::setw(int n)
```

donde n es la longitud del campo de salida a establecer

std :: setprecision

Cuando se usa en una expresión `out << setprecision(n)` o `in >> setprecision(n)` , establece el parámetro de precisión de la transmisión o en n exactamente. El parámetro de esta función es entero, que es un nuevo valor para la precisión.

Ejemplo:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.L);
    std::cout << "default precision (6): " << pi << '\n'
              << "std::precision(10):    " << std::setprecision(10) << pi << '\n'
              << "max precision:         "
              << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
```

```
              << pi << '\n';
}
//Output
//default precision (6): 3.14159
//std::precision(10):    3.141592654
//max precision:         3.141592653589793239
```

std :: setfill

Cuando se utiliza en una expresión `out << setfill(c)` establece el carácter de relleno de la transmisión en `c` .

Nota: El carácter de relleno actual se puede obtener con `std::ostream::fill` .

Ejemplo:

```
#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
          << "setfill('*'): " << std::setfill('*')
                              << std::setw(10) << 42 << '\n';
}
//output::
//default fill:         42
//setfill('*'): ********42
```

std :: setiosflags

Cuando se utiliza en una expresión `out << setiosflags(mask)` o `in >> setiosflags(mask)` , establece todos los indicadores de formato de la transmisión hacia fuera o dentro como se especifica en la máscara.

Lista de todos los `std::ios_base::fmtflags` :

- `dec` - usar base decimal para enteros I / O
- `oct` : use la base octal para la E / S entera
- `hex` : use la base hexadecimal para la E / S entera
- `basefield` - `dec|oct|hex|0` útil para operaciones de enmascaramiento
- `left` - ajuste izquierdo (agregar caracteres de relleno a la derecha)
- `right` - ajuste a la derecha (agrega caracteres de relleno a la izquierda)
- `internal` - ajuste interno (agrega caracteres de relleno al punto interno designado)
- `adjustfield` - `left|right|internal` . Útil para operaciones de enmascaramiento.
- `scientific` : genere tipos de punto flotante utilizando notación científica o notación hexadecimal si se combina con fijos
- `fixed` : genera tipos de punto flotante usando notación fija o notación hexadecimal si se combina con
- `floatfield` - `scientific|fixed|(scientific|fixed)|0` . Útil para operaciones de enmascaramiento.
- `boolalpha` - inserte y extraiga el tipo `bool` en formato alfanumérico

- `showbase` - genera un prefijo que indica la base numérica para la salida de enteros, requiere el indicador de moneda en I / O monetaria
- `showpoint` : genere incondicionalmente un carácter de punto decimal para la salida del número de punto flotante
- `showpos` - genera un carácter `+` para salida numérica no negativa
- `skipws` : omite los espacios en blanco `skipws` antes de ciertas operaciones de entrada
- `unitbuf` descarga la salida después de cada operación de salida
- `uppercase` : reemplace ciertas letras minúsculas con sus equivalentes en mayúsculas en ciertas operaciones de salida de salida

Ejemplo de manipuladores:

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
  int l_iTemp = 47;
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::oct)<<l_iTemp<<std::endl;
  //output: 57
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::hex)<<l_iTemp<<std::endl;
  //output: 2f
  std::cout<<std::setiosflags( std::ios_base::uppercase)<<l_iTemp<<std::endl;
  //output 2F
  std::cout<<std::setfill('0')<<std::setw(12);
  std::cout<<std::resetiosflags(std::ios_base::uppercase);
  std::cout<<std::setiosflags( std::ios_base::right)<<l_iTemp<<std::endl;
  //output: 00000000002f

  std::cout<<std::resetiosflags(std::ios_base::basefield|std::ios_base::adjustfield);
  std::cout<<std::setfill('.')<<std::setw(10);
  std::cout<<std::setiosflags( std::ios_base::left)<<l_iTemp<<std::endl;
  //output: 47........

  std::cout<<std::resetiosflags(std::ios_base::adjustfield)<<std::setfill('#');
  std::cout<<std::setiosflags(std::ios_base::internal|std::ios_base::showpos);
  std::cout<<std::setw(10)<<l_iTemp<<std::endl;
  //output +#######47

  double l_dTemp = -1.2;
  double pi = 3.14159265359;
  std::cout<<pi<<"    "<<l_dTemp<<std::endl;
  //output +3.14159   -1.2
  std::cout<<std::setiosflags(std::ios_base::showpoint)<<l_dTemp<<std::endl;
  //output -1.20000
  std::cout<<setiosflags(std::ios_base::scientific)<<pi<<std::endl;
  //output: +3.141593e+00
  std::cout<<std::resetiosflags(std::ios_base::floatfield);
  std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
  //output: +3.141593
  bool b = true;
  std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
  //output: true
  return 0;
}
```

Lea std :: iomanip en línea: https://riptutorial.com/es/cplusplus/topic/6936/std----iomanip

Capítulo 128: std :: map

Observaciones

- Para usar cualquiera de `std::map` o `std::multimap` debe incluir el archivo de encabezado `<map>` .
- `std::map` y `std::multimap` mantienen sus elementos ordenados según el orden ascendente de las teclas. En el caso de `std::multimap` , no se produce una clasificación para los valores de la misma clave.
- La diferencia básica entre `std::map` y `std::multimap` es que `std::map` one no permite valores duplicados para la misma clave que `std::multimap` .
- Los mapas se implementan como árboles binarios de búsqueda. Entonces `search()` , `insert()` , `erase()` toma Θ (log n) tiempo en promedio. Para una operación de tiempo constante use `std::unordered_map` .
- `size()` funciones `size()` y `empty()` tienen una complejidad de tiempo Θ (1), el número de nodos se almacena en caché para evitar recorrer el árbol cada vez que se llama a estas funciones.

Examples

Elementos de acceso

Un `std::map` toma pares `(key, value)` como entrada.

Considere el siguiente ejemplo de `std::map` initialization:

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),
                                        std::make_pair("docs-beta", 1) };
```

En un `std::map` , los elementos se pueden insertar de la siguiente manera:

```
ranking["stackoverflow"]=2;
ranking["docs-beta"]=1;
```

En el ejemplo anterior, si el `stackoverflow` llaves ya está presente, su valor se actualizará a 2. Si no está ya presente, se creará una nueva entrada.

En un `std::map` , se puede acceder a los elementos directamente dando la clave como un índice:

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

Tenga en cuenta que usar el `operator[]` en el mapa *insertará un nuevo valor* con la clave

consultada en el mapa. Esto significa que no puede usarlo en un `const std::map` , incluso si la clave ya está almacenada en el mapa. Para evitar esta inserción, verifique si el elemento existe (por ejemplo, utilizando `find()`) o use `at()` como se describe a continuación.

C ++ 11

Se puede acceder `at()` elementos de un `std::map` con `at()` :

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

Tenga `at()` cuenta que `at()` lanzará una excepción `std::out_of_range` si el contenedor no contiene el elemento solicitado.

En ambos contenedores `std::map` y `std::multimap` , se puede acceder a los elementos utilizando los iteradores:

C ++ 11

```
// Example using begin()
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };
auto it = mmp.begin();
std::cout << it->first << " : " << it->second << std::endl; // Output: "1 : docs-beta"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackoverflow"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackexchange"

// Example using rbegin()
std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
auto it2 = mp.rbegin();
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "1 : docs-beta"
```

Inicializando un std :: map o std :: multimap

`std::map` y `std::multimap` pueden inicializarse proporcionando pares clave-valor separados por comas. Los pares clave-valor pueden ser proporcionados por `{key, value}` o pueden ser creados explícitamente por `std::make_pair(key, value)` . Como `std::map` no permite claves duplicadas y el operador de coma se realiza de derecha a izquierda, el par a la derecha se sobrescribiría con el par con la misma tecla a la izquierda.

```
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
// 2 stackexchange
```

```
std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
```

Ambos podrían inicializarse con iterador.

```
// From std::map or std::multimap iterator
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                                {6, 7} };
                        // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //moved cursor on first {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//From std::pair array
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//From std::vector of std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                         // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Borrando elementos

Eliminando todos los elementos:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap
```

Eliminando elementos de algún lugar con la ayuda de iterador:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // moved cursor on first {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}
```

Eliminando todos los elementos en un rango:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;
it++; //moved first cursor on first {3, 4}
std::advance(it2,3);  //moved second cursor on first {6, 5}
mmp.erase(it,it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

Eliminando todos los elementos que tengan un valor proporcionado como clave:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

Eliminando elementos que satisfacen un `pred` predicado:

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
   if (pred(*it))
       it = m.erase(it);
   else
       ++it;
}
```

Insertando elementos

Un elemento se puede insertar en un `std::map` solo si su clave ya no está presente en el mapa. Dado por ejemplo:

```
std::map< std::string, size_t > fruits_count;
```

- Un par clave-valor se inserta en un `std::map` través de la función miembro `insert()` . Requiere un `pair` como argumento:

  ```
  fruits_count.insert({"grapes", 20});
  fruits_count.insert(make_pair("orange", 30));
  fruits_count.insert(pair<std::string, size_t>("banana", 40));
  fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));
  ```

  La función `insert()` devuelve un `pair` consiste en un iterador y un valor `bool` :

  - Si la inserción fue exitosa, el iterador apunta al elemento recién insertado, y el valor `bool` es `true` .
  - Si ya existía un elemento con la misma `key` , la inserción falla. Cuando eso sucede, el iterador apunta al elemento que causa el conflicto, y el valor de `bool` es `false` .

  El siguiente método se puede utilizar para combinar la inserción y la operación de búsqueda:

  ```
  auto success = fruits_count.insert({"grapes", 20});
  if (!success.second) {           // we already have 'grapes' in the map
      success.first->second += 20; // access the iterator to update the value
  }
  ```

- Para mayor comodidad, el contenedor `std::map` proporciona el operador de subíndices para acceder a los elementos del mapa e insertar otros nuevos si no existen:

```
fruits_count["apple"] = 10;
```

Aunque es más simple, evita que el usuario verifique si el elemento ya existe. Si falta un elemento, `std::map::operator[]` crea implícitamente, inicializándolo con el constructor predeterminado antes de sobrescribirlo con el valor suministrado.

- `insert()` se puede usar para agregar varios elementos a la vez usando una lista de pares. Esta versión de insert () devuelve void:

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` también se puede usar para agregar elementos mediante el uso de iteradores que denotan el comienzo y el final de los valores de `value_type` valor:

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

Ejemplo:

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // insert an element with 'fruit' as key and '1' as value
    // (if the key is already stored in fruits_count, insert does nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){            // 'fruit' is already in the map
        ++ret.first->second;    // increment the counter
    }
}
```

La complejidad del tiempo para una operación de inserción es O (log n) porque `std::map` se implementa como árboles.

C ++ 11

Un `pair` puede construirse explícitamente usando `make_pair()` y `emplace()` :

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

Si sabemos dónde se insertará el nuevo elemento, entonces podemos usar `emplace_hint()` para especificar una `hint` iterador. Si el nuevo elemento se puede insertar justo antes de la `hint` , entonces la inserción se puede hacer en un tiempo constante. De lo contrario, se comporta de la misma manera que `emplace()` :

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // get iterator to the inserted element
// the next element will be before "Barry Bonds", so it is inserted before 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

Iterando sobre std :: map o std :: multimap

`std::map` o `std::multimap` se puede recorrer de las siguientes maneras:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

//Range based loop - since C++11
for(const auto &x: mmp)
    std::cout<< x.first <<":"<< x.second << std::endl;

//Forward iterator for loop: it would loop through first element to last element
//it will be a std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
std::cout<< it->first <<":"<< it->second << std::endl; //Do something with iterator

//Backward iterator for loop: it would loop through last element to first element
//it will be a std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
std::cout<< it->first <<" "<< it->second << std::endl; //Do something with iterator
```

Mientras se recorre un `std::map` o un `std::multimap` , se prefiere el uso de `auto` para evitar conversiones implícitas inútiles (consulte esta respuesta SO para obtener más detalles).

Buscando en std :: map o en std :: multimap

Hay varias formas de buscar una clave en `std::map` o en `std::multimap` .

- Para obtener el iterador de la primera aparición de una clave, se puede usar la función `find()` . Devuelve `end()` si la clave no existe.

  ```
  std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
  auto it = mmp.find(6);
  if(it!=mmp.end())
      std::cout << it->first << ", " << it->second << std::endl; //prints: 6, 5
  else
      std::cout << "Value does not exist!" << std::endl;

  it = mmp.find(66);
  if(it!=mmp.end())
      std::cout << it->first << ", " << it->second << std::endl;
  else
      std::cout << "Value does not exist!" << std::endl; // This line would be executed.
  ```

- Otra forma de averiguar si existe una entrada en `std::map` o en `std::multimap` es usar la función `count()` , que cuenta cuántos valores están asociados con una clave determinada. Como `std::map` asocia solo un valor con cada tecla, su función `count()` solo puede devolver 0 (si la tecla no está presente) o 1 (si lo está). Para `std::multimap` , `count()` puede devolver valores mayores que 1 ya que puede haber varios valores asociados con la misma clave.

  ```
  std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
  if(mp.count(3) > 0) // 3 exists as a key in map
      std::cout << "The key exists!" << std::endl; // This line would be executed.
  else
  ```

```
    std::cout << "The key does not exist!" << std::endl;
```

Si solo le importa si existe algún elemento, `find` es estrictamente mejor: documenta su intención y, para los `multimaps` , puede detenerse una vez que se haya encontrado el primer elemento coincidente.

- En el caso de `std::multimap` , podría haber varios elementos que tengan la misma clave. Para obtener este rango, se utiliza la función `equal_range()` que devuelve `std::pair` con un límite inferior (inclusivo) del iterador y un límite superior (exclusivo) respectivamente. Si la clave no existe, ambos iteradores apuntarían a `end()` .

```
auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
    // prints: 6, 5
    //         6, 7
```

Comprobando el número de elementos

El contenedor `std::map` tiene una función miembro `empty()` , que devuelve `true` o `false` , dependiendo de si el mapa está vacío o no. El `size()` función miembro `size()` devuelve el número de elementos almacenados en un contenedor `std::map` :

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Number of elements in the rank map: " << rank.size() << std::endl;
}
else{
    std::cout << "The rank map is empty" << std::endl;
}
```

Tipos de mapas

Mapa regular

Un mapa es un contenedor asociativo, que contiene pares clave-valor.

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

En el ejemplo anterior, `std::string` es el tipo de *clave* , y `size_t` es un *valor* .

La clave actúa como un índice en el mapa. Cada clave debe ser única y debe ser ordenada.

- Si necesita elementos múltiples con la misma clave, considere usar el `multimap` (que se explica a continuación)

- Si su tipo de valor no especifica ningún pedido, o si desea anular el orden predeterminado, puede proporcionar uno:

```
#include <string>
#include <map>
#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
               //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;
```

  Si el comparador `StrLess` devuelve `false` para dos claves, se consideran iguales incluso si sus contenidos reales difieren.

Multi-Mapa

Multimap permite que múltiples pares clave-valor con la misma clave se almacenen en el mapa. De lo contrario, su interfaz y creación es muy similar al mapa regular.

```
#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;
```

Hash-Map (Mapa desordenado)

Un mapa hash almacena pares clave-valor similares a un mapa normal. Sin embargo, no ordena los elementos con respecto a la clave. En su lugar, se utiliza un valor hash para la clave para acceder rápidamente a los pares clave-valor necesarios.

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

Los mapas desordenados suelen ser más rápidos, pero los elementos no se almacenan en ningún orden predecible. Por ejemplo, iterar sobre todos los elementos en un `unordered_map` da los elementos en un orden aparentemente aleatorio.

Creando std :: map con tipos definidos por el usuario como clave

Para poder utilizar una clase como la clave en un mapa, todo lo que se requiere de la clave es que sea `copiable` y `assignable` . El orden dentro del mapa se define por el tercer argumento de la plantilla (y el argumento del constructor, si se usa). Por *defecto,* este es `std::less<KeyType>` , que por defecto es el operador `<` , pero no hay ningún requisito para usar los valores por defecto. Simplemente escriba un operador de comparación (preferiblemente como un objeto funcional):

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        //  ...
    }
};
```

En C ++, el predicado de "comparación" debe ser un ordenamiento estricto y débil . En particular, `compare(X,X)` debe devolver `false` para cualquier `X` es decir, si `CmpMyType()(a, b)` devuelve true, `CmpMyType()(b, a)` debe devolver false, y si ambos devuelven false, los elementos se consideran iguales (miembros de la misma clase de equivalencia).

Ordenamiento estricto y débil

Este es un término matemático para definir una relación entre dos objetos.
Su definición es:

> Dos objetos x e y son equivalentes si f (x, y) y f (y, x) son falsos. Tenga en cuenta que un objeto es siempre (por la invariante irreflexividad) equivalente a sí mismo.

En términos de C ++, esto significa que si tiene dos objetos de un tipo determinado, debe devolver los siguientes valores en comparación con el operador <.

```
X    a;
X    b;

Condition:                  Test:     Result
a is equivalent to b:       a < b     false
a is equivalent to b        b < a     false

a is less than b            a < b     true
a is less than b            b < a     false

b is less than a            a < b     false
b is less than a            b < a     true
```

La forma en que defina el equivalente / menor depende totalmente del tipo de su objeto.

Lea std :: map en línea: https://riptutorial.com/es/cplusplus/topic/681/std----map

Capítulo 129: std :: opcional

Examples

Introducción

Los opcionales (también conocidos como tipos Maybe) se utilizan para representar un tipo cuyo contenido puede o no estar presente. Se implementan en C ++ 17 como la clase `std::optional` . Por ejemplo, un objeto de tipo `std::optional<int>` puede contener algún valor de tipo `int` , o puede que no contenga ningún valor.

Los opcionales se utilizan comúnmente para representar un valor que puede no existir o como un tipo de retorno de una función que puede fallar en devolver un resultado significativo.

Otros enfoques opcionales

Hay muchos otros enfoques para resolver el problema que resuelve `std::optional` , pero ninguno de ellos es completo: usar un puntero, usar un centinela o usar un `pair<bool, T>` .

Opcional vs puntero

En algunos casos, podemos proporcionar un puntero a un objeto existente o `nullptr` para indicar un error. Pero esto se limita a aquellos casos en los que ya existen objetos: `optional` , como tipo de valor, también se puede usar para devolver nuevos objetos sin tener que recurrir a la asignación de memoria.

Opcional vs Sentinel

Un lenguaje común es usar un valor especial para indicar que el valor no tiene sentido. Esto puede ser 0 o -1 para tipos integrales, o `nullptr` para punteros. Sin embargo, esto reduce el espacio de valores válidos (no puede diferenciar entre un 0 válido y un 0 sin sentido) y muchos tipos no tienen una opción natural para el valor centinela.

Opcional vs `std::pair<bool, T>`

Otro modismo común es proporcionar un par, donde uno de los elementos es un `bool` indica si el valor es significativo o no.

Esto se basa en que el tipo de valor es constructible por defecto en el caso de error, que no es posible para algunos tipos y es posible pero no deseable para otros. Un `optional<T>` , en el caso de error, no necesita construir nada.

Usando opcionales para representar la ausencia de un valor.

Antes de C ++ 17, tener punteros con un valor de `nullptr` representaba comúnmente la ausencia

de un valor. Esta es una buena solución para objetos grandes que han sido asignados dinámicamente y ya están gestionados por punteros. Sin embargo, esta solución no funciona bien para tipos pequeños o primitivos, como `int` , que rara vez se asignan o administran dinámicamente por punteros. `std::optional` proporciona una solución viable a este problema común.

En este ejemplo, se define `struct Person` . Es posible que una persona tenga una mascota, pero no es necesario. Por lo tanto, el miembro `pet` de `Person` se declara con un envoltorio `std::optional` .

```
#include <iostream>
#include <optional>
#include <string>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " is alone." << std::endl;
    }
}
```

Usando opcionales para representar el fallo de una función.

Antes de C ++ 17, una función representaba el error de varias maneras:

- Se devolvió un puntero nulo.
  - por ejemplo, llamar a una función `Delegate *App::get_delegate()` en una instancia de la `App` que no tenía un delegado devolvería `nullptr` .
  - Esta es una buena solución para los objetos que han sido asignados dinámicamente o que son grandes y están administrados por punteros, pero no es una buena solución para los objetos pequeños que normalmente se asignan y se pasan copiando.
- Se reservó un valor específico del tipo de retorno para indicar el fallo.
  - por ejemplo, llamar a una función `unsigned shortest_path_distance(Vertex a, Vertex b)` en dos vértices que no están conectados puede devolver cero para indicar este hecho.
- El valor se emparejó junto con un `bool` para indicar si el valor devuelto fue significativo.
  - por ejemplo, llamar a una función `std::pair<int, bool> parse(const std::string &str)` con un argumento de cadena que no es un número entero devolvería un par con un `int` indefinido y un `bool` establecido en `false` .

En este ejemplo, John recibe dos mascotas, Fluffy y Furball. Luego se llama a la función `Person::pet_with_name()` para recuperar los bigotes de la mascota de John. Como John no tiene una mascota llamada Whiskers, la función falla y `std::nullopt` se devuelve en su lugar.

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;

    std::optional<Animal> pet_with_name(const std::string &name) {
        for (const Animal &pet : pets) {
            if (pet.name == name) {
                return pet;
            }
        }
        return std::nullopt;
    }
};

int main() {
    Person john;
    john.name = "John";

    Animal fluffy;
    fluffy.name = "Fluffy";
    john.pets.push_back(fluffy);

    Animal furball;
    furball.name = "Furball";
    john.pets.push_back(furball);

    std::optional<Animal> whiskers = john.pet_with_name("Whiskers");
    if (whiskers) {
        std::cout << "John has a pet named Whiskers." << std::endl;
    }
    else {
        std::cout << "Whiskers must not belong to John." << std::endl;
    }
}
```

opcional como valor de retorno

```
std::optional<float> divide(float a, float b) {
  if (b!=0.f) return a/b;
  return {};
}
```

Aquí devolvemos la fracción `a/b` , pero si no está definida (sería infinito), devolvemos el opcional vacío.

Un caso más complejo:

```
template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
  using std::begin; using std::end;
  auto b = begin(r), e = end(r);
  auto r = std::find_if(b, e , p );
  using iterator = decltype(r);
  if (r==e)
    return std::optional<iterator>();
  return std::optional<iterator>(r);
}
template<class Range, class T>
auto find( Range&& r, T const& t ) {
  return find_if( std::forward<Range>(r), [&t](auto&& x){return x==t;} );
}
```

`find( some_range, 7 )` busca en el contenedor o rango `some_range` para algo igual al número `7` . `find_if` hace con un predicado.

Devuelve un opcional vacío si no fue encontrado, o un opcional que contiene un iterador para el elemento si fue encontrado.

Esto te permite hacer:

```
if (find( vec, 7 )) {
  // code
}
```

o incluso

```
if (auto oit = find( vec, 7 )) {
  vec.erase(*oit);
}
```

sin tener que meterse con iteradores y pruebas de inicio / finalización.

valor_o

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {
  std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}
```

`value_or` devuelve el valor almacenado en el opcional o el argumento si no hay nada almacenado allí.

Esto le permite tomar el opcional tal vez nulo y dar un comportamiento predeterminado cuando realmente necesita un valor. Al hacerlo de esta manera, la decisión de "comportamiento predeterminado" se puede retrasar hasta el punto en que se realice mejor y se necesite de inmediato, en lugar de generar algún valor predeterminado en las entrañas de algún motor.

Lea std :: opcional en línea: https://riptutorial.com/es/cplusplus/topic/2423/std----opcional

Capítulo 130: std :: par

Examples

Creando un par y accediendo a los elementos.

El par nos permite tratar dos objetos como un solo objeto. Los pares se pueden construir fácilmente con la ayuda de la función de plantilla `std::make_pair` .

Una forma alternativa es crear un par y asignar sus elementos (`first` y `second`) más tarde.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //Creating the pair
    std::cout << p.first << " " << p.second << std::endl; //Accessing the elements




    //We can also create a pair and assign the elements later
    std::pair<int,int> p1;
    p1.first = 3;
    p1.second = 4;
    std::cout << p1.first << " " << p1.second << std::endl;

    //We can also create a pair using a constructor
    std::pair<int,int> p2 = std::pair<int,int>(5, 6);
    std::cout << p2.first << " " << p2.second << std::endl;

    return 0;
}
```

Comparar operadores

Los parámetros de estos operadores son `lhs` y `rhs`

- `operator==` comprueba si ambos elementos en el par de `lhs` y `rhs` son iguales. El valor de retorno es `true` si ambos `lhs.first == rhs.first` Y `lhs.second == rhs.second` , de lo contrario `false`

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);

if (p1 == p2)
    std::cout << "equals";
else
    std::cout << "not equal"//statement will show this, because they are not identical
```

- `operator!=` prueba si algún elemento en el par de `lhs` y `rhs` no es igual. El valor de retorno es

`true` si `lhs.first != rhs.first` O `lhs.second != rhs.second` , de lo contrario, devuelve `false` .

- `operator<` prueba si `lhs.first<rhs.first` , devuelve `true` . De lo contrario, si `rhs.first<lhs.first` devuelve `false` . De lo contrario, si `lhs.second<rhs.second` devuelve `true` , de lo contrario, devuelve `false` .
- `operator<=` devuelve `!(rhs<lhs)`
- `operator>` devuelve `rhs<lhs`
- `operator>=` devuelve `!(lhs<rhs)`

Otro ejemplo con contenedores de pares. Utiliza el `operator<` porque necesita ordenar el contenedor.

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"},
                                                   {2, "bar"},
                                                   {1, "foo"} };
    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

Lea std :: par en línea: https://riptutorial.com/es/cplusplus/topic/4834/std----par

Capítulo 131: std :: set y std :: multiset

Introducción

`set` es un tipo de contenedor cuyos elementos están ordenados y son únicos. `multiset` es similar, pero, en el caso de `multiset` , múltiples elementos pueden tener el mismo valor.

Observaciones

Se han usado diferentes estilos de C ++ en esos ejemplos. Tenga cuidado de que si está utilizando un compilador C ++ 98; Es posible que parte de este código no sea utilizable.

Examples

Insertando valores en un conjunto

Se pueden utilizar tres métodos diferentes de inserción con conjuntos.

- Primero, una simple inserción del valor. Este método devuelve un par que permite a la persona que llama verificar si realmente se produjo la inserción.
- Segundo, una inserción dando una pista de donde se insertará el valor. El objetivo es optimizar el tiempo de inserción en tal caso, pero saber dónde debe insertarse un valor no es el caso común. **Ten cuidado en ese caso; La forma de dar una pista difiere con las versiones del compilador** .
- Finalmente, puede insertar un rango de valores dando un puntero inicial y uno final. El inicio se incluirá en la inserción, el final se excluye.

```
#include <iostream>
#include <set>

int main ()
{
  std::set<int> sut;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;

  // Basic insert
  sut.insert(7);
  sut.insert(5);
  sut.insert(12);

  ret = sut.insert(23);
  if (ret.second==true)
    std::cout << "# 23 has been inserted!" << std::endl;

  ret = sut.insert(23); // since it's a set and 23 is already present in it, this insert
should fail
  if (ret.second==false)
    std::cout << "# 23 already present in set!" << std::endl;
```

```
  // Insert with hint for optimization
  it = sut.end();
  // This case is optimized for C++11 and above
  // For earlier version, point to the element preceding your insertion
  sut.insert(it, 30);

  // inserting a range of values
  std::set<int> sut2;
  sut2.insert(20);
  sut2.insert(30);
  sut2.insert(45);
  std::set<int>::iterator itStart = sut2.begin();
  std::set<int>::iterator itEnd = sut2.end();

  sut.insert (itStart, itEnd); // second iterator is excluded from insertion

  std::cout << std::endl << "Set under test contains:" << std::endl;
  for (it = sut.begin(); it != sut.end(); ++it)
  {
    std::cout << *it << std::endl;
  }

  return 0;
}
```

La salida será:

```
# 23 has been inserted!

# 23 already present in set!



Set under test contains:

5

7

12

20

23

30

45
```

Insertar valores en un multiset

Todos los métodos de inserción de conjuntos también se aplican a multisets. Sin embargo, existe otra posibilidad, que es proporcionar una initial__list:

```
auto il = { 7, 5, 12 };
std::multiset<int> msut;
```

```
msut.insert(il);
```

Cambiar el tipo predeterminado de un conjunto

`set` y `multiset` tienen métodos de comparación predeterminados, pero en algunos casos es posible que deba sobrecargarlos.

Imaginemos que estamos almacenando valores de cadena en un conjunto, pero sabemos que esas cadenas contienen solo valores numéricos. Por defecto, la clasificación será una comparación de cadena lexicográfica, por lo que el orden no coincidirá con la clasificación numérica. Si desea aplicar una clasificación equivalente a la que tendría con los valores `int` , necesita un functor para sobrecargar el método de comparación:

```
#include <iostream>
#include <set>
#include <stdlib.h>

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom({"1", "2", "5", "23", "6", "290"},
                                                     custom_compare{}); //< Compare object
optional as its default constructible.

    std::cout << std::endl << "### Custom sort on set :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << std::endl;

    auto compare_via_lambda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
    set_via_lambda sut_reverse_via_lambda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lambda);

    std::cout << std::endl << "### Lambda sort on set :" << std::endl;
    for (auto &&data : sut_reverse_via_lambda)
        std::cout << data << std::endl;

    return 0;
}
```

La salida será:

```
### Default sort on std::set<std::string> :
1
2
23
290
5
6
### Custom sort on set :
1
2
5
6
23
290

### Lambda sort on set :
6
5
290
23
2
1
```

En el ejemplo anterior, uno puede encontrar 3 formas diferentes de agregar operaciones de comparación al `std::set` , cada una de ellas es útil en su propio contexto.

Orden predeterminado

Esto usará el operador de comparación de la clave (primer argumento de la plantilla). A menudo, la clave ya proporcionará un buen valor predeterminado para la función `std::less<T>` . A menos que esta función sea especializada, utiliza el `operator<` del objeto. Esto es especialmente útil cuando otro código también intenta usar algunos pedidos, ya que esto permite la coherencia en toda la base del código.

Escribir el código de esta manera reducirá el esfuerzo de actualizar su código cuando la clave cambie en API, como: una clase que contiene 2 miembros que cambia a una clase que contiene 3 miembros. Al actualizar el `operator<` en la clase, todas las ocurrencias se actualizarán.

Como es de esperar, el uso de la ordenación predeterminada es un valor predeterminado razonable.

Orden personalizado

La adición de una ordenación personalizada a través de un objeto con un operador de comparación se usa a menudo cuando la comparación predeterminada no cumple. En el ejemplo anterior, esto se debe a que las cadenas se refieren a números enteros. En otros casos, a menudo se usa cuando se desean comparar (inteligentes) los punteros en función del objeto al que hacen referencia o porque se necesitan diferentes restricciones para comparar (ejemplo: comparar `std::pair` por el valor de `first`).

Al crear un operador de comparación, esto debería ser una clasificación estable. Si el resultado

del operador de comparación cambia después de la inserción, tendrá un comportamiento indefinido. Como buena práctica, su operador de comparación solo debe usar los datos constantes (miembros const, funciones const ...).

Como en el ejemplo anterior, a menudo encontrará clases sin miembros como operadores de comparación. Esto da como resultado constructores por defecto y constructores de copia. El constructor predeterminado le permite omitir la instancia en el momento de la construcción y se requiere el constructor de copia ya que el conjunto toma una copia del operador de comparación.

Tipo lambda

Las Lambdas son una forma más corta de escribir objetos de función. Esto permite escribir el operador de comparación en menos líneas, lo que hace que el código general sea más legible.

La desventaja del uso de lambdas es que cada lambda obtiene un tipo específico en el momento de la compilación, por lo que `decltype(lambda)` será diferente para cada compilación de la misma unidad de compilación (archivo cpp) que sobre varias unidades de compilación (cuando se incluye a través del archivo de cabecera). Por esta razón, se recomienda usar objetos de función como operador de comparación cuando se usa dentro de los archivos de encabezado.

Esta construcción se encuentra a menudo cuando se usa un `std::set` dentro del alcance local de una función, mientras que el objeto de la función se prefiere cuando se usa como argumentos de la función o miembros de la clase.

Otras opciones de clasificación

Como el operador de comparación de `std::set` es un argumento de plantilla, todos los objetos que se pueden llamar se pueden usar como operador de comparación y los ejemplos anteriores son solo casos específicos. Las únicas restricciones que tienen estos objetos invocables son:

- Deben ser copiables construibles.
- Deben ser invocables con 2 argumentos del tipo de la clave. (las conversiones implícitas están permitidas, aunque no se recomiendan ya que pueden afectar el rendimiento)

Buscando valores en set y multiset

Hay varias formas de buscar un valor dado en `std::set` o en `std::multiset` :

Para obtener el iterador de la primera aparición de una clave, se puede usar la función `find()` . Devuelve `end()` si la clave no existe.

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3); // contains 3, 10, 15, 22

auto itS = sut.find(10); // the value is found, so *itS == 10
itS = sut.find(555); // the value is not found, so itS == sut.end()
```

```
  std::multiset<int> msut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(15);
  sut.insert(3); // contains 3, 10, 15, 15, 22

  auto itMS = msut.find(10);
```

Otra forma es usar la función `count()` , que cuenta cuántos valores correspondientes se han encontrado en el `set` / `set multiset` (en el caso de un `set` , el valor de retorno puede ser solo 0 o 1). Usando los mismos valores que arriba, tendremos:

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

En el caso de `std::multiset` , puede haber varios elementos que tengan el mismo valor. Para obtener este rango, se puede usar la función `equal_range()` . Devuelve `std::pair` con iterador límite inferior (inclusivo) y límite superior (exclusivo) respectivamente. Si la clave no existe, ambos iteradores apuntarían al valor superior más cercano (basado en el método de comparación usado para ordenar el `multiset` dado).

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // point to first element '15'
auto en = eqr.second; // point to element '22'

eqr = msut.equal_range(9); // both eqr.first and eqr.second point to element '10'
```

Eliminar valores de un conjunto

El método más obvio, si solo desea restablecer su conjunto / conjunto múltiple a uno vacío, es usar `clear` :

```
  std::set<int> sut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(3);
  sut.clear(); //size of sut is 0
```

Luego se puede utilizar el método de `erase` . Ofrece algunas posibilidades que parecen un tanto equivalentes a la inserción:

```
std::set<int> sut;
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
```

```
sut.insert(22);
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// Basic deletion
sut.erase(3);

// Using iterator
it = sut.find(22);
sut.erase(it);

// Deleting a range of values
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
  std::cout << *it << std::endl;
}
```

La salida será:

```
Set under test contains:

10

15

30
```

Todos esos métodos también se aplican a `multiset` . Tenga en cuenta que si solicita eliminar un elemento de un `multiset` y está presente varias veces, **se eliminarán todos los valores equivalentes** .

Lea std :: set y std :: multiset en línea: https://riptutorial.com/es/cplusplus/topic/9005/std----set-y-std----multiset

Capítulo 132: std :: string

Introducción

Las cadenas son objetos que representan secuencias de caracteres. El estándar `string` clase proporciona una alternativa sencilla, segura y versátil para la utilización de matrices explícitas de `char` s cuando se trata de texto y otras secuencias de caracteres. La clase de `string` C ++ es parte del `std` nombres estándar y se estandarizó en 1998.

Sintaxis

- *// declaración de cadena vacía*

  std :: string s;

- *// Construyendo desde const char * (c-string)*

  std :: string s ("Hello");

  std :: string s = "Hello";

- *// Construyendo usando el constructor de copia*

  std :: string s1 ("Hello");

  std :: string s2 (s1);

- *// Construyendo a partir de subcadenas*

  std :: string s1 ("Hello");

  std :: string s2 (s1, 0, 4); *// Copia 4 caracteres de la posición 0 de s1 en s2*

- *// Construyendo desde un buffer de personajes.*

  std :: string s1 ("Hello World");
  std :: string s2 (s1, 5); *// Copia los primeros 5 caracteres de s1 en s2*

- *// Construir usando constructor de relleno (solo char)*

  std :: string s (5, 'a'); *// s contiene aaaaa*

- *// Construir usando el constructor de rango e iterador*

  std :: string s1 ("Hello World");

  std :: string s2 (s1.begin (), s1.begin () + 5); *// Copia los primeros 5 caracteres de s1 en s2*

Observaciones

Antes de usar `std::string` , debe incluir la `string` del encabezado, ya que incluye funciones / operadores / sobrecargas que otros encabezados (por ejemplo, `iostream`) no incluyen.

El uso de const char * constructor con un nullptr conduce a un comportamiento indefinido.

```
std::string oops(nullptr);
std::cout << oops << "\n";
```

El método `at` lanza un `std::out_of_range` excepción si `index >= size()` .

El comportamiento del `operator[]` es un poco más complicado, en todos los casos tiene un comportamiento indefinido si `index > size()` , pero cuando `index == size()` :

C ++ 11

1. En una cadena no constante, el comportamiento *no* está *definido* ;
2. En una cadena constante, se devuelve una referencia a un carácter con valor `CharT()` (el carácter *nulo*).

C ++ 11

1. Se `CharT()` una referencia a un carácter con valor `CharT()` (el carácter *nulo*).
2. Modificar esta referencia es *un comportamiento indefinido* .

Como C ++ 14, en lugar de usar `"foo"` , se recomienda usar `"foo"s` , ya que `s` es un sufijo literal definido por el usuario , que convierte el `const char* "foo"` en `std::string "foo"` .

Nota: tienes que usar el espacio de nombres `std::string_literals` o `std::literals` `std::string_literals` *para obtener el* `s` *literal.*

Examples

Terrible

Usa `std::string::substr` para dividir una cadena. Hay dos variantes de esta función miembro.

La primera toma una *posición inicial* desde la cual debe comenzar la subcadena devuelta. La posición inicial debe ser válida en el rango `(0, str.length()]` :

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

El segundo toma una posición inicial y una *longitud* total de la nueva subcadena. Independientemente de la *longitud* , la subcadena nunca pasará del final de la cadena de origen:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(15, 3); // "and"
```

Tenga en cuenta que también puede llamar a `substr` sin argumentos, en este caso se devuelve una copia exacta de la cadena

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

Reemplazo de cuerdas

Reemplazar por posición

Para reemplazar una porción de una `std::string` puede usar el método `replace` de la `std::string` .

`replace` tiene una gran cantidad de sobrecargas útiles:

```
//Define string
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
```

C ++ 14

```
//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
```

```
//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
```

C ++ 11

```
//8)
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"
```

Reemplazar las ocurrencias de una cadena con otra cadena

Reemplace solo la primera vez que se `replace` con `with` en `str` :

```
std::string replaceString(std::string str,
                          const std::string& replace,
                          const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}
```

Reemplace todas las ocurrencias de `replace` con `with` en `str` :

```
std::string replaceStringAll(std::string str,
                             const std::string& replace,
                             const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}
```

Concatenación

Puede concatenar `std::string` s utilizando los operadores sobrecargados `+` y `+=` . Usando el operador `+` :

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

Usando el operador `+=` :

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

También puede agregar cadenas C, incluidos los literales de cadena:

```
std::string hello = "Hello";
std::string world = "world";
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

También puede utilizar `push_back()` para hacer retroceder individuo `char` s:

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

También hay `append()` , que es bastante parecido a `+=` :

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Accediendo a un personaje

Hay varias formas de extraer caracteres de una `std::string` y cada una es sutilmente diferente.

```
std::string str("Hello world!");
```

---

operador [] (n)

Devuelve una referencia al carácter en el índice n.

`std::string::operator[]` no está verificada por límites y no lanza una excepción. La persona que llama es responsable de afirmar que el índice está dentro del rango de la cadena:

```
char c = str[6]; // 'w'
```

---

en (n)

Devuelve una referencia al carácter en el índice n.

`std::string::at` *es los* límites `std::out_of_range` , y lanzará `std::out_of_range` si el índice no está dentro del rango de la cadena:

```
char c = str.at(7); // 'o'
```

C ++ 11

Nota: Ambos ejemplos darán como resultado un comportamiento indefinido si la cadena está vacía.

---

frente()

Devuelve una referencia al primer carácter:

```
char c = str.front(); // 'H'
```

---

atrás()

Devuelve una referencia al último carácter:

```
char c = str.back(); // '!'
```

Tokenizar

Listado de menos costoso a más costoso en tiempo de ejecución:

1. `str::strtok` es el método de tokenización estándar más barato, también permite que el delimitador se modifique entre tokens, pero incurre en 3 dificultades con C ++ moderno:

    - `std::strtok` no se puede usar en varias `strings` al mismo tiempo (aunque algunas implementaciones se extienden para admitir esto, como: `strtok_s`)
    - Por la misma razón, `std::strtok` no se puede usar en varios subprocesos simultáneamente (sin embargo, esto puede ser definido por la implementación, por ejemplo: la implementación de Visual Studio es segura para subprocesos)
    - Al llamar a `std::strtok` modifica la `std::string` que está funcionando, por lo que no se puede usar en `const string` `const char*` , caracteres `const char*` o cadenas literales, para señalizar cualquiera de ellas con `std::strtok` o para operar en una `std::string` cuyo contenido debe conservarse, la entrada debería copiarse, luego la copia podría operarse

    En general, cualquiera de estas opciones de costo estará oculto en el costo de asignación de los tokens, pero si se requiere el algoritmo más barato y las dificultades de `std::strtok` strtok no se pueden superar, considere una solución manual .

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Ejemplo vivo

2. `std::istream_iterator` usa el operador de extracción del flujo de forma iterativa. Si la entrada `std::string` está delimitada por espacios en blanco, esto es capaz de expandirse en la opción `std::strtok` eliminando sus dificultades, permitiendo la tokenización en línea, lo que permite la generación de un `const vector<string>` , y agregando soporte para múltiples delimitando caracteres de espacio en blanco:

```
// String to tokenize
const std::string str("The  quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
                                        std::istream_iterator<std::string>(is),
```

```
                                        std::istream_iterator<std::string>());
```

Ejemplo vivo

3. El `std::regex_token_iterator` utiliza un `std::regex` para tokenizar iterativamente. Proporciona una definición de delimitador más flexible. Por ejemplo, comas no delimitadas y espacios en blanco:

C ++ 11

```
// String to tokenize
const std::string str{ "The ,qu\\,ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\\\\,]|\\\\.)*?)\\s*(?:,|$)" };
// Vector to store tokens
const std::vector<std::string> tokens{
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),
    std::sregex_token_iterator()
};
```

Ejemplo vivo

Vea el ejemplo `regex_token_iterator` para más detalles.

Conversión a (const) char *

Para obtener acceso `const char*` a los datos de un `std::string` , puede usar la función miembro `c_str()` del string. Tenga en cuenta que el puntero solo es válido mientras el objeto `std::string` esté dentro del alcance y permanezca sin cambios, lo que significa que solo se puede llamar a los métodos `const` en el objeto.

C ++ 17

La función miembro de `data()` se puede usar para obtener un `char*` modificable, que se puede usar para manipular los datos del objeto `std::string` .

C ++ 11

También se puede obtener un `char*` modificable tomando la dirección del primer carácter: `&s[0]` . En C ++ 11, se garantiza que esto produce una cadena bien formada, terminada en nulo. Tenga en cuenta que `&s[0]` está bien formado incluso si `s` está vacío, mientras que `&s.front()` no está definido si `s` está vacío.

C ++ 11

```
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr points to: "This is a string.\0"
const char* data = str.data();  // data points to: "This is a string.\0"
```

```
std::string str("This is a string.");

// Copy the contents of str to untie lifetime from the std::string object
```

```
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);

// Alternative to the line above (no exception safety):
// char* cstr_unsafe = new char[str.size() + 1];

std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // A null-terminator needs to be added

// delete[] cstr_unsafe;
std::cout << cstr.get();
```

Encontrar caracteres en una cadena

Para encontrar un carácter u otra cadena, puede usar `std::string::find` . Devuelve la posición del primer carácter de la primera partida. Si no se encontraron coincidencias, la función devuelve `std::string::npos`

```
std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

Encontrado en la posición: 21

Las oportunidades de búsqueda se amplían aún más por las siguientes funciones:

```
find_first_of     // Find first occurrence of characters
find_first_not_of // Find first absence of characters
find_last_of      // Find last occurrence of characters
find_last_not_of  // Find last absence of characters
```

Estas funciones pueden permitirle buscar caracteres desde el final de la cadena, así como encontrar el caso negativo (es decir, los caracteres que no están en la cadena). Aquí hay un ejemplo:

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

Encontrado en la posición: 6

Nota: tenga en cuenta que las funciones anteriores no buscan subcadenas, sino caracteres contenidos en la cadena de búsqueda. En este caso, la última aparición de `'g'` se encontró en la posición `6` (los otros caracteres no se encontraron).

Recorte de caracteres al inicio / final

Este ejemplo requiere los encabezados `<algorithm>` , `<locale>` y `<utility>` .

C ++ 11

Recortar una secuencia o cadena significa eliminar todos los elementos (o caracteres) iniciales y finales que coinciden con un determinado predicado. Primero recortamos los elementos finales, porque no implica mover ningún elemento, y luego recortamos los elementos iniciales. Tenga en cuenta que las generalizaciones siguientes funcionan para todos los tipos de `std::basic_string` (por ejemplo, `std::string` y `std::wstring`), y accidentalmente también para contenedores de secuencias (por ejemplo, `std::vector` y `std::list`).

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred>     // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

Recortar los elementos finales implica encontrar el *último* elemento que no coincide con el predicado y borrar de allí en adelante:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                 seq.rend(),
                                 pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Recortar los elementos iniciales implica encontrar el *primer* elemento que no coincide con el predicado y borrar hasta allí:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

Para especializar lo anterior para recortar espacios en blanco en una `std::string` `std::isspace()` podemos usar la función `std::isspace()` como predicado:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

De manera similar, podemos usar la función `std::iswspace()` para `std::wstring` etc.

Si desea crear una *nueva* secuencia que sea una copia recortada, puede usar una función separada:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Comparacion lexicografica

Dos `std::string` s se pueden comparar lexicográficamente usando los operadores `==` `!=` , `<` , `<=` , `>` Y `>=` :

```
std::string str1 = "Foo";
std::string str2 = "Bar";

assert(!(str1 < str2));
assert(str > str2);
assert(!(str1 <= str2));
assert(str1 >= str2);
assert(!(str1 == str2));
assert(str1 != str2);
```

Todas estas funciones utilizan el método `std::string::compare()` subyacente para realizar la comparación, y devuelven por conveniencia valores booleanos. El funcionamiento de estas funciones se puede interpretar de la siguiente manera, independientemente de la implementación real:

- operador `==` :

  Si `str1.length() == str2.length()` y cada par de caracteres coinciden, devuelve `true` , de lo contrario devuelve `false` .

- operador `!=` :

  Si `str1.length() != str2.length()` o un par de caracteres no coincide, devuelve `true` , de lo contrario, devuelve `false` .

- operador `<` u operador `>` :

  Busca el primer par de caracteres diferentes, los compara y luego devuelve el resultado booleano.

- operador `<=` u operador `>=` :

  Busca el primer par de caracteres diferentes, los compara y luego devuelve el resultado booleano.

Nota: el término **par de caracteres** significa los caracteres correspondientes en ambas cadenas de las mismas posiciones. Para una mejor comprensión, si dos cadenas de ejemplo son `str1` y `str2`

, y sus longitudes son `n` y `m` respectivamente, entonces los pares de caracteres de ambas cadenas significan cada par `str1[i]` y `str2[i]` donde *i = 0, 1, 2,. .., max (n, m)* . Si por alguna donde no existe *i* el carácter correspondiente, es decir, cuando *i* es mayor o igual a `n` o `m` , sería considerado como el valor más bajo.

Aquí hay un ejemplo del uso de `<` :

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

Los pasos son los siguientes:

1. Compara los primeros caracteres, `'B' == 'B'` - sigue adelante.
2. Compara los segundos caracteres, `'a' == 'a'` - sigue adelante.
3. Compara los terceros caracteres, `'r' == 'r'` - sigue adelante.
4. El rango `str2` ahora está agotado, mientras que el rango `str1` aún tiene caracteres. Así, `str2 < str1` .

Conversión a std :: wstring

En C ++, las secuencias de caracteres se representan especializando la clase `std::basic_string` con un tipo de carácter nativo. Las dos colecciones principales definidas por la biblioteca estándar son `std::string` y `std::wstring` :

- `std::string` está construido con elementos de tipo `char`
- `std::wstring` está construido con elementos de tipo `wchar_t`

Para convertir entre los dos tipos, use `wstring_convert` :

```
#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);

std::string wstr_turned_to_str =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

Para mejorar la facilidad de uso y / o la legibilidad, puede definir funciones para realizar la conversión:

```
#include <string>
#include <codecvt>
```

```
#include <locale>

using convert_t = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_t, wchar_t> strconverter;

std::string to_string(std::wstring wstr)
{
    return strconverter.to_bytes(wstr);
}

std::wstring to_wstring(std::string str)
{
    return strconverter.from_bytes(str);
}
```

Uso de la muestra:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

Eso es ciertamente más legible que
`std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` .

Tenga en cuenta que `char` y `wchar_t` no implican codificación, y no da ninguna indicación de tamaño en bytes. Por ejemplo, `wchar_t` se implementa comúnmente como un tipo de datos de 2 bytes y generalmente contiene datos codificados en UTF-16 en Windows (o UCS-2 en versiones anteriores a Windows 2000) y como un tipo de datos de 4 bytes codificados en UTF-32 en Linux Esto contrasta con los tipos más nuevos `char16_t` y `char32_t` , que se introdujeron en C ++ 11 y se garantiza que son lo suficientemente grandes como para contener cualquier "carácter" UTF16 o UTF32 (o, más precisamente, el *punto de código*) respectivamente.

Usando la clase std :: string_view

C ++ 17

C ++ 17 introduce `std::string_view` , que es simplemente un rango no propietario de caracteres `const char` , implementable como un par de punteros o un puntero y una longitud. Es un tipo de parámetro superior para funciones que requieren datos de cadena no modificables. Antes de C ++ 17, había tres opciones para esto:

```
void foo(std::string const& s);      // pre-C++17, single argument, could incur
                                     // allocation if caller's data was not in a string
                                     // (e.g. string literal or vector<char> )

void foo(const char* s, size_t len); // pre-C++17, two arguments, have to pass them
                                     // both everywhere

void foo(const char* s);             // pre-C++17, single argument, but need to call
                                     // strlen()

template <class StringT>
void foo(StringT const& s);          // pre-C++17, caller can pass arbitrary char data
                                     // provider, but now foo() has to live in a header
```

Todos estos pueden ser reemplazados por:

```
void foo(std::string_view s);        // post-C++17, single argument, tighter coupling
                                     // zero copies regardless of how caller is storing
                                     // the data
```

Tenga en cuenta que `std::string_view` ***no puede*** *modificar sus datos subyacentes* .

`string_view` es útil cuando quiere evitar copias innecesarias.

Ofrece un subconjunto útil de la funcionalidad que hace `std::string` , aunque algunas de las funciones se comportan de manera diferente:

```
std::string str = "lllloooonnnnggggg sssstttrrriiinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

Recorriendo cada personaje

C ++ 11

`std::string` admite iteradores, por lo que puede utilizar un bucle *basado* en rangos para recorrer cada carácter:

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

Puedes usar un bucle tradicional `for` cada bucle:

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Conversión a enteros / tipos de punto flotante

Una `std::string` contiene un número se puede convertir en un tipo entero, o un tipo de punto flotante, usando funciones de conversión.

Tenga en cuenta que todas estas funciones dejan de analizar la cadena de entrada tan pronto como encuentran un carácter no numérico, por lo que `"123abc"` se convertirá en `123` .

La familia de funciones `std::ato*` convierte cadenas de estilo C (matrices de caracteres) en tipos

enteros o de punto flotante:

```
std::string ten = "10";

double num1 = std::atof(ten.c_str());
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
```

C ++ 11

```
long long num4 = std::atoll(ten.c_str());
```

Sin embargo, se desaconseja el uso de estas funciones porque devuelven `0` si no pueden analizar la cadena. Esto es malo porque `0` también podría ser un resultado válido, si, por ejemplo, la cadena de entrada era "0", por lo que es imposible determinar si la conversión realmente falló.

La nueva familia de funciones `std::sto*` convierte `std::string` s en tipos enteros o de punto flotante, y lanza excepciones si no pueden analizar su entrada. *Deberías usar estas funciones si es posible* :

C ++ 11

```
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Además, estas funciones también manejan cadenas octales y hexagonales a diferencia de la familia `std::ato*` . El segundo parámetro es un puntero al primer carácter no convertido en la cadena de entrada (no ilustrado aquí), y el tercer parámetro es la base a utilizar. `0` es la detección automática de octal (comenzando con `0`) y hexadecimal (comenzando con `0x` o `0X`), y cualquier otro valor es la base para usar

```
std::string ten = "10";
std::string ten_octal = "12";
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16);  // Returns 10
long num4 = std::stol(ten_hex);  // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Convertir entre codificaciones de caracteres.

La conversión entre codificaciones es fácil con C ++ 11 y la mayoría de los compiladores pueden tratarla de forma multiplataforma a través de los `<codecvt>` y `<locale>` .

```
#include <iostream>
#include <codecvt>
#include <locale>
#include <string>
using namespace std;

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between u16string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

Tenga en cuenta que Visual Studio 2015 proporciona soporte para estas conversiones, pero un error en la implementación de su biblioteca requiere el uso de una plantilla diferente para `wstring_convert` cuando se trata de `char16_t` :

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
  std::basic_string<utf16_char> tmp;
  tmp.resize(utf16.length());
  std::copy(utf16.begin(), utf16.end(), tmp.begin());
  utf8 = conv_utf8_utf16.to_bytes(tmp);
}
void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
  std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
  utf16.clear();
  utf16.resize(tmp.length());
  std::copy(tmp.begin(), tmp.end(), utf16.begin());
}
```

Comprobando si una cadena es un prefijo de otra

C ++ 14

En C ++ 14, esto se hace fácilmente mediante `std::mismatch` que devuelve el primer par no coincidente de dos rangos:

```
std::string prefix = "foo";
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();
```

Tenga en cuenta que existía una versión con rango y medio de `mismatch()` anterior a C ++ 14, pero esto no es seguro en el caso de que la segunda cadena sea la más corta de las dos.

C ++ 14

Todavía podemos usar la versión de rango y medio de `std::mismatch()` , pero primero debemos verificar que la primera cadena sea tan grande como la segunda:

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

C ++ 17

Con `std::string_view` , podemos escribir la comparación directa que queremos sin tener que preocuparnos por la sobrecarga de asignación o hacer copias:

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

Convertir a std :: string

`std::ostringstream` se puede utilizar para convertir cualquier tipo transmitible en una representación de cadena, insertando el objeto en un objeto `std::ostringstream` (con el operador de inserción de flujo `<<`) y luego convirtiendo el `std::ostringstream` en un `std::string` .

Por ejemplo, `int` :

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Escribiendo tu propia función de conversión, la simple:

```
template<class T>
std::string toString(const T& x)
{
```

```
  std::ostringstream ss;
  ss << x;
  return ss.str();
}
```

funciona pero no es adecuado para el código crítico de rendimiento.

Las clases definidas por el usuario pueden implementar el operador de inserción de flujo si lo desea:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

C ++ 11

Aparte de las secuencias, desde C ++ 11 también puede usar la función `std::to_string` (y `std::to_wstring`) que está sobrecargada para todos los tipos fundamentales y devuelve la representación de cadena de su parámetro.

```
std::string s = to_string(0x12f3);  // after this the string s contains "4851"
```

Lea std :: string en línea: https://riptutorial.com/es/cplusplus/topic/488/std----string

Capítulo 133: std :: variante

Observaciones

La variante es un reemplazo para el uso de la `union` cruda. Es de tipo seguro y sabe de qué tipo es, y construye y destruye cuidadosamente los objetos dentro de él cuando debería.

Casi nunca está vacío: solo en los casos de esquina donde se reemplaza el contenido y no se puede retroceder de manera segura, se queda en un estado vacío.

Se comporta algo así como un `std::tuple` , y algo así como un `std::optional` .

Usar `std::get` y `std::get_if` suele ser una mala idea. La respuesta correcta suele ser `std::visit` , que le permite lidiar con todas las posibilidades allí mismo. `if constexpr` se puede usar `if constexpr` dentro de la `visit` si necesita ramificar su comportamiento, en lugar de hacer una secuencia de verificaciones de tiempo de ejecución que dupliquen lo que la `visit` hará de manera más eficiente.

Examples

Basic std :: uso variante

Esto crea una variante (una unión etiquetada) que puede almacenar un `int` o una `string` .

```
std::variant< int, std::string > var;
```

Podemos almacenar uno de los dos tipos en él:

```
var = "hello"s;
```

Y podemos acceder a los contenidos a través de `std::visit` :

```
// Prints "hello\n":
visit( [](auto&& e) {
  std::cout << e << '\n';
}, var );
```

Pasando en un objeto lambda polimórfico o función similar.

Si estamos seguros de saber qué tipo es, podemos obtenerlo:

```
auto str = std::get<std::string>(var);
```

Pero esto tirará si nos equivocamos. `get_if` :

```
auto* str  = std::get_if<std::string>(&var);
```

devuelve `nullptr` si `nullptr` mal.

Las variantes no garantizan una asignación de memoria dinámica (aparte de la que se asigna por sus tipos contenidos). Solo uno de los tipos en una variante se almacena allí, y en casos raros (que involucran excepciones al asignar y no hay una forma segura de retroceder), la variante puede quedar vacía.

Las variantes le permiten almacenar múltiples tipos de valores en una variable de manera segura y eficiente. Básicamente son `union` inteligentes y seguros.

Crear punteros pseudo-método

Este es un ejemplo avanzado.

Puede utilizar la variante para borrar el tipo de peso ligero.

```
template<class F>
struct pseudo_method {
  F f;
  // enable C++17 class type deduction:
  pseudo_method( F&& fin ):f(std::move(fin)) {}

  // Koenig lookup operator->*, as this is a pseudo-method it is appropriate:
  template<class Variant> // maybe add SFINAE test that LHS is actually a variant.
  friend decltype(auto) operator->*( Variant&& var, pseudo_method const& method ) {
    // var->*method returns a lambda that perfect forwards a function call,
    // behaving like a method pointer basically:
    return [&](auto&&...args)->decltype(auto) {
      // use visit to get the type of the variant:
      return std::visit(
        [&](auto&& self)->decltype(auto) {
          // decltype(x)(x) is perfect forwarding in a lambda:
          return method.f( decltype(self)(self), decltype(args)(args)... );
        },
        std::forward<Var>(var)
      );
    };
  }
};
```

esto crea un tipo que sobrecarga al `operator->*` con una `Variant` en el lado izquierdo.

```
// C++17 class type deduction to find template argument of `print` here.
// a pseudo-method lambda should take `self` as its first argument, then
// the rest of the arguments afterwards, and invoke the action:
pseudo_method print = [](auto&& self, auto&&...args)->decltype(auto) {
  return decltype(self)(self).print( decltype(args)(args)... );
};
```

Ahora si tenemos 2 tipos cada uno con un método de `print` :

```
struct A {
  void print( std::ostream& os ) const {
    os << "A";
```

```
  }
};
struct B {
  void print( std::ostream& os ) const {
    os << "B";
  }
};
```

Tenga en cuenta que son tipos no relacionados. Podemos:

```
std::variant<A,B> var = A{};

(var->*print)(std::cout);
```

y enviará la llamada directamente a `A::print(std::cout)` para nosotros. Si en su lugar inicializamos la `var` con `B{}` , se enviaría a `B::print(std::cout)` .

Si creamos un nuevo tipo C:

```
struct C {};
```

entonces:

```
std::variant<A,B,C> var = A{};
(var->*print)(std::cout);
```

no se compilará porque no hay ningún `C.print(std::cout)` .

La extensión de lo anterior permitiría que se detecten y utilicen las `print` función libre, posiblemente con el uso de `if constexpr` dentro del pseudo-método de `print` .

Ejemplo vivo actualmente usando `boost::variant` en lugar de `std::variant` .

Construyendo un `std :: variant`

Esto no cubre los asignadores.

```
struct A {};
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // contains a A()
std::variant<A,B> var_ab1 = 7; // contains a B(7)
std::variant<A,B> var_ab2 = var_ab1; // contains a B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // contains a C(7)
std::variant<C> var_c0; // illegal, no default ctor for C
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // contains D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // contains A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // contains D{1,3,3,4}
```

Lea std :: variante en línea: https://riptutorial.com/es/cplusplus/topic/5239/std----variante

Capítulo 134: std :: vector

Introducción

Un vector es una matriz dinámica con almacenamiento manejado automáticamente. Se puede acceder a los elementos de un vector con la misma eficacia que los de una matriz, con la ventaja de que los vectores pueden cambiar dinámicamente de tamaño.

En términos de almacenamiento, los datos vectoriales (por lo general) se colocan en la memoria asignada dinámicamente, por lo que requieren una pequeña sobrecarga; por el contrario, `C-arrays` y `std::array` utilizan el almacenamiento automático en relación con la ubicación declarada y, por lo tanto, no tienen gastos generales.

Observaciones

El uso de un `std::vector` requiere la inclusión del encabezado `<vector>` usando `#include <vector>` .

Los elementos en un `std::vector` se almacenan de forma contigua en la tienda libre. Cabe señalar que cuando los vectores se anidan como en `std::vector<std::vector<int> >` , los elementos de cada vector son contiguos, pero cada vector asigna su propio búfer subyacente en el almacén libre.

Examples

Inicializando un std :: vector

Un `std::vector` puede inicializarse de varias maneras mientras lo declara:

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };  // v becomes {1, 2, 3}

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 };     // v becomes {3, 6}
```

```
// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6);  // v becomes {6, 6, 6}

std::vector<int> v(4);     // v becomes {0, 0, 0, 0}
```

Un vector se puede inicializar desde otro contenedor de varias maneras:

Copiar construcción (solo de otro vector), que copia datos de `v2` :

```
std::vector<int> v(v2);
std::vector<int> v = v2;
```

C ++ 11

Mueve la construcción (solo desde otro vector), que mueve los datos de `v2` :

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

Construcción de copia de iterador (rango), que copia elementos en `v` :

```
// from another vector
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}

// from an array
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3);                   // v becomes {1, 2, 3}

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

C ++ 11

Iterador move-construction, utilizando `std::make_move_iterator` , que mueve elementos a `v` :

```
// from another vector
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end());

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

Con la ayuda de la función miembro `assign()` , un `std::vector` se puede reinicializar después de su construcción:

```
v.assign(4, 100);                       // v becomes {100, 100, 100, 100}

v.assign(v2.begin(), v2.begin() + 3);   // v becomes {v2[0], v2[1], v2[2]}

int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4);                 // v becomes {2, 3, 4}
```

Insertando Elementos

Anexando un elemento al final de un vector (copiando / moviendo):

```
struct Point {
  double x, y;
  Point(double x, double y) : x(x), y(y) {}
};
std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p);  // p is copied into the vector.
```

C ++ 11

Anexando un elemento al final de un vector construyendo el elemento en su lugar:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Tenga en cuenta que `std::vector` *no* tiene una función miembro `push_front()` debido a razones de rendimiento. Al agregar un elemento al principio, se mueven todos los elementos existentes en el vector. Si desea insertar elementos con frecuencia al principio de su contenedor, puede usar `std::list` o `std::deque` .

Insertando un elemento en cualquier posición de un vector:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9);          // v now contains {9, 1, 2, 3}
```

C ++ 11

Insertando un elemento en cualquier posición de un vector construyendo el elemento en su lugar:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9);     // v now contains {1, 9, 2, 3}
```

Insertando otro vector en cualquier posición del vector:

```
std::vector<int> v(4);      // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin()+2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Insertando una matriz en cualquier posición de un vector:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a [] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin()+1, a, a+sizeof(a)/sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Use `reserve()` antes de insertar múltiples elementos si el tamaño del vector resultante se conoce de antemano para evitar múltiples reasignaciones (consulte el tamaño y la capacidad del vector):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

Asegúrese de no cometer el error de llamar a `resize()` en este caso, o de lo contrario, creará un vector con 200 elementos en los que solo los últimos cien tendrán el valor deseado.

Iterando Sobre std :: vector

Puedes iterar sobre un `std::vector` de varias maneras. Para cada una de las siguientes secciones, `v` se define como sigue:

```
std::vector<int> v;
```

Iterando en la dirección hacia adelante

C ++ 11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}

// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}

std::for_each(std::begin(v), std::end(v), fun);

// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});
```

C ++ 11

```
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}
```

Iterando en la dirección inversa

C ++ 14

```
// There is no standard way to use range based for for this.
// See below for alternatives.
```

```
// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Aunque no hay una forma integrada de usar el rango para revertir iteración; Es relativamente sencillo arreglar esto. El rango basado en los usos `begin()` y `end()` para obtener iteradores y, por lo tanto, simular esto con un objeto de envoltura puede lograr los resultados que requerimos.

C ++ 14

```
template<class C>
struct ReverseRange {
  C c; // could be a reference or a copy, if the original was a temporary
  ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
  ReverseRange(ReverseRange&&)=default;
  ReverseRange& operator=(ReverseRange&&)=delete;
  auto begin() const {return std::rbegin(c);}
  auto end()   const {return std::rend(c);}
};
// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Hacer cumplir elementos const

Desde C ++ 11, los `cbegin()` y `cend()` permiten obtener un *iterador constante* para un vector, incluso si el vector no es constante. Un iterador constante le permite leer pero no modificar los contenidos del vector, lo que es útil para imponer la corrección constante:

C ++ 11

```
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
   // type of pos is vector<T>::const_iterator
```

```
   // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operand()(T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operand()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
```

C ++ 17

as_const extiende esto a la iteración de rango:

```
for (auto const& e : std::as_const(v)) {
  std::cout << e << '\n';
}
```

Esto es fácil de implementar en versiones anteriores de C ++:

C ++ 14

```
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
  return t;
}
```

Una nota sobre la eficiencia

Dado que la clase `std::vector` es básicamente una clase que administra una matriz contigua asignada dinámicamente, el mismo principio explicado aquí se aplica a los vectores C ++. El acceso al contenido del vector por índice es mucho más eficiente cuando se sigue el principio de orden de fila mayor. Por supuesto, cada acceso al vector también pone su contenido de administración en el caché, pero como se ha debatido muchas veces (especialmente aquí y aquí), la diferencia en el rendimiento para iterar sobre un `std::vector` comparación con una matriz en bruto es despreciable. Por lo tanto, el mismo principio de eficiencia para matrices en bruto en C también se aplica para `std::vector` de C ++.

Elementos de acceso

Hay dos formas principales de acceder a los elementos en un `std::vector`

- acceso basado en índices
- iteradores

Acceso basado en índices:

Esto se puede hacer con el operador de subíndice `[]` o la función miembro `at()` .

Ambos devuelven una referencia al elemento en la posición respectiva en `std::vector` (a menos que sea un `vector<bool>`), para que pueda leerse y modificarse (si el vector no es `const`).

`[]` y `at()` difieren en que `[]` no se garantiza que realice ninguna comprobación de límites, mientras que `at()` hace. El acceso a los elementos donde `index < 0` o `index >= size` es un comportamiento indefinido para `[]` , mientras que `at()` lanza una excepción `std::out_of_range` .

Nota: Los ejemplos a continuación utilizan la inicialización del estilo C ++ 11 para mayor claridad, pero los operadores se pueden usar con todas las versiones (a menos que estén marcados con C ++ 11).

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
```

```
// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Debido a que el método `at()` realiza la verificación de límites y puede lanzar excepciones, es más lento que `[]` . Esto hace que `[]` código preferido donde la semántica de la operación garantice que el índice está dentro de los límites. En cualquier caso, los accesos a elementos de vectores se realizan en tiempo constante. Eso significa que acceder al primer elemento del vector tiene el mismo costo (en el tiempo) de acceder al segundo elemento, al tercer elemento y así sucesivamente.

Por ejemplo, considere este bucle

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

Aquí sabemos que la variable de índice `i` siempre está dentro de los límites, por lo que sería una pérdida de ciclos de CPU comprobar que `i` está dentro de los límites para cada llamada al `operator[]` .

Las funciones de miembro `front()` y `back()` permiten un acceso de referencia fácil al primer y último elemento del vector, respectivamente. Estas posiciones se utilizan con frecuencia y los accesores especiales pueden ser más legibles que sus alternativas utilizando `[]` :

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose

int a = v.front();   // a is 4, v.front() is equivalent to v[0]
v.front() = 3;       // v now contains {3, 5, 6}
int b = v.back();    // b is 6, v.back() is equivalent to v[v.size() - 1]
v.back() = 7;        // v now contains {3, 5, 7}
```

Nota : es un comportamiento indefinido invocar `front()` o `back()` en un vector vacío. Debe verificar que el contenedor no esté vacío utilizando la función miembro `empty()` (que verifica si el contenedor está vacío) antes de llamar a `front()` o `back()` . A continuación se muestra un ejemplo simple del uso de 'empty ()' para probar un vector vacío:

```
int main ()
{
  std::vector<int> v;
  int sum (0);

  for (int i=1;i<=10;i++) v.push_back(i);//create and initialize the vector

  while (!v.empty())//loop through until the vector tests to be empty
  {
     sum += v.back();//keep a running total
     v.pop_back();//pop out the element which removes it from the vector
  }

  std::cout << "total: " << sum << '\n';//output the total to the user

  return 0;
}
```

El ejemplo anterior crea un vector con una secuencia de números del 1 al 10. Luego saca los elementos del vector hasta que el vector está vacío (usando 'vacío ()') para evitar un comportamiento indefinido. Luego, la suma de los números en el vector se calcula y se muestra al usuario.

C ++ 11

El método `data()` devuelve un puntero a la memoria sin formato utilizada por `std::vector` para almacenar internamente sus elementos. Esto se usa con más frecuencia cuando se pasan los datos vectoriales a un código heredado que espera una matriz de estilo C.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
int* p = v.data(); // p points to 1
*p = 4;            // v now contains {4, 2, 3, 4}
++p;               // p points to 2
*p = 3;            // v now contains {4, 3, 3, 4}
p[1] = 2;          // v now contains {4, 3, 2, 4}
*(p + 2) = 1;      // v now contains {4, 3, 2, 1}
```

C ++ 11

Antes de C ++ 11, el método `data()` se puede simular llamando a `front()` y tomando la dirección del valor devuelto:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

Esto funciona porque los vectores siempre tienen la garantía de almacenar sus elementos en ubicaciones de memoria contiguas, asumiendo que el contenido del vector no anula al `operator&` unario `operator&` . Si lo hace, tendrás que volver a implementar `std::addressof` en pre-C ++ 11. También supone que el vector no está vacío.

Iteradores

Los iteradores se explican más detalladamente en el ejemplo "Iterando sobre `std::vector` " y el artículo Iteradores . En resumen, actúan de manera similar a los punteros a los elementos del vector:

C ++ 11

```
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;         // i is 4
++it;
i = *it;             // i is 5
*it = 6;             // v contains { 4, 6, 6 }
auto e = v.end();    // e points to the element after the end of v. It can be
                     // used to check whether an iterator reached the end of the vector:
++it;
it == v.end();       // false, it points to the element at position 2 (with value 6)
++it;
it == v.end();       // true
```

Es consistente con el estándar que los iteradores de `std::vector<T>` realidad *son* `T*` s, pero la mayoría de las bibliotecas estándar no lo hacen. No hacer esto mejora los mensajes de error, atrapa el código no portátil y se puede usar para instrumentar los iteradores con las comprobaciones de depuración en las compilaciones no liberadas. Luego, en las compilaciones de lanzamiento, la clase que rodea el puntero subyacente se optimiza.

Puede persistir una referencia o un puntero a un elemento de un vector para el acceso indirecto. Estas referencias o punteros a elementos en el `vector` permanecen estables y el acceso permanece definido a menos que agregue / elimine elementos en o antes del elemento en el `vector` , o haga que cambie la capacidad del `vector` . Esta es la misma que la regla para invalidar iteradores.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;      // p points to 2
v.insert(v.begin(), 0);     // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;           // p points to 1
v.reserve(10);              // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;           // p points to 1
```

```
v.erase(v.begin());        // p is now invalid, accessing *p is a undefined behavior.
```

Usando std :: vector como una matriz C

Hay varias formas de usar un `std::vector` como una matriz de C (por ejemplo, para la compatibilidad con las bibliotecas de C). Esto es posible porque los elementos en un vector se almacenan de forma contigua.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

A diferencia de las soluciones basadas en estándares de C ++ anteriores (ver más abajo), la función miembro `.data()` también puede aplicarse a vectores vacíos, porque en este caso no causa un comportamiento indefinido.

Antes de C ++ 11, tomaría la dirección del primer elemento del vector para obtener un puntero equivalente, si el vector no está vacío, estos dos métodos son intercambiables:

```
int* p = &v[0];      // combine subscript operator and 0 literal

int* p = &v.front(); // explicitly reference the first element
```

Nota: Si el vector está vacío, `v[0]` y `v.front()` no están definidos y no se pueden usar.

Al almacenar la dirección base de los datos del vector, tenga en cuenta que muchas operaciones (como `push_back` , `resize` , etc.) pueden cambiar la ubicación de la memoria de datos del vector, lo que invalida los punteros de datos anteriores . Por ejemplo:

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // internal memory location changed; value of p is now invalid
```

Iterador / Invalidación de puntero

Los iteradores y los punteros que apuntan a un `std::vector` pueden volverse inválidos, pero solo cuando se realizan ciertas operaciones. El uso de iteradores / punteros no válidos resultará en un comportamiento indefinido.

Las operaciones que invalidan los iteradores / punteros incluyen:

- Cualquier operación de inserción que cambie la `capacity` del `vector` invalidará *todos los* iteradores / punteros:

  ```
  vector<int> v(5); // Vector has a size of 5; capacity is unknown.
  int *p1 = &v[0];
  v.push_back(2);   // p1 may have been invalidated, since the capacity was unknown.
  ```

```
v.reserve(20);    // Capacity is now at least 20.
int *p2 = &v[0];
v.push_back(4);   // p2 is *not* invalidated, since the size of `v` is now 7.
v.insert(v.end(), 30, 9); // Inserts 30 elements at the end. The size exceeds the
                          // requested capacity of 20, so `p2` is (probably) invalidated.
int *p3 = &v[0];
v.reserve(v.capacity() + 20); // Capacity exceeded, thus `p3` is invalid.
```

C ++ 11

```
auto old_cap = v.capacity();
v.shrink_to_fit();
if(old_cap != v.capacity())
    // Iterators were invalidated.
```

- Cualquier operación de inserción, que no aumente la capacidad, todavía invalidará los iteradores / punteros que apuntan a los elementos en la posición de inserción y los pasan. Esto incluye el iterador `end` :

```
vector<int> v(5);
v.reserve(20);                 // Capacity is at least 20.
int *p1 = &v[0];
int *p2 = &v[3];
v.insert(v.begin() + 2, 5, 0); // `p2` is invalidated, but since the capacity
                               // did not change, `p1` remains valid.
int *p3 = &v[v.size() - 1];
v.push_back(10); // The capacity did not change, so `p3` and `p1` remain valid.
```

- Cualquier operación de eliminación invalidará los iteradores / punteros que apuntan a los elementos eliminados y a cualquier elemento más allá de los elementos eliminados. Esto incluye el iterador `end` :

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` is invalid, but `p1` remains valid.
```

- `operator=` (copiar, mover u otro) y `clear()` invalidarán todos los iteradores / punteros que apuntan al vector.

Borrando elementos

Eliminando el último elemento:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back();                          // v becomes {1, 2}
```

Eliminando todos los elementos:

```
std::vector<int> v{ 1, 2, 3 };
v.clear();                              // v becomes an empty vector
```

Eliminando elemento por índice:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3);                 // v becomes {1, 2, 3, 5, 6}
```

Nota: Para un `vector` elimina un elemento que no es el último elemento, todos los elementos más allá del elemento eliminado deben copiarse o moverse para llenar el espacio, consulte la nota a continuación y std :: list .

Borrar todos los elementos en un rango:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5);  // v becomes {1, 6}
```

Nota: Los métodos anteriores no cambian la capacidad del vector, solo el tamaño. Ver Tamaño y Capacidad del Vector .

El método de `erase` , que elimina una gama de elementos, se utiliza a menudo como parte del lenguaje de **borrado-eliminar** . Es decir, primero `std::remove` mueve algunos elementos al final del vector, y luego `erase` cortes. Esta es una operación relativamente ineficiente para cualquier índice menor que el último índice del vector porque todos los elementos después de los segmentos borrados deben reubicarse en nuevas posiciones. Para aplicaciones críticas de velocidad que requieren la eliminación eficiente de elementos arbitrarios en un contenedor, vea std :: list .

Eliminando elementos por valor:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v becomes {1, 1, 3, 3}
```

Eliminando elementos por condición:

```
// std::remove_if needs a function, that takes a vector element as argument and returns true,
// if the element shall be removed
bool _predicate(const int& element) {
    return (element > 3); // This will cause all elements to be deleted that are larger than 3
}
...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v becomes {1, 2, 3}
```

Eliminar elementos por lambda, sin crear una función de predicado adicional

C ++ 11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
     [](auto& element){return element > 3;} ), v.end()
);
```

Borrar elementos por condición de un bucle:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // after erasing, 'it' will be set to the next element in v
    else
        ++it;             // manually set 'it' to the next element in v
}
```

Si bien es importante que *no* se incrementará `it` en caso de una eliminación, se debe considerar el uso de un método diferente cuando el entonces borrar repetidamente en un bucle. Considere `remove_if` para una manera más eficiente.

Eliminar elementos por condición de un bucle inverso:

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
rev_itr it = v.rbegin();

while (it != v.rend()) { // after the loop only '0' will be in v
    int value = *it;
    if (value) {
        ++it;
        // See explanation below for the following line.
        it = rev_itr(v.erase(it.base()));
    } else
        ++it;
}
```

Note algunos puntos para el bucle anterior:

- Dado un iterador inverso `it` apunta a algún elemento, la `base` del método proporciona el iterador regular (no inverso) que apunta al mismo elemento.

- `vector::erase(iterator)` borra el elemento apuntado por un iterador y devuelve un iterador al elemento que siguió al elemento dado.
- `reverse_iterator::reverse_iterator(iterator)` construye un iterador inverso a partir de un iterador.

Poner en total, la línea `it = rev_itr(v.erase(it.base()))` dice: tomar el iterador inverso `it` , han `v` borrar el elemento señalado por su iterador regular; tomar el iterador resultante, construir un iterador inverso de ella, y asignarla al iterador inverso `it` .

Eliminar todos los elementos usando `v.clear()` no libera memoria (la `capacity()` del vector permanece sin cambios). Para reclamar espacio, usa:

```
std::vector<int>().swap(v);
```

C ++ 11

`shrink_to_fit()` libera la capacidad vectorial no utilizada:

```
v.shrink_to_fit();
```

`shrink_to_fit` no garantiza reclamar realmente el espacio, pero la mayoría de las implementaciones actuales lo hacen.

Encontrando un Elemento en std :: vector

La función `std::find` , definida en el encabezado `<algorithm>` , se puede usar para encontrar un elemento en un `std::vector` .

`std::find` usa el `operator==` para comparar elementos para la igualdad. Devuelve un iterador al primer elemento en el rango que se compara igual al valor.

Si no se encuentra el elemento en cuestión, `std::find` devuelve `std::vector::end` (o `std::vector::cend` si el vector es `const`).

C ++ 11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

C ++ 11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Si necesita realizar muchas búsquedas en un vector grande, puede considerar ordenar primero el vector, antes de usar el algoritmo de `binary_search` .

Para encontrar el primer elemento en un vector que satisface una condición, se puede usar `std::find_if` . Además de los dos parámetros dados a `std::find` , `std::find_if` acepta un tercer argumento que es un objeto de función o puntero de función a una función de predicado. El predicado debe aceptar un elemento del contenedor como argumento y devolver un valor convertible a `bool` , sin modificar el contenedor:

C ++ 11

```
bool isEven(int val) {
    return (val % 2 == 0);
}

struct moreThan {
    moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};

static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element

std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

C ++ 11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [](int val){return val % 2 == 0;});
// `it` points to 8, the first even element

auto missing = std::find_if(v.begin(), v.end(), [](int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Convertir una matriz a std :: vector

Una matriz se puede convertir fácilmente en un `std::vector` usando `std::begin` y `std::end` :

C ++ 11

```
int values[5] = { 1, 2, 3, 4, 5 }; // source array

std::vector<int> v(std::begin(values), std::end(values)); // copy array to new vector

for(auto &x: v)
    std::cout << x << " ";
std::cout << std::endl;
```

1 2 3 4 5

```
int main(int argc, char* argv[]) {
    // convert main arguments into a vector of strings.
    std::vector<std::string>  args(argv, argv + argc);
}
```

También se puede usar un inicializador de C ++ 11 <> para inicializar el vector a la vez

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

vector : La excepción a tantas, tantas reglas

El estándar (sección 23.3.7) especifica que se proporciona una especialización del `vector<bool>` , que optimiza el espacio al empaquetar los valores `bool` , de modo que cada uno ocupa solo un bit. Dado que los bits no son direccionables en C ++, esto significa que varios requisitos en el `vector` no se colocan en el `vector<bool>` :

- No se requiere que los datos almacenados sean contiguos, por lo que no se puede pasar un `vector<bool>` a una API de C que espera una matriz `bool` .
- `at()` , `operator []` y la anulación de referencias de los iteradores no devuelven una referencia a `bool` . Más bien, devuelven un objeto proxy que (de manera imperfecta) simula una referencia a un `bool` al sobrecargar sus operadores de asignación. Como ejemplo, es posible que el siguiente código no sea válido para `std::vector<bool>` , porque al eliminar la referencia a un iterador no se devuelve una referencia:

C ++ 11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

De manera similar, las funciones que esperan un `bool&` argumento no se pueden usar con el resultado del `operator []` o `at()` aplicado al `vector<bool>` , o con el resultado de la desreferenciación de su iterador:

```
void f(bool& b);
f(v[0]);             // error
f(*v.begin());       // error
```

La implementación de `std::vector<bool>` depende tanto del compilador como de la arquitectura. La especialización se implementa empaquetando `n` booleanos en la sección más baja de la memoria. Aquí, `n` es el tamaño en bits de la memoria direccionable más baja. En la mayoría de los sistemas modernos esto es de 1 byte u 8 bits. Esto significa que un byte puede almacenar 8 valores booleanos. Esta es una mejora sobre la implementación tradicional donde 1 valor booleano se almacena en 1 byte de memoria.

Nota: el siguiente ejemplo muestra los posibles valores bitwise de bytes individuales en un `vector<bool>` tradicional vs. optimizado. Esto no siempre será cierto en todas las arquitecturas. Sin embargo, es una buena manera de visualizar la optimización. En los ejemplos siguientes, un byte se representa como [x, x, x, x, x, x, x, x].

Tradicional *`std::vector<char>` almacena 8 valores booleanos:*

C ++ 11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Representación bitwise:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

Specialized *`std::vector<bool>` almacena 8 valores booleanos:*

C ++ 11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Representación bitwise:

```
[1,0,0,0,1,0,1,1]
```

Observe en el ejemplo anterior, que en la versión tradicional de `std::vector<bool>` , 8 los valores booleanos ocupan 8 bytes de memoria, mientras que en la versión optimizada de `std::vector<bool>` , solo usan 1 byte de memoria. Esta es una mejora significativa en el uso de la memoria. Si necesita pasar un `vector<bool>` a una API de estilo C, es posible que deba copiar los valores a una matriz o encontrar una mejor manera de usar la API, si la memoria y el rendimiento están en riesgo.

Tamaño y capacidad del vector

El tamaño del vector es simplemente el número de elementos en el vector:

1. El **tamaño del** vector actual es consultado por la función miembro `size()` . La función

`empty()` conveniencia devuelve `true` si el tamaño es 0:

```
vector<int> v = { 1, 2, 3 }; // size is 3
const vector<int>::size_type size = v.size();
cout << size << endl; // prints 3
cout << boolalpha << v.empty() << endl; // prints false
```

2. El vector construido predeterminado comienza con un tamaño de 0:

```
vector<int> v; // size is 0
cout << v.size() << endl; // prints 0
```

3. La adición de `N` elementos al vector aumenta el **tamaño** en `N` (por ejemplo, mediante las `push_back()` , `insert()` o `resize()`).

4. La eliminación de `N` elementos del vector disminuye el **tamaño** en `N` (por ejemplo, mediante las `pop_back()` , `erase()` o `clear()`).

5. Vector tiene un límite superior específico de la implementación en su tamaño, pero es probable que se quede sin RAM antes de alcanzarla:

```
vector<int> v;
const vector<int>::size_type max_size = v.max_size();
cout << max_size << endl; // prints some large number
v.resize( max_size ); // probably won't work
v.push_back( 1 ); // definitely won't work
```

Error común: el **tamaño** no es necesariamente (o incluso generalmente) `int` :

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

La capacidad del vector difiere del **tamaño** . Mientras que el **tamaño** es simplemente cuántos elementos tiene el vector actualmente, la **capacidad** es para cuántos elementos asignó / reservó la memoria. Esto es útil porque las (re) asignaciones demasiado frecuentes de tamaños demasiado grandes pueden ser costosas.

1. La **capacidad** vectorial actual es consultada por la función miembro de `capacity()` . **La capacidad** es siempre mayor o igual al **tamaño** :

```
vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
const vector<int>::size_type capacity = v.capacity();
cout << capacity << endl; // prints number >= 3
```

2. Puede reservar la capacidad manualmente mediante la función de `reserve( N )` (cambia la capacidad del vector a `N`):

```
// !!!bad!!!evil!!!
vector<int> v_bad;
for( int i = 0; i < 10000; ++i ) {
    v_bad.push_back( i ); // possibly lot of reallocations
}

// good
vector<int> v_good;
v_good.reserve( 10000 ); // good! only one allocation
for( int i = 0; i < 10000; ++i ) {
    v_good.push_back( i ); // no allocations needed anymore
}
```

3. Puede solicitar la liberación del exceso de capacidad mediante `shrink_to_fit()` (pero la implementación no tiene que obedecerle). Esto es útil para conservar la memoria usada:

```
vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but
possibly false)
```

Vector, en parte, administra la capacidad automáticamente, cuando agrega elementos, puede decidir crecer. A los implementadores les gusta usar 2 o 1.5 para el factor de crecimiento (la proporción de oro sería el valor ideal, pero no es práctico debido a que es un número racional). Por otro lado, el vector generalmente no se encoge automáticamente. Por ejemplo:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!

v = { 1, 2, 3, 4 }; // size is 4, and lets assume capacity is 4.
v.push_back( 5 ); // capacity grows - let's assume it grows to 6 (1.5 factor)
v.push_back( 6 ); // no change in capacity
v.push_back( 7 ); // capacity grows - let's assume it grows to 9 (1.5 factor)
// and so on
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // capacity stays the same
```

Vectores de concatenacion

One `std::vector` se puede agregar a otro usando la función miembro `insert()` :

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

Sin embargo, esta solución falla si intenta anexarse un vector a sí mismo, porque el estándar especifica que los iteradores dados a `insert()` no deben ser del mismo rango que los elementos del objeto receptor.

c ++ 11

En lugar de usar las funciones miembro del vector, las funciones `std::begin()` y `std::end()` se

pueden usar:

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

Esta es una solución más general, por ejemplo, porque `b` también puede ser una matriz. Sin embargo, también esta solución no le permite agregar un vector a sí mismo.

Si el orden de los elementos en el vector receptor no importa, considerando la cantidad de elementos en cada vector puede evitar operaciones de copia innecesarias:

```
if (b.size() < a.size())
  a.insert(a.end(), b.begin(), b.end());
else
  b.insert(b.end(), a.begin(), a.end());
```

Reduciendo la capacidad de un vector

Un `std::vector` aumenta automáticamente su capacidad al insertarse según sea necesario, pero nunca reduce su capacidad después de la eliminación del elemento.

```
// Initialize a vector with 100 elements
std::vector<int> v(100);

// The vector's capacity is always at least as large as its size
auto const old_capacity = v.capacity();
// old_capacity >= 100

// Remove half of the elements
v.erase(v.begin() + 50, v.end());  // Reduces the size from 100 to 50 (v.size() == 50),
                                   // but not the capacity (v.capacity() == old_capacity)
```

Para reducir su capacidad, podemos copiar el contenido de un vector en un nuevo vector temporal. El nuevo vector tendrá la capacidad mínima necesaria para almacenar todos los elementos del vector original. Si la reducción de tamaño del vector original fue significativa, entonces es probable que la reducción de capacidad para el nuevo vector sea significativa. Luego podemos intercambiar el vector original con el temporal para conservar su capacidad minimizada:

```
std::vector<int>(v).swap(v);
```

C ++ 11

En C ++ 11 podemos usar la función miembro `shrink_to_fit()` para un efecto similar:

```
v.shrink_to_fit();
```

Nota: la función miembro `shrink_to_fit()` es una solicitud y no garantiza la reducción de la capacidad.

Uso de un vector ordenado para la búsqueda rápida de elementos

El encabezado `<algorithm>` proporciona una serie de funciones útiles para trabajar con vectores ordenados.

Un requisito previo importante para trabajar con vectores ordenados es que los valores almacenados sean comparables con `<` .

Un vector sin clasificar se puede ordenar usando la función `std::sort()` :

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

Los vectores `std::lower_bound()` permiten una búsqueda eficiente de elementos usando la función `std::lower_bound()` . A diferencia de `std::find()` , esto realiza una búsqueda binaria eficiente en el vector. El inconveniente es que solo proporciona resultados válidos para rangos de entrada ordenados:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Nota: Si el valor solicitado no es parte del vector, `std::lower_bound()` devolverá un iterador al primer elemento que sea *mayor* que el valor solicitado. Este comportamiento nos permite insertar un nuevo elemento en su lugar correcto en un vector ya ordenado:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

Si necesita insertar muchos elementos a la vez, podría ser más eficiente llamar a `push_back()` para todos ellos primero y luego llamar a `std::sort()` una vez que se hayan insertado todos los elementos. En este caso, el aumento del costo de la clasificación puede compensar el costo reducido de insertar nuevos elementos al final del vector y no en el medio.

Si su vector contiene múltiples elementos del mismo valor, `std::lower_bound()` intentará devolver un iterador al primer elemento del valor buscado. Sin embargo, si necesita insertar un nuevo elemento *después* del último elemento del valor buscado, debe usar la función `std::upper_bound()` ya que esto causará un menor desplazamiento de los elementos:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

Si necesita los iteradores de límite superior e inferior, puede usar la función `std::equal_range()` para recuperar ambos de manera eficiente con una llamada:

```
std::pair<std::vector<int>::iterator,
          std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

Para probar si un elemento existe en un vector ordenado (aunque no es específico de los vectores), puede usar la función `std::binary_search()` :

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Funciones que devuelven grandes vectores

C ++ 11

En C ++ 11, los compiladores deben moverse implícitamente desde una variable local que se está devolviendo. Además, la mayoría de los compiladores pueden realizar copias de la copia en muchos casos y evitar el movimiento por completo. Como resultado de esto, devolver objetos grandes que se pueden mover a bajo costo ya no requiere un manejo especial:

```
#include <vector>
#include <iostream>

// If the compiler is unable to perform named return value optimization (NRVO)
// and elide the move altogether, it is required to move from v into the return value.
std::vector<int> fillVector(int a, int b) {
    std::vector<int> v;
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
    return v; // implicit move
}

int main() { // declare and fill vector
    std::vector<int> vec = fillVector(1, 10);

    // print vector
    for (auto value : vec)
        std::cout << value << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "

    std::cout << std::endl;

    return 0;
}
```

C ++ 11

Antes de C ++ 11, la mayoría de los compiladores ya permitían e implementaban el uso de copias. Sin embargo, debido a la ausencia de semántica de movimiento, en el código heredado o el código que debe compilarse con versiones anteriores del compilador que no implementan esta optimización, puede encontrar vectores que se pasan como argumentos de salida para evitar la copia innecesaria:

```
#include <vector>
#include <iostream>

// passing a std::vector by reference
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
```

```
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
}

int main() {// declare vector
    std::vector<int> vec;

    // fill vector
    fillVectorFrom_By_Ref(1, 10, vec);
    // print vector
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}
```

Encuentre el elemento máximo y mínimo y el índice respectivo en un vector

Para encontrar el elemento más grande o más pequeño almacenado en un vector, puede usar los métodos `std::max_element` y `std::min_element` , respectivamente. Estos métodos se definen en el encabezado `<algorithm>` . Si varios elementos son equivalentes al elemento más grande (el más pequeño), los métodos devuelven el iterador al primer elemento de este tipo. Devuelve `v.end()` para vectores vacíos.

```
std::vector<int> v = {5, 2, 8, 10, 9};
int maxElementIndex = std::max_element(v.begin(),v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(),v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << '\n';
std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << '\n';
```

Salida:

> maxElementIndex: 3, maxElement: 10
> minElementIndex: 1, minElement: 2

C ++ 11

El elemento mínimo y máximo en un vector se puede recuperar al mismo tiempo usando el método `std::minmax_element` , que también se define en el encabezado `<algorithm>` :

```
std::vector<int> v = {5, 2, 8, 10, 9};
auto minmax = std::minmax_element(v.begin(), v.end());

std::cout << "minimum element: " << *minmax.first << '\n';
std::cout << "maximum element: " << *minmax.second << '\n';
```

Salida:

elemento minimo: 2
elemento máximo: 10

Matrices usando vectores

Los vectores se pueden utilizar como una matriz 2D definiéndolos como un vector de vectores.

Una matriz con 3 filas y 4 columnas con cada celda inicializada como 0 puede definirse como:

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

C ++ 11

La sintaxis para inicializarlos utilizando listas de inicializadores o de otro modo son similares a las de un vector normal.

```
  std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                           {4,5,6,7},
                                           {8,9,10,11}
                                         };
```

Se puede acceder a los valores en un vector similar a una matriz 2D

```
int var = matrix[0][2];
```

La iteración en toda la matriz es similar a la de un vector normal pero con una dimensión adicional.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
```

C ++ 11

```
for(auto& row: matrix)
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

Un vector de vectores es una forma conveniente de representar una matriz, pero no es el más eficiente: los vectores individuales están dispersos alrededor de la memoria y la estructura de datos no es compatible con el caché.

Además, en una matriz adecuada, la longitud de cada fila debe ser la misma (este no es el caso

de un vector de vectores). La flexibilidad adicional puede ser una fuente de errores.

Lea std :: vector en línea: https://riptutorial.com/es/cplusplus/topic/511/std----vector

Capítulo 135: Técnicas de refactorización

Introducción

Refactorización se refiere a la modificación del código existente en una versión mejorada. Aunque la refactorización a menudo se realiza mientras se cambia el código para agregar características o corregir errores, el término en particular se refiere a mejorar el código sin necesariamente agregar características o corregir errores.

Examples

Recorrer la refactorización

Aquí hay un programa que podría beneficiarse de la refactorización. Es un programa simple que utiliza C ++ 11, cuyo objetivo es calcular e imprimir todos los números primos del 1 al 100 y se basa en un programa que se publicó en CodeReview para su revisión.

```
#include <iostream>
#include <vector>
#include <cmath>

int main()
{
    int l = 100;
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < l; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                std::cout << no << "\n";
                break;
            }
        }
        if (isprime) {
            std::cout << no << " ";
            primes.push_back(no);
        }
    }
    std::cout << "\n";
}
```

La salida de este programa se ve así:

    3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Lo primero que notamos es que el programa no imprime `2` que es un número primo. Podríamos

simplemente agregar una línea de código para simplemente imprimir esa constante sin modificar el resto del programa, pero sería mejor *refactorizar* el programa para dividirlo en dos partes: una que crea la lista de números primos y otra que las imprime. . Así es como podría verse:

```
#include <iostream>
#include <vector>
#include <cmath>

std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}

int main()
{
    std::vector<int> primes = prime_list(100);
    for (std::size_t i = 0; i < primes.size(); ++i) {
        std::cout << primes[i] << ' ';
    }
    std::cout << '\n';
}
```

Al probar esta versión, vemos que sí funciona correctamente ahora:

    2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

El siguiente paso es observar que la segunda cláusula `if` no es realmente necesaria. La lógica en el bucle busca los factores primos de cada número dado hasta la raíz cuadrada de ese número. Esto funciona porque si hay algunos factores primos de un número, al menos uno de ellos debe ser menor o igual a la raíz cuadrada de ese número. Trabajando solo esa función (el resto del programa sigue siendo el mismo) obtenemos este resultado:

```
std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
```

```
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}
```

Podemos ir más lejos, cambiando los nombres de las variables para que sean un poco más descriptivos. Por ejemplo, `primecount` no es realmente un conteo de primos. En su lugar, es una variable de índice en el vector de números primos conocidos. Asimismo, si bien `no` se utiliza a veces como una abreviatura de "Número", en la escritura matemática, es más común el uso de `n` . También podemos hacer algunas modificaciones eliminando la `break` y declarando las variables más cerca de donde se usan.

```
std::vector<int> prime_list(int limit)
{
    std::vector<int> primes{2};
    for (int n = 3; n < limit; n += 2) {
        bool isprime = true;
        for (int i=0; isprime && primes[i] <= std::sqrt(n); ++i) {
            isprime &= (n % primes[i] != 0);
        }
        if (isprime) {
            primes.push_back(n);
        }
    }
    return primes;
}
```

También podemos refactorizar `main` para usar un "rango para" para hacerlo un poco más limpio:

```
int main()
{
    std::vector<int> primes = prime_list(100);
    for (auto p : primes) {
        std::cout << p << ' ';
    }
    std::cout << '\n';
}
```

Esta es solo una de las formas en que se puede realizar la refactorización. Otros pueden hacer elecciones diferentes. Sin embargo, el propósito de la refactorización sigue siendo el mismo, que es mejorar la legibilidad y posiblemente el rendimiento del código sin necesariamente agregar características.

Ir a la limpieza

En las bases de código C ++ que solían ser C, uno puede encontrar el patrón que se va a `goto cleanup` . Como el comando `goto` hace que el flujo de trabajo de una función sea más difícil de

entender, a menudo esto se evita. A menudo, puede ser reemplazado por declaraciones de devolución, bucles, funciones. Sin embargo, con la `goto cleanup` uno necesita deshacerse de la lógica de limpieza.

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
       goto cleanup;  //< Could become return false

    // ... Calculation which 'new's VectorStr

    result = TRUE;
cleanup:
    delete [] vec;
    return result;
}
```

En C ++ se podría usar RAII para solucionar este problema:

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
       return FALSE;  //< Could become return false

    // ... Calculation which 'new's VectorStr and stores it in vec.data

    return TRUE;
}
```

A partir de este punto, uno podría continuar refactorizando el código real. Por ejemplo, reemplazando el `VectorRAII` por `std::unique_ptr` o `std::vector` .

Lea Técnicas de refactorización en línea: https://riptutorial.com/es/cplusplus/topic/7600/tecnicas-de-refactorizacion

Capítulo 136: Tipo de borrado

Introducción

El borrado de tipo es un conjunto de técnicas para crear un tipo que puede proporcionar una interfaz uniforme a varios tipos subyacentes, mientras oculta la información de tipo subyacente del cliente. `std::function<R(A...)>` , que tiene la capacidad de contener objetos de diferentes tipos, es quizás el mejor ejemplo conocido de borrado de tipo en C ++.

Examples

Mecanismo basico

El borrado de tipo es una forma de ocultar el tipo de un objeto del código que lo usa, aunque no se derive de una clase base común. Al hacerlo, proporciona un puente entre los mundos del polimorfismo estático (plantillas; en el lugar de uso, el tipo exacto se debe conocer en el momento de la compilación, pero no es necesario declararlo para que se ajuste a una interfaz en la definición) y el polimorfismo dinámico. (herencia y funciones virtuales; en el lugar de uso, no es necesario conocer el tipo exacto en el momento de la compilación, pero debe declararse que se ajusta a una interfaz en la definición).

El siguiente código muestra el mecanismo básico de borrado de tipo.

```
#include <ostream>

class Printable
{
public:
  template <typename T>
  Printable(T value) : pValue(new Value<T>(value)) {}
  ~Printable() { delete pValue; }
  void print(std::ostream &os) const { pValue->print(os); }

private:
  Printable(Printable const &)        /* in C++1x: =delete */; // not implemented
  void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
  struct ValueBase
  {
      virtual ~ValueBase() = default;
      virtual void print(std::ostream &) const = 0;
  };
  template <typename T>
  struct Value : ValueBase
  {
      Value(T const &t) : v(t) {}
      virtual void print(std::ostream &os) const { os << v; }
      T v;
  };
  ValueBase *pValue;
};
```

En el sitio de uso, solo la definición anterior debe estar visible, al igual que con las clases base con funciones virtuales. Por ejemplo:

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

Tenga en cuenta que esto *no* es una plantilla, sino una función normal que solo se debe declarar en un archivo de encabezado y se puede definir en un archivo de implementación (a diferencia de las plantillas, cuya definición debe ser visible en el lugar de uso).

En la definición del tipo concreto, no hay que saber nada acerca de `Printable` , solo debe ajustarse a una interfaz, al igual que con las plantillas:

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
  return os << "MyType {" << mc.i << "}";
}
```

Ahora podemos pasar un objeto de esta clase a la función definida anteriormente:

```
MyType foo = { 42 };
print_value(foo);
```

Borrado a un tipo regular con vtable manual

C ++ prospera en lo que se conoce como un tipo Regular (o al menos Pseudo-Regular).

Un tipo Regular es un tipo que se puede construir y asignar y asignar a través de copiar o mover, se puede destruir y se puede comparar igual a. También se puede construir a partir de ningún argumento. Finalmente, también tiene soporte para algunas otras operaciones que son muy útiles en varios algoritmos y contenedores `std` .

Este es el documento raíz , pero en C ++ 11 querría agregar compatibilidad con `std::hash` .

Voy a utilizar el enfoque manual de vtable para el borrado de tipo aquí.

```
using dtor_unique_ptr = std::unique_ptr<void, void(*)(void*)>;
template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
  return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self);
}};
}
struct regular_vtable {
  void(*copy_assign)(void* dest, void const* src); // T&=(T const&)
  void(*move_assign)(void* dest, void* src); // T&=(T&&)
  bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
  bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
```

```
  std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
  std::type_info const&(*type)(); // typeid(T)
  dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
  return {
    [](void* dest, void const* src){ *static_cast<T*>(dest) = *static_cast<T const*>(src); },
    [](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
    [](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T
const*>(rhs); },
    [](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T
const*>(lhs),*static_cast<T const*>(rhs)); },
    [](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
    []()->decltype(auto){ return typeid(T); },
    [](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
  };
}
template<class T>
regular_vtable const* get_regular_vtable() noexcept {
  static const regular_vtable vtable=make_regular_vtable<T>();
  return &vtable;
}

struct regular_type {
  using self=regular_type;
  regular_vtable const* vtable = 0;
  dtor_unique_ptr ptr{nullptr, [](void*){}};

  bool empty() const { return !vtable; }

  template<class T, class...Args>
  void emplace( Args&&... args ) {
    ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
    if (ptr)
      vtable = get_regular_vtable<T>();
    else
      vtable = nullptr;
  }
  friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
    if (lhs.vtable != rhs.vtable) return false;
    return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
  }
  bool before(regular_type const& rhs) const {
    auto const& lhs = *this;
    if (!lhs.vtable || !rhs.vtable)
      return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
    if (lhs.vtable != rhs.vtable)
      return lhs.vtable->type().before(rhs.vtable->type());
    return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
  }
  // technically friend bool operator< that calls before is also required

  std::type_info const* type() const {
    if (!vtable) return nullptr;
    return &vtable->type();
  }
  regular_type(regular_type&& o):
    vtable(o.vtable),
    ptr(std::move(o.ptr))
```

```
  {
    o.vtable = nullptr;
  }
  friend void swap(regular_type& lhs, regular_type& rhs){
    std::swap(lhs.ptr, rhs.ptr);
    std::swap(lhs.vtable, rhs.vtable);
  }
  regular_type& operator=(regular_type&& o) {
    if (o.vtable == vtable) {
      vtable->move_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = std::move(o);
    swap(*this, tmp);
    return *this;
  }
  regular_type(regular_type const& o):
    vtable(o.vtable),
    ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr{nullptr, [](void*){}})
  {
    if (!ptr && vtable) vtable = nullptr;
  }
  regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
      vtable->copy_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
  }
  std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
  }
  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
  >
  regular_type(T&& t) {
    emplace<std::decay_t<T>>(std::forward<T>(t));
  }
};
namespace std {
  template<>
  struct hash<regular_type> {
    std::size_t operator()( regular_type const& r )const {
      return r.hash();
    }
  };
  template<>
  struct less<regular_type> {
    bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
      return lhs.before(rhs);
    }
  };
}
```

ejemplo vivo .

Dicho tipo regular se puede usar como una clave para un `std::map` o un `std::unordered_map` que

acepta *cualquier regular* para una clave, como:

```
std::map<regular_type, std::any>
```

Básicamente sería un mapa desde algo normal, a cualquier cosa copiable.

A diferencia de `any` , mi tipo de `regular_type` no hace optimización de objetos pequeños ni admite recuperar los datos originales. Recuperar el tipo original no es difícil.

La optimización de objetos pequeños requiere que almacenemos un búfer de almacenamiento alineado dentro del tipo `regular_type` , y que `regular_type` cuidadosamente el eliminador de `ptr` para que solo destruya el objeto y no lo elimine.

Comenzaría en `make_dtor_unique_ptr` y le enseñaría cómo almacenar los datos a veces en un búfer, y luego en el montón si no hay espacio en el búfer. Eso puede ser suficiente.

Una función `std :: function` solo para movimiento

`std::function` tipo de `std::function` borra hasta unas pocas operaciones. Una de las cosas que requiere es que el valor almacenado sea copiable.

Esto causa problemas en algunos contextos, como lambdas que almacenan ptrs únicos. Si está utilizando la `std::function` en un contexto en el que no importa la copia, como un grupo de subprocesos donde se envían tareas a subprocesos, este requisito puede agregar una sobrecarga.

En particular, `std::packaged_task<Sig>` es un objeto que se puede llamar y que solo se puede mover. Puede almacenar `std::packaged_task<R(Args...)>` en `std::packaged_task<void(Args...)>` , pero es una forma bastante pesada y oscura de crear un movimiento solo Clase de borrado de tipos de llamada.

De `task` la `task` . Esto demuestra cómo se puede escribir un tipo de `std::function` simple. Omití el constructor de copia (lo que implicaría agregar un método de `clone` a los `details::task_pimpl<...>` también).

```
template<class Sig>
struct task;

// putting it in a namespace allows us to specialize it nicely for void return value:
namespace details {
  template<class R, class...Args>
  struct task_pimpl {
    virtual R invoke(Args&&...args) const = 0;
    virtual ~task_pimpl() {};
    virtual const std::type_info& target_type() const = 0;
  };

  // store an F.  invoke(Args&&...) calls the f
  template<class F, class R, class...Args>
  struct task_pimpl_impl:task_pimpl<R,Args...> {
    F f;
    template<class Fin>
```

```
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual R invoke(Args&&...args) const final override {
      return f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };

  // the void version discards the return value of f:
  template<class F, class...Args>
  struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual void invoke(Args&&...args) const final override {
      f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };
};

template<class R, class...Args>
struct task<R(Args...)> {
  // semi-regular:
  task()=default;
  task(task&&)=default;
  // no copy

private:
  // aliases to make some SFINAE code below less ugly:
  template<class F>
  using call_r = std::result_of_t<F const&(Args...)>;
  template<class F>
  using is_task = std::is_same<std::decay_t<F>, task>;
public:
  // can be constructed from a callable F
  template<class F,
    // that can be invoked with Args... and converted-to-R:
    class= decltype( (R)(std::declval<call_r<F>>()) ),
    // and is not this same type:
    std::enable_if_t<!is_task<F>{}, int>* = nullptr
  >
  task(F&& f):
    m_pImpl( make_pimpl(std::forward<F>(f)) )
  {}

  // the meat: the call operator
  R operator()(Args... args)const {
        return m_pImpl->invoke( std::forward<Args>(args)... );
  }
  explicit operator bool() const {
    return (bool)m_pImpl;
  }
  void swap( task& o ) {
    std::swap( m_pImpl, o.m_pImpl );
  }
  template<class F>
  void assign( F&& f ) {
```

```
    m_pImpl = make_pimpl(std::forward<F>(f));
  }
  // Part of the std::function interface:
  const std::type_info& target_type() const {
    if (!*this) return typeid(void);
    return m_pImpl->target_type();
  }
  template< class T >
  T* target() {
    return target_impl<T>();
  }
  template< class T >
  const T* target() const {
    return target_impl<T>();
  }
  // compare with nullptr    :
  friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
  friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
  friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }
  friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
  template<class T>
  using pimpl_t = details::task_pimpl_impl<T, R, Args...>;

  template<class F>
  static auto make_pimpl( F&& f ) {
    using dF=std::decay_t<F>;
    using pImpl_t = pimpl_t<dF>;
    return std::make_unique<pImpl_t>(std::forward<F>(f));
  }
  std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;

  template< class T >
  T* target_impl() const {
    return dynamic_cast<pimpl_t<T>*>(m_pImpl.get());
  }
};
```

Para hacer que esta biblioteca valga la pena, querría agregar una pequeña optimización de búfer, para que no almacene todos los invocables en el montón.

Agregar SBO requeriría una `task(task&&)` no predeterminada `task(task&&)` , algo de `std::aligned_storage_t` dentro de la clase, un `m_pImpl unique_ptr` con un eliminador que se puede configurar para destruir solo (y no devolver la memoria al montón), y un `emplace_move_to( void* ) = 0` en el `task_pimpl` .

Ejemplo vivo del código anterior (sin SBO).

Borrado hasta un búfer contiguo de T

No todo el borrado de tipos implica herencia virtual, asignaciones, ubicación nueva o incluso punteros a funciones.

Lo que hace que se borre el tipo de borrado de tipo es que describe un (conjunto de) comportamiento (s), y toma cualquier tipo que admita ese comportamiento y lo envuelve. Toda la información que no está en ese conjunto de comportamientos es "olvidada" o "borrada".

Una `array_view` toma su rango entrante o tipo de contenedor y borra todo excepto el hecho de que es un búfer contiguo de `T`

```
// helper traits for SFINAE:
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} ||
std::is_same< data_t<Src>, std::remove_const_t<T>* >{}>;

template<class T>
struct array_view {
  // the core of the class:
  T* b=nullptr;
  T* e=nullptr;
  T* begin() const { return b; }
  T* end() const { return e; }

  // provide the expected methods of a good contiguous range:
  T* data() const { return begin(); }
  bool empty() const { return begin()==end(); }
  std::size_t size() const { return end()-begin(); }

  T& operator[](std::size_t i)const{ return begin()[i]; }
  T& front()const{ return *begin(); }
  T& back()const{ return *(end()-1); }

  // useful helpers that let you generate other ranges from this one
  // quickly and safely:
  array_view without_front( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin()+i, end()};
  }
  array_view without_back( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin(), end()-i};
  }

  // array_view is plain old data, so default copy:
  array_view(array_view const&)=default;
  // generates a null, empty range:
  array_view()=default;

  // final constructor:
  array_view(T* s, T* f):b(s),e(f) {}
  // start and length is useful in my experience:
  array_view(T* s, std::size_t length):array_view(s, s+length) {}

  // SFINAE constructor that takes any .data() supporting container
  // or other range in one fell swoop:
  template<class Src,
    std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
    std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
  >
  array_view( Src&& src ):
    array_view( src.data(), src.size() )
  {}

  // array constructor:
```

```
  template<std::size_t N>
  array_view( T(&arr)[N] ):array_view(arr, N) {}

  // initializer list, allowing {} based:
  template<class U,
    std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
  >
  array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {}
};
```

una `array_view` toma cualquier contenedor que admita `.data()` devolviendo un puntero a `T` y un método `.size()` , o una matriz, y lo borra a un rango de acceso aleatorio sobre `T` s contiguos.

Puede tomar un `std::vector<T>` , un `std::string<T>` un `std::array<T, N>` a `T[37]` , una lista de inicializadores (incluidos los basados en `{}`), o algo más Usted crea que lo admite (a través de `T* x.data()` y `size_t x.size()`).

En este caso, los datos que podemos extraer de la cosa que estamos borrando, junto con nuestro estado no propietario "vista", significa que no tenemos que asignar memoria o escribir funciones personalizadas dependientes de los tipos.

Ejemplo vivo .

Una mejora sería utilizar `data` no miembros y un `size` no miembros en un contexto habilitado para ADL.

Borrado de tipos Borrado de tipos con std :: any

Este ejemplo usa C ++ 14 y `boost::any` . En C ++ 17 puedes intercambiar en `std::any` lugar.

La sintaxis con la que terminamos es:

```
const auto print =
  make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "\n"; });

super_any<decltype(print)> a = 7;

(a->*print)(std::cout);
```

que es casi óptimo

Este ejemplo se basa en el trabajo de @dyp y @cpplearner , así como el mío.

Primero usamos una etiqueta para pasar tipos:

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

Esta clase de rasgo obtiene la firma almacenada con `any_method` :

Esto crea un tipo de puntero de función y una fábrica para dichos punteros de función, dado un

`any_method` :

```
template<class any_method>
using any_sig_from_method = typename any_method::signature;

template<class any_method, class Sig=any_sig_from_method<any_method>>
struct any_method_function;

template<class any_method, class R, class...Args>
struct any_method_function<any_method, R(Args...)>
{
  template<class T>
  using decorate = std::conditional_t< any_method::is_const, T const, T >;

  using any = decorate<boost::any>;

  using type = R(*)(any&, any_method const*, Args&&...);
  template<class T>
  type operator()( tag_t<T> )const{
    return +[](any& self, any_method const* method, Args&&...args) {
      return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );
    };
  }
};
```

`any_method_function::type` es el tipo de puntero a una función que almacenaremos junto con la instancia. `any_method_function::operator()` toma un `tag_t<T>` y escribe una instancia personalizada del `any_method_function::type` que asume que `any&` va a ser una `T`

Queremos poder escribir más de un método a la vez. Así que los agrupamos en una tupla, y escribimos un envoltorio de ayuda para pegar la tupla en el almacenamiento estático por tipo y mantener un puntero a ellos.

```
template<class...any_methods>
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;

template<class...any_methods, class T>
any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
  return std::make_tuple(
    any_method_function<any_methods>{}(tag<T>)...
  );
}

template<class...methods>
struct any_methods {
private:
  any_method_tuple<methods...> const* vtable = 0;
  template<class T>
  static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
    static const auto table = make_vtable<methods...>(tag<T>);
    return &table;
  }
public:
  any_methods() = default;
  template<class T>
  any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
  any_methods& operator=(any_methods const&)=default;
  template<class T>
```

```
  void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }

  template<class any_method>
  auto get_invoker( tag_t<any_method> ={} ) const {
    return std::get<typename any_method_function<any_method>::type>( *vtable );
  }
};
```

Podríamos especializarlo en casos en los que el vtable es pequeño (por ejemplo, 1 elemento), y usar punteros directos almacenados en clase en esos casos por eficiencia.

Ahora empezamos la `super_any` . Utilizo `super_any_t` para hacer la declaración de `super_any` un poco más fácil.

```
template<class...methods>
struct super_any_t;
```

Esto busca los métodos que Super Super admite para SFINAE y mejores mensajes de error:

```
template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
    std::integral_constant<bool, std::is_same<M0, method>{}  ||
super_method_applies_helper<super_any_t<Methods...>, method>{}>
{};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} && method::is_const >{};
}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} >{};
}

template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};
```

A continuación creamos el tipo `any_method` . Un `any_method` es un pseudo-método-puntero. Lo creamos globalmente y `const` usando sintaxis como:

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

o en C ++ 17:

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

Tenga en cuenta que usar un dispositivo no lambda puede hacer que las cosas se pongan peludas, ya que usamos el tipo para un paso de búsqueda. Esto se puede arreglar, pero haría este ejemplo más largo de lo que ya es. Por lo tanto, siempre inicialice cualquier método desde un lambda, o desde un tipo parametarizado en un lambda.

```
template<class Sig, bool const_method, class F>
struct any_method {
  using signature=Sig;
  enum{is_const=const_method};
private:
  F f;
public:

  template<class Any,
    // SFINAE testing that one of the Anys's matches this type:
    std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
  >
  friend auto operator->*( Any&& self, any_method const& m ) {
    // we don't use the value of the any_method, because each any_method has
    // a unique type (!) and we check that one of the auto*'s in the super_any
    // already has a pointer to us.  We then dispatch to the corresponding
    // any_method_data...

    return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)-
>decltype(auto)
    {
      return invoke( decltype(self)(self), &m, decltype(args)(args)... );
    };
  }
  any_method( F fin ):f(std::move(fin)) {}

  template<class...Args>
  decltype(auto) operator()(Args&&...args)const {
    return f(std::forward<Args>(args)...);
  }
};
```

Un método de fábrica, no es necesario en C ++ 17, creo:

```
template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
make_any_method( F&& f ) {
  return {std::forward<F>(f)};
}
```

Este es el aumento de `any` . Es a la vez una `any` , y lleva en torno a un conjunto de punteros a funciones de tipo de borrado que cambian cada vez que el contenido `any` lo hace:

```
template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
  using vtable=any_methods<methods...>;
public:
  template<class T,
    std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
  >
  super_any_t( T&& t ):
    boost::any( std::forward<T>(t) )
```

```
  {
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
  }

  boost::any& as_any()&{return *this;}
  boost::any&& as_any()&&{return std::move(*this);}
  boost::any const& as_any()const&{return *this;}
  super_any_t()=default;
  super_any_t(super_any_t&& o):
    boost::any( std::move( o.as_any() ) ),
    vtable(o)
  {}
  super_any_t(super_any_t const& o):
    boost::any( o.as_any() ),
    vtable(o)
  {}
  template<class S,
    std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
  >
  super_any_t( S&& o ):
    boost::any( std::forward<S>(o).as_any() ),
    vtable(o)
  {}
  super_any_t& operator=(super_any_t&&)=default;
  super_any_t& operator=(super_any_t const&)=default;

  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>* =nullptr
  >
  super_any_t& operator=( T&& t ) {
    ((boost::any&)*this) = std::forward<T>(t);
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
    return *this;
  }
};
```

Debido a que almacenamos los `any_method` s como objetos `const` , esto hace que hacer un `super_any` un poco más fácil:

```
template<class...Ts>
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

Código de prueba:

```
const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p
<< "\n"; });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os
<< p << L"\n"; });

int main()
{
  super_any<decltype(print), decltype(wprint)> a = 7;
  super_any<decltype(print), decltype(wprint)> a2 = 7;

  (a->*print)(std::cout);
  (a->*wprint)(std::wcout);
```

```
}
```

ejemplo vivo .

Originalmente publicado aquí en una auto-pregunta y respuesta de SO (y las personas mencionadas anteriormente ayudaron con la implementación).

Lea Tipo de borrado en línea: https://riptutorial.com/es/cplusplus/topic/2872/tipo-de-borrado

Capítulo 137: Tipo de Devolución Covarianza

Observaciones

La covarianza de un parámetro o un valor de retorno para una función miembro virtual `m` es donde su tipo `T` vuelve más específico en el reemplazo de `m` de una clase derivada. El tipo `T` luego varía (*varianza*) en especificidad de la misma manera (*co*) que las clases que proporcionan `m` . C ++ proporciona soporte de lenguaje para los *tipos de retorno* covariantes que son punteros en bruto o referencias en bruto, la covarianza es para el tipo de pointee o referente.

El soporte de C ++ está limitado a los tipos devueltos porque los valores de retorno de la función son los únicos **argumentos de salida** puros en C ++, y la covarianza solo es segura para un argumento de salida puro. De lo contrario, el código de llamada podría proporcionar un objeto de un tipo menos específico del que espera el código de recepción. La profesora del MIT Barbara Liskov investigó esto y relacionó los problemas de seguridad de tipo de varianza, y ahora se conoce como el Principio de Sustitución de Liskov, o **LSP** .

El soporte de covarianza esencialmente ayuda a evitar la reducción de emisiones y la comprobación dinámica de tipos.

Dado que los punteros inteligentes son del tipo de clase, no se puede usar el soporte integrado para la covarianza directamente para los resultados del puntero inteligente, pero se pueden definir funciones de envoltura de resultados del puntero inteligente no `virtual` *aparentemente covariantes* para una función `virtual` covariante que produce punteros en bruto.

Examples

1. Ejemplo de base sin devoluciones covariantes, muestra por qué son deseables

```
// 1. Base example not using language support for covariance, dynamic type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{

public:
    Top* clone() const override
    { return new D( *this ); }
};

class DD : public D
{
```

```
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}

    Top* clone() const override
    { return new DD( *this ); }
};

#include <assert.h>
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
    cout << boolalpha;

    DD* p1 = new DD();
    Top* p2 = p1->clone();
    bool const  correct_dynamic_type = (typeid( *p2 ) == typeid( DD ));
    cout << correct_dynamic_type << endl;               // "true"

    assert( correct_dynamic_type ); // This is essentially dynamic type checking. :(
    auto p2_most_derived = static_cast<DD*>( p2 );
    cout << p2_most_derived->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

2. Versión de resultado covariante del ejemplo base, comprobación de tipos estática.

```
// 2. Covariant result version of the base example, static type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
    D* /* ← Covariant return */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}
```

```
    DD* /* ← Covariant return */ clone() const override
    { return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

3. Resultado del puntero inteligente covariante (limpieza automatizada).

```
// 3. Covariant smart pointer result (automated cleanup).

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }

class Top
{
private:
    virtual Top* virtual_clone() const = 0;

public:
    unique_ptr<Top> clone() const
    { return up( virtual_clone() ); }

    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
private:
    D* /* ← Covariant return */ virtual_clone() const override
    { return new D( *this ); }

public:
    unique_ptr<D> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

class DD : public D
{
private:
    int answer_ = 42;

    DD* /* ← Covariant return */ virtual_clone() const override
    { return new DD( *this ); }
```

```
public:
    int answer() const
    { return answer_;}

    unique_ptr<DD> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

#include <iostream>
using namespace std;

int main()
{
    auto  p1 = unique_ptr<DD>(new DD());
    auto  p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    // Cleanup is automated via unique_ptr.
 }
```

Lea Tipo de Devolución Covarianza en línea: https://riptutorial.com/es/cplusplus/topic/5411/tipo-de-devolucion-covarianza

Capítulo 138: Tipo de rasgos

Observaciones

Los rasgos de tipo son construcciones con plantillas que se utilizan para comparar y probar las propiedades de diferentes tipos en el momento de la compilación. Se pueden usar para proporcionar una lógica condicional en el momento de la compilación que puede limitar o ampliar la funcionalidad de su código de una manera específica. La biblioteca de rasgos de tipo se incorporó con el estándar `c++11` , que proporciona una serie de funcionalidades diferentes. También es posible crear sus propias plantillas de comparación de rasgos de tipo.

Examples

Rasgos de tipo estándar

C ++ 11

El encabezado `type_traits` contiene un conjunto de clases de plantilla y ayudantes para transformar y verificar las propiedades de los tipos en tiempo de compilación.

Estos rasgos se usan normalmente en las plantillas para verificar errores de los usuarios, admitir la programación genérica y permitir optimizaciones.

La mayoría de los rasgos de tipo se utilizan para verificar si un tipo cumple con algunos criterios. Estos tienen la siguiente forma:

```
template <class T> struct is_foo;
```

Si la clase de plantilla está instanciada con un tipo que cumple con algunos criterios `foo` , `is_foo<T>` hereda de `std::integral_constant<bool,true>` (también conocido como `std::true_type`), de lo contrario, hereda de `std::integral_constant<bool,false>` (también conocido como `std::false_type`). Esto le da al rasgo los siguientes miembros:

Constantes

`static constexpr bool value`

`true` si `T` cumple con los criterios `foo` , `false` contrario

Funciones

`operator bool`

Devuelve `value`

C ++ 14

`bool operator()`

Devuelve `value`

Los tipos

Nombre	Definición
value_type	bool
type	std::integral_constant<bool,value>

El rasgo se puede usar en construcciones como `static_assert` o `std::enable_if` . Un ejemplo con `std::is_pointer` :

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T must be a pointer type");
}

//Overload for when T is not a pointer type
template <typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something boring
}

//Overload for when T is a pointer type
template <typename T>
typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something special
}
```

También hay varios rasgos que transforman los tipos, como `std::add_pointer` y `std::underlying_type` . Estos rasgos generalmente exponen un `type` miembro de tipo único que contiene el tipo transformado. Por ejemplo, `std::add_pointer<int>::type` is `int*` .

Escribe relaciones con std :: is_same

C ++ 11

La relación de tipo `std::is_same<T, T>` se utiliza para comparar dos tipos. Se evaluará como booleano, verdadero si los tipos son los mismos y falso en caso contrario.

p.ej

```
// Prints true on most x86 and x86_64 compilers.
std::cout << std::is_same<int, int32_t>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<float, int>::value << "\n";
// Prints false on all compilers.
std::cout  << std::is_same<unsigned int, int>::value << "\n";
```

La relación de tipo `std::is_same` también funcionará independientemente de typedefs. Esto se demuestra realmente en el primer ejemplo cuando se compara `int == int32_t` sin embargo, esto no está del todo claro.

p.ej

```
// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value <<  "\n";
```

Usar `std::is_same` para advertir cuando se usa incorrectamente una clase o función con plantilla.

Cuando se combina con una `std::is_same` estática, la plantilla `std::is_same` puede ser una herramienta valiosa para imponer el uso adecuado de las clases y funciones de plantilla.

por ejemplo, una función que solo permite la entrada desde un `int` y una opción de dos estructuras.

```
#include <type_traits>
struct foo {
  int member;
  // Other variables
};

struct bar {
  char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
  // If type T != foo || T != bar then show error message.
  static_assert(std::is_same<T, foo>::value ||
    std::is_same<T, bar>::value,
    "This function does not support the specified type.");
  return var1.member + var2;
}
```

Rasgos fundamentales de tipo

C ++ 11

Hay una serie de rasgos de tipos diferentes que comparan tipos más generales.

Es integral:

Evalúa como verdadero para todos los tipos de enteros `int` , `char` , `long` , `unsigned int` etc.

```
std::cout << std::is_integral<int>::value << "\n"; // Prints true.
std::cout << std::is_integral<char>::value << "\n"; // Prints true.
std::cout << std::is_integral<float>::value << "\n"; // Prints false.
```

Es punto flotante:

Evalúa como verdadero para todos los tipos de punto flotante. `float` , `double` , `long double` etc.

```
std::cout << std::is_floating_point<float>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<double>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<char>::value << "\n"; // Prints false.
```

Es enum:

Evalúa como verdadero para todos los tipos enumerados, incluida la `enum class` .

```
enum fruit {apple, pair, banana};
enum class vegetable {carrot, spinach, leek};
std::cout << std::is_enum<fruit>::value << "\n"; // Prints true.
std::cout << std::is_enum<vegetable>::value << "\n"; // Prints true.
std::cout << std::is_enum<int>::value << "\n"; // Prints false.
```

Es puntero:

Evalúa como verdadero para todos los punteros.

```
std::cout << std::is_pointer<int *>::value << "\n"; // Prints true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << "\n"; // Prints true.
std::cout << std::is_pointer<int>::value << "\n"; // Prints false.
```

Es la clase:

Se evalúa como verdadero para todas las clases y estructuras, con la excepción de la `enum class` .

```
struct FOO {int x, y;};
class BAR {
 public:
  int x, y;
};
enum class fruit {apple, pair, banana};
std::cout << std::is_class<FOO>::value << "\n"; // Prints true.
std::cout << std::is_class<BAR>::value << "\n"; // Prints true.
std::cout << std::is_class<fruit>::value << "\n"; // Prints false.
std::cout << std::is_class<int>::value << "\n"; // Prints false.
```

Tipo de propiedades

C ++ 11

Las propiedades de tipo comparan los modificadores que se pueden colocar sobre diferentes variables. La utilidad de estos rasgos de tipo no siempre es obvia.

Nota: El ejemplo a continuación solo ofrecería una mejora en un compilador que no optimiza. Es una simple prueba de concepto, en lugar de un ejemplo complejo.

Por ejemplo, dividir rápido por cuatro.

```
template<typename T>
inline T FastDivideByFour(cont T &var) {
  // Will give an error if the inputted type is not an unsigned integral type.
  static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
    "This function is only designed for unsigned integral types.");
  return (var >> 2);
}
```

Es constante:

Esto se evaluará como verdadero cuando el tipo es constante.

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

Es volátil:

Esto se evaluará como verdadero cuando el tipo es volátil.

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

Está firmado:

Esto se evaluará como verdadero para todos los tipos firmados.

```
std::cout << std::is_signed<int>::value << "\n"; // Prints true.
std::cout << std::is_signed<float>::value << "\n"; // Prints true.
std::cout << std::is_signed<unsigned int>::value << "\n"; // Prints false.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints false.
```

Está sin firmar:

Se evaluará como verdadero para todos los tipos sin firmar.

```
std::cout << std::is_unsigned<unsigned int>::value << "\n"; // Prints true.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints true.
std::cout << std::is_unsigned<int>::value << "\n"; // Prints false.
std::cout << std::is_signed<float>::value << "\n"; // Prints false.
```

Lea Tipo de rasgos en línea: https://riptutorial.com/es/cplusplus/topic/4750/tipo-de-rasgos

Capítulo 139: Tipo de retorno final

Sintaxis

- *function_name* ([*function_args*]) [*function_attributes*] [*function_qualifiers*] `->` *trailing-return-type* [*require_clause*]

Observaciones

La sintaxis anterior muestra una declaración de función completa que utiliza un tipo final, donde los corchetes indican una parte opcional de la declaración de función (como la lista de argumentos si es una función sin argumento).

Además, la sintaxis del tipo de retorno final prohíbe la definición de una clase, unión o tipo de enumeración dentro de un tipo de retorno final (tenga en cuenta que esto tampoco está permitido en un tipo de retorno inicial). Aparte de eso, los tipos pueden escribirse de la misma manera después de `->` como sería en cualquier otro lugar.

Examples

Evite calificar un nombre de tipo anidado

```
class ClassWithAReallyLongName {
  public:
    class Iterator { /* ... */ };
    Iterator end();
};
```

Definiendo el `end` del miembro con un tipo de retorno final:

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

Definiendo el `end` del miembro sin un tipo de retorno final:

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

El tipo de retorno final se busca en el alcance de la clase, mientras que el tipo de retorno inicial se busca en el ámbito de espacio de nombres adjunto y, por lo tanto, puede requerir una calificación "redundante".

Expresiones lambda

Un lambda *solo* puede tener un tipo de retorno final; la sintaxis de tipo de retorno inicial no es aplicable a las lambdas. Tenga en cuenta que en muchos casos no es necesario especificar un tipo de retorno para un lambda en absoluto.

```
struct Base {};
struct Derived1 : Base {};
struct Derived2 : Base {};
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

Lea Tipo de retorno final en línea: https://riptutorial.com/es/cplusplus/topic/4142/tipo-de-retorno-final

Capítulo 140: Tipos atómicos

Sintaxis

- std :: atómico <T>
- std :: atomic_flag

Observaciones

`std::atomic` permite el acceso atómico a un tipo TriviallyCopyable, es dependiente de la implementación si esto se realiza mediante operaciones atómicas o mediante el uso de bloqueos. El único tipo atómico sin bloqueo garantizado es `std::atomic_flag` .

Examples

Acceso multihilo

Se puede usar un tipo atómico para leer y escribir de forma segura en una ubicación de memoria compartida entre dos subprocesos.

Un mal ejemplo que puede causar una carrera de datos:

```
#include <thread>
#include <iostream>


//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //a primitive data type has no thread safety
    int shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //attempt to print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //this may cause undefined behavior or print a corrupted value
        //if the addingThread tries to write to 'shared' while the main thread is reading it
        std::cout << shared << std::endl;
    }
```

```
    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

El ejemplo anterior puede causar una lectura dañada y puede llevar a un comportamiento indefinido.

Un ejemplo con seguridad de hilo:

```
#include <atomic>
#include <thread>
#include <iostream>


    //function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //atomically add 'i' to result
        result->fetch_add(i);
    }
}

int main() {
    //atomic template used to store non-atomic objects
    std::atomic<int> shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //safe way to read the value of shared atomically for thread safe read
        std::cout << shared.load() << std::endl;
    }


    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

El ejemplo anterior es seguro porque todas las operaciones de `store()` y `load()` del tipo de datos `atomic` protegen el `int` encapsulado del acceso simultáneo.

Lea Tipos atómicos en línea: https://riptutorial.com/es/cplusplus/topic/3804/tipos-atomicos

Capítulo 141: Tipos sin nombre

Examples

Clases sin nombre

A diferencia de una clase o estructura con nombre, las clases y estructuras sin nombre deben crearse instancias donde están definidas, y no pueden tener constructores o destructores.

```
struct {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

class {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

Miembros anónimos

Como una extensión no estándar de C ++, los compiladores comunes permiten el uso de clases como miembros anónimos.

```
struct Example {
    struct {
        int inner_b;
    };

    int outer_b;

    //The anonymous struct's members are accessed as if members of the parent struct
    Example() : inner_b(2), outer_b(4) {
        inner_b = outer_b + 2;
    }
};

Example ex;

//The same holds true for external code referencing the struct
ex.inner_b -= ex.outer_b;
```

Como un alias de tipo

Los tipos de clase sin nombre también se pueden usar al crear alias de tipo, es decir, a través de `typedef` y `using` :

C ++ 11

```
using vec2d = struct {
    float x;
    float y;
};
```

```
typedef struct {
    float x;
    float y;
} vec2d;
```

```
vec2d pt;
pt.x = 4.f;
pt.y = 3.f;
```

Anonima union

Los nombres de los miembros de una unión anónima pertenecen al alcance de la declaración de la unión y deben ser distintos a todos los demás nombres de este alcance. El ejemplo aquí tiene la misma construcción que los miembros anónimos de ejemplo que usan "struct" pero es de conformidad estándar.

```
struct Sample {
    union {
        int a;
        int b;
    };
    int c;
};
int main()
{
  Sample sa;
  sa.a =3;
  sa.b =4;
  sa.c =5;
}
```

Lea Tipos sin nombre en línea: https://riptutorial.com/es/cplusplus/topic/2704/tipos-sin-nombre

Capítulo 142: Typedef y alias de tipo

Introducción

El `typedef` y (desde C ++ 11) `using` palabras clave se pueden usar para dar un nuevo nombre a un tipo existente.

Sintaxis

- typedef *type-specifier-seq init-declarator-list* ;
- *atributo-especificador-seq* typedef *decl-especificador-seq init-declarator-list* ; // desde C ++ 11
- utilizando el *identificador de atributo identificador -seq* (*opt*) = *id-tipo* ; // desde C ++ 11

Examples

Sintaxis básica de typedef

Una declaración `typedef` tiene la misma sintaxis que una declaración de variable o función, pero contiene la palabra `typedef` . La presencia de `typedef` hace que la declaración declare un tipo en lugar de una variable o función.

```
int T;          // T has type int
typedef int T; // T is an alias for int

int A[100];          // A has type "array of 100 ints"
typedef int A[100]; // A is an alias for the type "array of 100 ints"
```

Una vez que se ha definido un alias de tipo, se puede usar indistintamente con el nombre original del tipo.

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

`typedef` nunca crea un tipo distinto. Solo da otra forma de referirse a un tipo existente.

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

Usos más complejos de typedef.

La regla de que las declaraciones `typedef` tienen la misma sintaxis que las declaraciones de variables y funciones ordinarias se pueden usar para leer y escribir declaraciones más complejas.

```
void (*f)(int);         // f has type "pointer to function of int returning void"
typedef void (*f)(int); // f is an alias for "pointer to function of int returning void"
```

Esto es especialmente útil para construcciones con sintaxis confusa, como punteros a miembros no estáticos.

```
void (Foo::*pmf)(int);         // pmf has type "pointer to member function of Foo taking int
                               // and returning void"
typedef void (Foo::*pmf)(int); // pmf is an alias for "pointer to member function of Foo
                               // taking int and returning void"
```

Es difícil recordar la sintaxis de las siguientes declaraciones de funciones, incluso para programadores experimentados:

```
void (Foo::*Foo::f(const char*))(int);
int (&g())[100];
```

`typedef` se puede utilizar para facilitar su lectura y escritura:

```
typedef void (Foo::pmf)(int);  // pmf is a pointer to member function type
pmf Foo::f(const char*);       // f is a member function of Foo

typedef int (&ra)[100];        // ra means "reference to array of 100 ints"
ra g();                        // g returns reference to array of 100 ints
```

Declarando múltiples tipos con typedef

La palabra clave `typedef` es un especificador, por lo que se aplica por separado a cada declarador. Por lo tanto, cada nombre declarado se refiere al tipo que ese nombre tendría en ausencia de `typedef` .

```
int *x, (*p)();         // x has type int*, and p has type int(*)()
typedef int *x, (*p)(); // x is an alias for int*, while p is an alias for int(*)()
```

Declaración de alias con "utilizando"

C ++ 11

La sintaxis de `using` es muy simple: el nombre a definir va en el lado izquierdo y la definición en el lado derecho. No es necesario escanear para ver dónde está el nombre.

```
using I = int;
using A = int[100];             // array of 100 ints
using FP = void(*)(int);        // pointer to function of int returning void
using MP = void (Foo::*)(int);  // pointer to member function of Foo of int returning void
```

Crear un alias de tipo con el `using` tiene exactamente el mismo efecto que crear un alias de tipo con `typedef` . Es simplemente una sintaxis alternativa para lograr lo mismo.

A diferencia de `typedef` , el `using` puede ser templado. Una "plantilla typedef" creada con el `using` se llama una plantilla de alias .

Lea Typedef y alias de tipo en línea: https://riptutorial.com/es/cplusplus/topic/9328/typedef-y-alias-de-tipo

Capítulo 143: Uniones

Observaciones

Los sindicatos son herramientas muy útiles, pero vienen con algunas advertencias importantes:

- Es un comportamiento indefinido, según el estándar de C ++, acceder a un elemento de una unión que no fue el miembro modificado más recientemente. Aunque muchos compiladores de C ++ permiten este acceso de manera bien definida, estas son extensiones y no se pueden garantizar a través de compiladores.

  Una `std::variant` (desde C ++ 17) es como una unión, solo que le dice lo que contiene actualmente (parte de su estado visible es el tipo de valor que tiene en un momento dado: impone que el acceso al valor ocurra solo a ese tipo).

- Las implementaciones no necesariamente alinean miembros de diferentes tamaños a la misma dirección.

Examples

Características básicas de la unión

Los sindicatos son una estructura especializada dentro de la cual todos los miembros ocupan una memoria superpuesta.

```
union U {
    int a;
    short b;
    float c;
};
U u;

//Address of a and b will be equal
(void*)&u.a == (void*)&u.b;
(void*)&u.a == (void*)&u.c;

//Assigning to any union member changes the shared memory of all members
u.c = 4.f;
u.a = 5;
u.c != 4.f;
```

Uso tipico

Los sindicatos son útiles para minimizar el uso de la memoria para datos exclusivos, como cuando se implementan tipos de datos mixtos.

```
struct AnyType {
    enum {
        IS_INT,
```

```
        IS_FLOAT
    } type;

    union Data {
        int as_int;
        float as_float;
    } value;

    AnyType(int i) : type(IS_INT) { value.as_int = i; }
    AnyType(float f) : type(IS_FLOAT) { value.as_float = f; }

    int get_int() const {
        if(type == IS_INT)
            return value.as_int;
        else
            return (int)value.as_float;
    }

    float get_float() const {
        if(type == IS_FLOAT)
            return value.as_float;
        else
            return (float)value.as_int;
    }
};
```

Comportamiento indefinido

```
union U {
    int a;
    short b;
    float c;
};
U u;

u.a = 10;
if (u.b == 10) {
   // this is undefined behavior since 'a' was the last member to be
   // written to. A lot of compilers will allow this and might issue a
   // warning, but the result will be "as expected"; this is a compiler
   // extension and cannot be guaranteed across compilers (i.e. this is
   // not compliant/portable code).
}
```

Lea Uniones en línea: https://riptutorial.com/es/cplusplus/topic/2678/uniones

Capítulo 144: Usando std :: unordered_map

Introducción

std :: unordered_map es solo un contenedor asociativo. Funciona sobre las teclas y sus mapas. Clave como van los nombres, ayuda a tener singularidad en el mapa. Mientras que el valor asignado es solo un contenido que está asociado con la clave. Los tipos de datos de esta clave y mapa pueden ser cualquiera de los tipos de datos predefinidos o definidos por el usuario.

Observaciones

Como su nombre indica, los elementos del mapa no ordenado no se almacenan en una secuencia ordenada. Se almacenan de acuerdo con sus valores hash y, por lo tanto, el uso de un mapa desordenado tiene muchos beneficios, ya que solo se necesita O (1) para buscar cualquier elemento en él. También es más rápido que otros contenedores de mapas. También se puede ver en el ejemplo que es muy fácil de implementar ya que el operador ([]) nos ayuda a acceder directamente al valor asignado.

Examples

Declaración y uso

Como ya se mencionó, puede declarar un mapa desordenado de cualquier tipo. Tengamos un mapa desordenado nombrado primero con cadena y tipo entero.

```
unordered_map<string, int> first; //declaration of the map
first["One"] = 1; // [] operator used to insert the value
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

Algunas funciones básicas

```
unordered_map<data_type, data_type> variable_name; //declaration

variable_name[key_value] = mapped_value; //inserting values

variable_name.find(key_value); //returns iterator to the key value

variable_name.begin(); // iterator to the first element

variable_name.end(); // iterator to the last + 1 element
```

Lea Usando std :: unordered_map en línea:
https://riptutorial.com/es/cplusplus/topic/10540/usando-std----unordered-map

Capítulo 145: Utilizando declaración

Introducción

Una declaración de `using` introduce un nombre único en el alcance actual que se declaró anteriormente en otro lugar.

Sintaxis

- utilizando typename (*opt*) nested *-name-specifier unqualified-id* ;
- utilizando :: *id no calificado* ;

Observaciones

Una *declaración de uso* es distinta de una directiva de uso , que le dice al compilador que busque en un espacio de nombres particular cuando busca *cualquier* nombre. Una *directiva de uso* comienza con el `using namespace` .

Una *declaración de uso* también es distinta de una declaración de alias, que da un nuevo nombre a un tipo existente de la misma manera que `typedef` . Una declaración de alias contiene un signo igual.

Examples

Importando nombres individualmente desde un espacio de nombres

Una vez que se `using` uso para introducir el nombre `cout` del espacio de nombres `std` en el alcance de la función `main` , el objeto `std::cout` se puede denominar solo `cout` .

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Volver a declarar miembros de una clase base para evitar ocultar el nombre

Si se produce una *declaración de uso* en el ámbito de la clase, solo se permite volver a declarar a un miembro de una clase base. Por ejemplo, el `using std::cout` no está permitido en el alcance de la clase.

A menudo, el nombre redeclarado es uno que de otra manera estaría oculto. Por ejemplo, en el código de abajo, `d1.foo` solo hace referencia a `Derived1::foo(const char*)` y se producirá un error de compilación. La función `Base::foo(int)` está oculta y no se considera en absoluto. Sin embargo, `d2.foo(42)` está bien porque la *declaración de uso* trae `Base::foo(int)` al conjunto de

entidades llamadas `foo` en `Derived2` . La búsqueda de nombres luego encuentra ambos `foo` s y la resolución de sobrecarga selecciona `Base::foo` .

```
struct Base {
    void foo(int);
};
struct Derived1 : Base {
    void foo(const char*);
};
struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};
int main() {
    Derived1 d1;
    d1.foo(42);  // error
    Derived2 d2;
    d2.foo(42);  // OK
}
```

Heredando constructores

C ++ 11

Como un caso especial, una *declaración de uso* en el alcance de la clase puede referirse a los constructores de una clase base directa. Esos constructores luego son *heredados* por la clase derivada y pueden usarse para inicializar la clase derivada.

```
struct Base {
    Base(int x, const char* s);
};
struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
struct Derived2 : Base {
    using Base::Base;
};
int main() {
    Derived1 d1(42, "Hello, world");
    Derived2 d2(42, "Hello, world");
}
```

En el código anterior, tanto `Derived1` como `Derived2` tienen constructores que envían los argumentos directamente al constructor correspondiente de `Base` . `Derived1` realiza el reenvío explícitamente, mientras que `Derived2` , utilizando la función C ++ 11 de heredar constructores, lo hace de manera implícita.

Lea Utilizando declaración en línea: https://riptutorial.com/es/cplusplus/topic/9301/utilizando-declaracion

Capítulo 146: Valor y semántica de referencia

Examples

Copia profunda y soporte de movimiento.

Si un tipo desea tener una semántica de valor, y necesita almacenar objetos que se asignan dinámicamente, en las operaciones de copia, el tipo deberá asignar nuevas copias de esos objetos. También debe hacer esto para la copia de la asignación.

Este tipo de copia se llama "copia profunda". Toma efectivamente lo que de otro modo habría sido la semántica de referencia y lo convierte en semántica de valor:

```
struct Inner {int i;};

const int NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }
};
```

C ++ 11

La semántica de Move permite que un tipo como `Value` evite copiar realmente los datos a los que se hace referencia. Si el usuario utiliza el valor de una manera que provoca un movimiento, el objeto "copiado" se puede dejar vacío de los datos a los que hace referencia:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.
```

```
public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

De hecho, incluso podemos hacer que dicho tipo no se pueda copiar, si queremos prohibir las copias profundas y al mismo tiempo permitir que el objeto se mueva.

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) = delete;
  Value &operator=(const Value &val) = delete;

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
```

```
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

Incluso podemos aplicar la Regla de cero, mediante el uso de `unique_ptr` :

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  unique_ptr<Inner []>array_; //Move-only type.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //No need to explicitly delete. Or even declare.
  ~Value() = default; {delete[] array_;}

  //No need to explicitly delete. Or even declare.
  Value(const Value &val) = default;
  Value &operator=(const Value &val) = default;

  //Will perform an element-wise move.
  Value(Value &&val) noexcept = default;

  //Will perform an element-wise move.
  Value &operator=(Value &&val) noexcept = default;
};
```

Definiciones

Un tipo tiene valor semántico si el estado observable del objeto es funcionalmente distinto de todos los demás objetos de ese tipo. Esto significa que si copia un objeto, tiene un nuevo objeto y las modificaciones del nuevo objeto no serán visibles de ninguna manera desde el objeto anterior.

La mayoría de los tipos básicos de C ++ tienen valor semántico:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

La mayoría de los tipos definidos de biblioteca estándar también tienen semántica de valor:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

Se dice que un tipo tiene semántica de referencia si una instancia de ese tipo puede compartir su estado observable con otro objeto (externo a él), de modo que la manipulación de un objeto hará que el estado cambie dentro de otro objeto.

Los punteros de C ++ tienen semántica de valor con respecto a qué objeto apuntan, pero tienen semántica de referencia con respecto al *estado* del objeto al que apuntan:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

Las referencias de C ++ también tienen semántica de referencia.

Lea Valor y semántica de referencia en línea: https://riptutorial.com/es/cplusplus/topic/1955/valor-y-semantica-de-referencia

Capítulo 147: Variables en linea

Introducción

Una variable en línea puede definirse en varias unidades de traducción sin violar la Regla de una definición . Si se define de forma múltiple, el enlazador combinará todas las definiciones en un solo objeto en el programa final.

Examples

Definición de un miembro de datos estáticos en la definición de clase

Un miembro de datos estáticos de la clase puede estar completamente definido dentro de la definición de la clase si se declara en `inline` . Por ejemplo, la siguiente clase se puede definir en un encabezado. Antes de C ++ 17, habría sido necesario proporcionar un archivo `.cpp` para contener la definición de `Foo::num_instances` para que se definiera solo una vez, pero en C ++ 17 las definiciones múltiples de la variable en línea `Foo::num_instances` todos se refieren al mismo objeto `int` .

```
// warning: not thread-safe...
class Foo {
  public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

Como caso especial, un miembro de datos estáticos `constexpr` está implícitamente en línea.

```
class MyString {
  public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};
// in C++14, this definition was required in a single translation unit:
// constexpr int MyString::max_size;
```

Lea Variables en linea en línea: https://riptutorial.com/es/cplusplus/topic/9265/variables-en-linea

Creditos

S. No	Capítulos	Contributors
1	Empezando con C ++	Adhokshaj Mishra, ankit dassor, aquirdturtle, ArchbishopOfBanterbury, Bakhtiar Hasan, Bart van Nierop, Ben H, Bo Persson, Brandon, Brian, BullshitPingu, cb4, celtschk, Cheers and hth. - Alf, chrisb2244, Cody Gray, Community, cpatricio, Curious, Daemon, Daksh Gupta, Danh, darkpsychic, David Bippes, David G., DeepCoder, Dim_ov, dlemstra, Donald Duck, Dr t, Dylan Little, Edward, emlai, Erick Q., ethanwu10, Fantastic Mr Fox, Florian, GIRISH kuniyal, greatwolf, honk, Humam Helfawi, Hurkyl, Ilyas Mimouni, Isak Combrinck, itzmukeshy7, Jason Watkins, JedaiCoder, Jerry Coffin, Jim Clark, Johan Lundberg, Jon Harper, jotik, Justin, Justin Time, JVApen, K48, Ken Y-N, Keshav Sharma, kiner_shah, krOoze, Leandros, maccard, Malcolm, Malick, Manan Sharma, manetsus, manlio, Marco A., Mark Gardner, MasterHD, Matt, Matt Lord, mnoronha, Muhammad Aladdin, Mustaghees, muXXmit2X, mynameisausten, Nathan Osman, Neil A., Nemanja Boric, neuro, Nicol Bolas, noɥʎdʎzɐɔO, Optimus Prime, Pavel Strakhov, Peter, Praetorian, Qchmqs, Quirk, RamenChef, Rushikesh Deshpande, SajithP, Sam Cristall, Serikov, Shoe, SirGuy, Soapy, Soul_man, theo2003, ɬolɐəz əuɬ qoq, Tom K, TriskalJM, Trizzle, UncleZeiv, VermillionAzure, Walter, Wen Qin, Wexiwa, πάντα ῥεῖ, パスカル
2	Administracion de recursos	Anonymous1847
3	Alcances	deepmax, Error - Syntactical Remorse
4	Algoritmos de la biblioteca estándar	Ami Tavory, Barry, Daniel, Duly Kinsky, Edgar Rokyan, Guillaume Pascal, Jarod42, NinjaDeveloper, Patryk Obara, Peter, Riom
5	Alineación	Brian, Marco A., Nicol Bolas
6	Archivo I / O	anderas, ankit dassor, Anonymous1847, AProgrammer, Bakhtiar Hasan, bitek, Chachmu, ComicSansMS, didiz, Dietmar Kühl, Dr t, Emanuel Vintilă, Galik, honk, Hurkyl, Jérémie Bolduc , John Strood, JVApen, Loki Astari, manlio, Mathieu K., MikeMB , mindriot, Nicol Bolas, nwp, patmanpato, Rakete1111, RomCoo, Serikov, sheng09, shrike, svgspnr, Алексей

		Неудачин
7	Archivos de encabezado	RamenChef, VermillionAzure
8	Aritmética de punto flotante	Xirema
9	Arrays	Cheers and hth. - Alf, Isak Combrinck, manlio, Matthew Brien, Wen Qin, Wolf, ΦXocę Пepeúpa ツ
10	Atributos	ibrahim5253, JVApen, Kerrek SB, MathSquared, SingerOfTheFall
11	auto	Artalus, Barry, celtschk, Daniele Pallastrelli, Edward, Igor Oks, Jarod42, Johan Lundberg, manlio, Yakk
12	Bucles	ankit dassor, anotherGatsby, Barry, ChemiCalChems, Chris, ChrisN, Christian Rau, ColleenV, Debanjan Dhar, DrZoo, Edward, emlai, holmicz, honk, Johannes Schaub - litb, Justin Time, L.V.Rao, manlio, Nicholas, Nicol Bolas, Null, Ped7g, pmelanson, Pyves, Rakete1111, Sergey, sp2danny, user1336087, VladimirS, Yakk
13	Búsqueda de nombre dependiente del argumento	Fanael, Johannes Schaub - litb
14	C ++ Streams	Ami Tavory, didiz, JVApen, mpromonet, Sergey
15	Campos de bits	Ajay, Perette Barella
16	Categorías de valor	Barry, ChemiCalChems, Curious, fefe, Johannes Schaub - litb, mnoronha, Nicol Bolas, Praetorian, SirGuy
17	Clases / Estructuras	Alexey Voytenko, anderas, aquirdturtle, Brian, callyalater, chrisb2244, Colin Basnett, Dan Hulme, darkpsychic, Dragma, Fantastic Mr Fox, Firas Moalla, Jarod42, Jerry Coffin, jotik, Justin Time, Kerrek SB, Nicol Bolas, Null, OliPro007, PcAF, Ph03n1x, pingul, Rakete1111, Sándor Mátyás Márton, Sergey, silvergasp, Skywrath, Yakk
18	Clasificación	anatolyg, Barry, Daniel, Ivan Kush, maccard, manetsus, manlio, MikeMB, MKAROL, Nicol Bolas, Patrick, Ravi Chandra, SajithP, timrau, Trevor Hickey
19	Comparaciones lado a lado de ejemplos clásicos de C ++	wasthishelpful

	resueltos a través de C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17	
20	Compilando y construyendo	4444, Adhokshaj Mishra, Ami Tavory, ArchbishopOfBanterbury, Barry, Ben Steffan, celtschk, Curious, Donald Duck, Dr t, elvis.dukaj, Fantastic Mr Fox, Florian, greatwolf, Griffin, Isak Combrinck, Jahid, Jarod42, Jason Watkins, Johan Lundberg, jotik, Justin, Justin Time, JVApen, madduci, Malick, manetsus, manlio, Matt, Michael Gaskill, Morten Kristensen, MSD, muXXmit2X, n.m., Nathan Osman, Nemanja Boric, Peter, Quirk , Richard Dally, Sergey, Tharindu Kumara, Toby, Trygve Laugstøl, VermillionAzure
21	Comportamiento definido por la implementación	2501, Bo Persson, Brian, Dutow, Jahid, Jarod42, jotik, Justin Time, lz96, manlio, Nicol Bolas, Peter
22	Comportamiento indefinido	Ami Tavory, AndreiM, Ben Steffan, Brian, Cody Gray, cshu, Dovahkiin, Elias Kosunen, emlai, Emma X, FedeWar, fefe, ggrr, GIRISH kuniyal, Hiura, Jeremi Podlasek, Johannes Schaub - litb, JVApen, kd1508, Ken Y-N, manetsus, manlio, Marco A., Mat, mceo, Motti, Naor Hadar, nbro, Nicol Bolas, Peter, Rakete1111, ralismark, RamenChef, Sebastian Ärleryd, Tannin, Trevor Hickey, Tyler Durden
23	Comportamiento no especificado	AndreiM, Brian, Jarod42, Yakk
24	Concurrencia con OpenMP	Andrea Chua, JVApen, Nicol Bolas, Sumurai8
25	Const Correccion	amanuel2, Justin Time
26	constexpr	Ajay, Brian, diegodfrf, mtb, Null
27	Construir sistemas	Ami Tavory, celtschk, Florian, Jahid, Jason Watkins, Justin, JVApen, Nathan Osman, RamenChef, VermillionAzure
28	Contenedores C ++	John DiFini
29	Control de flujo	anotherGatsby, Brian, JVApen, mkluwe, Qchmqs, RamenChef, Tejendra
30	Conversiones de tipo explícito	4444, Brian, JVApen, Nikola Vasilev
31	Copia elision	Nicol Bolas, TartanLlama
32	Copiando vs	amanuel2, Roland

	Asignación	
33	decltype	Ajay
34	deducción de tipo	Barry, Brian, Emmanuel Mathi-Amorim
35	Devolviendo varios valores de una función	aaronsnoswell, Bakhtiar Hasan, Barry, bitek, celtschk, Christopher Oezbek, Community, DeepCoder, Dr t, Ela782, Fantastic Mr Fox, Galik, honk, J_T, Jarod42, Johan Lundberg, Johannes Schaub - litb, John Slegers, Jon Chesterfield, Kevin Katzke, Let_Me_Be, Loki Astari, M. Sadeq H. E., manetsus, Menasheh, Michael Gaskill, mnoronha, Niall, Nicol Bolas, Null, Peter, Rakete1111, Richard Forrest, Ryan Hilbert, Stephen, T.C., templatetypedef, tenpercent, user3384414, Yakk, Ze Rubeus, パスカル
36	Diseño de tipos de objetos	Brian, Justin Time
37	Ejemplos de servidor cliente	Abhinav Gauniyal
38	El estándar ISO C ++	Bakhtiar Hasan, Barry, C.W.Holeman II, ComicSansMS, didiz, diegodfrf, Guillaume Pascal, Ivan Kush, Johan Lundberg, Justin Time, JVApen, manlio, Marco A., MSalters, Nicol Bolas, sth, vishal
39	El puntero este	amanuel2, Justin Time, RamenChef
40	Enhebrado	Alejandro, amchacon, Brian, CaffeineToCode, ComicSansMS, Dair, defube, didiz, Diligent Key Presser, Galik, James Adkison, james large, Jason Watkins, Jeremi Podlasek, mpromonet, Niall, nwp, Rakete1111, Stephen Cross, Sumurai8, Yakk, ysdx, Yuushi
41	Entrada / salida básica en c ++	Daemon, Nicol Bolas, Владимир Стрелец
42	Enumeración	Denkkar, Fantastic Mr Fox, Jarod42, Nicol Bolas, SajithP, stackptr, T.C.
43	Errores comunes de compilación / enlazador (GCC)	Asu, immerhart
44	Escriba palabras clave	Brian, Justin Time, Omnifarious, RamenChef
45	Espacios de	anderas, Andrea Chua, Barry, Brian, DeepCoder, emlai, Isak

	nombres	Combrinck, Jarod42, Jérémy Roy, Johannes Schaub - litb, Julien-L, JVApen, Nicol Bolas, Null, Rakete1111, randag, Roland, T.C., tenpercent, Yakk
46	Especificaciones de vinculación	Brian
47	Especificadores de clase de almacenamiento	Brian, start2learn
48	Estructuras de datos en C ++	Gaurav Sehgal
49	Estructuras de sincronización de hilos.	didiz, Galik, JVApen
50	Excepciones	Alexey Guseynov, Brian, callyalater, Dr t, Jahid, Jarod42, Johan Lundberg, jotik, Martin Ba, Nemanja Boric, Null, Peter, Rakete1111, Ronen Ness
51	Expresiones Fold	AndyG, Barry, cpplearner, Firas Moalla, Marco A., Rakete1111, T.C., Yakk
52	Expresiones regulares	honk, Jonathan Mee, Justin, JVApen
53	Fecha y hora usando encabezamiento	Edward, marcinj, Naor Hadar, RamenChef
54	Función de C ++ "llamada por valor" vs. "llamada por referencia"	Error - Syntactical Remorse, Henkersmann
55	Función de sobrecarga de plantillas	Johannes Schaub - litb, Kunal Tyagi, RamenChef
56	Funciones de miembro de clase constante	Vijayabhaskarreddy CH, Yakk
57	Funciones de miembro virtual	0x5f3759df, Daksh Gupta, Johan Lundberg, Justin Time, Motti, Sergey, T.C.
58	Funciones en linea	amanuel2, Aravind .KEN, Bim, Brian, legends2k
59	Funciones	Barry, krOoze, OliPro007, Reuben Thomas, TriskalJM

	especiales para miembros	
60	Funciones miembro no estáticas	Justin Time, RamenChef
61	Futuros y Promesas	didiz, Nicol Bolas
62	Generación de números aleatorios	Ha., manlio, merlinND, Sumurai8
63	Gestión de la memoria	Andrei, Brian, callyalater, Daksh Gupta, Galik, JVApen, madduci, nnrales, RamenChef, ThyReaper
64	Herramientas y Técnicas de Depuración y Prevención de Depuración de C ++	Adam Trhon, JVApen, King's jester, Misgevolution
65	Idioma Pimpl	Danh, Daniele Pallastrelli, emlai, Jordan Chapman, JVApen, manlio, Stephen Cross, Yakk
66	Implementación de patrones de diseño en C ++	Antonio Barreto, datosh, didiz, Jarod42, JVApen, Nikola Vasilev
67	Incompatibilidades C	パスカル
68	Inferencia de tipos	Andrea Chua, Jim Clark
69	Internacionalización en C ++	John Bargman
70	Iteración	Brian, Daniel Käfer, Emmanuel Mathi-Amorim, marquesm91, RamenChef
71	Iteradores	Barry, chrisb2244, cute_ptr, Daniel Jour, Edgar Rokyan, EvgeniyZh, fbrereto, Gal Dreiman, Gaurav Kumar Garg, GIRISH kuniyal, honk, Hurkyl, JPNotADragon, JVApen, Mike H-R, Null, Oz., Sergey, Serikov, tilz0R, Yakk
72	La Regla De Tres, Cinco Y Cero	Adrien Descamps, Barry, ChrisN, hello, honk, Johan Lundberg, Justin Time, JVApen, Loki Astari, mpromonet, Nicol Bolas, Nirmal4G, NonNumeric, Null, Peter, relgukxilef, Scott Weldon, T.C., TriskalJM, Venemo
73	Lambdas	Adi Lester, Aganju, Ajay, alain, anderas, Andrea Corbelli, Barry, bcmpinc, Brian, Christopher Oezbek, Community, cpplearner, derekerdmann, Edd, Falias, Firas Moalla, honk, Jean-Baptiste

		Yunès, Johan Lundberg, Johannes Schaub - litb, John Slegers, JVApen, Loki Astari, Loufylouf, M. Viaz, Mike Dvorkin, Nicol Bolas, Patryk, Praetorian, Rakete1111, RamenChef, Ryan Haining, Sergio, Serikov, Snowhawk, teivaz, Yakk, ygram
74	Literales	Brian, Nikola Vasilev, RamenChef
75	Literales definidos por el usuario	Brian, Cid1025, Jarod42, Roland, sigalor, sth
76	Manipulación de bits	A. Sarid, Barry, Cody Gray, CroCo, FedeWar, Jarod42, JVApen, manlio, tambre, Tarod, Trevor Hickey, Алексей Неудачин
77	Manipuladores de corriente	Nicol Bolas, Владимир Стрелец
78	Más comportamientos indefinidos en C ++	didiz
79	Mejoramiento	chema989, ralismark
80	Metaprogramacion	anderas, Barry, Brian, celtschk, Colin Basnett, DawidPi, deepmax, dmi_, Holt, Jarod42, Justin, manlio, Matthieu M., Nicol Bolas, Oz., rhynodegreat, rtmh, sth, TartanLlama, Venki, W.F., ysdx, πάντα ῥεῖ
81	Metaprogramacion aritmica	Meena Alfons
82	Modelo de memoria C ++ 11	NonNumeric
83	Mover la semantica	Barry, Cheers and hth. - Alf, ChemiCalChems, David Doria, didiz, Guillaume Racicot, Justin Time, JVApen
84	Mutex recursivo	didiz
85	Mutexes	didiz, hyoslee, JVApen
86	Objetos callables	JVApen, turoni
87	Operadores de Bits	Loki Astari, Mads Marquart, manlio, txtechhelp, Алексей Неудачин
88	Optimización en C ++	4444, JVApen, lorro, mindriot
89	Palabra clave amigo	Perette Barella, Sergey

90	palabra clave const	Barry, Jarod42, Jatin, Justin, Podgorskiy, tenpercent, ThyReaper
91	palabra clave mutable	Barry, Community, Dean Seo, start2learn, T.C., tenpercent
92	Palabras clave	ADITYA, amanuel2, Brian, Danh, John London, Justin Time, JVApen, Kerrek SB, Loki Astari, manlio, Marco A., Nicol Bolas, OliPro007, Rakete1111, RamenChef, Roland, start2learn
93	Palabras clave de la declaración variable	Brian, RamenChef, start2learn
94	Palabras clave de tipo básico	amanuel2, Brian, Kerrek SB, RamenChef
95	Paquetes de parametros	Marco A.
96	Patrón de diseño Singleton	deepmax, Galik, Jarod42, Johan Lundberg, JVApen, Stradigos
97	Patrón de Plantilla Curiosamente Recurrente (CRTP)	Barry, Brian, Gabriel, honk, Nicol Bolas, Ryan Haining
98	Perfilado	Ami Tavory, paul-g
99	Plantillas	Barry, Benjy Kessler, Brian, callyalater, cb4, celtschk, CodeMouse92, Colin Basnett, DeepCoder, Diligent Key Presser, Eldritch Cheese, eXPerience, FedeWar, Gabriel, Greg , Holt, honk, J_T, Johannes Schaub - litb, Justin, JVApen, Loki Astari, M. Viaz, manlio, Maxito, MSalters, Nicol Bolas, Pontus Gagge, Praetorian, Rakete1111, Ricardo Amores, Ryan Haining, Sergey, SirGuy, Smeeheey, Sumurai8, user1887915, W.F., WMios, Wolf, πάντα ῥεῖ
100	Plantillas de expresiones	Ajay, BigONotation, celtschk, defube, Jarod42, Jonathan Lee, Roland, T.C., Yakk
101	Polimorfismo	A. Sarid, Christophe, Jarod42, Jeremi Podlasek, Justin Time, manlio
102	precedencia del operador	an0o0nym, Brian, didiz, JVApen, start2learn, turoni
103	Preprocesador	alain, callyalater, Cheers and hth. - Alf, CygnusX1, Fantastic Mr Fox, Fox, Francisco P., Ian Ringrose, immerhart, InitializeSahib , Johan Lundberg, Justin, Justin Time, Ken Y-N, Kieran

		Chandler, krOoze, manlio, Marco A., Maxito, n.m., Nicol Bolas, Peter, phandinhlan, Richard Dally, Sean, signal, silvergasp, Sumurai8, T.C., Tanjim Hossain, tenpercent, The Philomath, Владимир Стрелец, パスカル
104	Pruebas unitarias en C ++	elvis.dukaj, VermillionAzure
105	Punteros	Baron, daB0bby, FedeWar, Hindrik Stegenga, Nicol Bolas, Nitinkumar Ambekar, Pietro Saccardi, Reverie Wisp, West
106	Punteros a los miembros	John Burger, start2learn
107	Punteros inteligentes	Abyx, Ajay, Alexey Voytenko, anderas, Barry, CaffeineToCode, Christopher Oezbek, Cody Gray, ComicSansMS, cpplearner, Daksh Gupta, Danh, Daniele Pallastrelli, DeepCoder, Edward, emlai, foxcub, Francis Cugler, honk, Jack Zhou, Jared Payne, Jarod42, Johan Lundberg, Johannes Schaub - litb, jotik, Justin, JVApen, Kerrek SB, King's jester, Loki Astari, manlio, Marco A., MC93, Menasheh, Meysam, PcAF, Rakete1111, Reuben Thomas, Richard Dally, rodrigo, Roland, sami1592, sth, Sumurai8, tysonite, user3684240, Xirema, Yakk
108	RAII: la adquisición de recursos es la inicialización	Barry, defube, Jarod42, JVApen, Loki Astari, Niall, Nicol Bolas, RamenChef, Sumurai8, Tannin
109	Recursion en C ++	celtschk, R_Kapp
110	Reenvío perfecto	In silico, Johannes Schaub - litb, Nicol Bolas, Roland
111	Referencias	Andrea Corbelli, Asu, Daksh Gupta, darkpsychic, rockoder
112	Regla de una definición (ODR)	Brian, Jarod42
113	Resolución de sobrecarga	Barry, Brian, didiz, Johannes Schaub - litb
114	RTTI: Información de tipo de tiempo de ejecución	Brian, deepmax, Pankaj Kumar Boora, Roland, Savas Mikail KAPLAN
115	Semáforo	didiz
116	Separadores de dígitos	diegodfrf, JVApen
117	SFINAE (el fallo de	Barry, Fox, Jarod42, Jason R, Jonathan Lee, Luc Danton,

	sustitución no es un error)	sp2danny, SU3, w1th0utnam3, Xosdy, Yakk
118	Sobrecarga de funciones	Bakhtiar Hasan, Barry, Cody Gray, CoffeeandCode, didiz, Galik , Jatin, Johannes Schaub - litb, Justin Time, JVApen, Rakete1111, Sean, Sumurai8, Tim Straubinger
119	Sobrecarga del operador	ArchbishopOfBanterbury, Archie Gertsman, Ates Goral, Barry, Brian, Candlemancer, chrisb2244, defube, enzom83, James Adkison, Rakete1111, Sergey, start2learn, Xeverous, Yakk
120	static_assert	Jarod42, JVApen, lorro, Marco A., Richard Dally, T.C.
121	std :: array	CinCout, Daksh Gupta, Dinesh Khandelwal, Error - Syntactical Remorse, Malcolm, Nikola Vasilev, plasmacel
122	std :: atómica	Nikola Vasilev
123	std :: cualquiera	demonplus, Marco A.
124	std :: forward_list	Nikola Vasilev
125	std :: function: Para envolver cualquier elemento que sea llamable	elimad, Evgeniy, Nicol Bolas, Tarod
126	std :: integer_sequence	Dietmar Kühl
127	std :: iomanip	kiner_shah, Nikola Vasilev, Yakk
128	std :: map	Andrea Corbelli, ankit dassor, ChrisN, CinCout, ComicSansMS, CygnusX1, davidsheldon, diegodfrf, Fantastic Mr Fox, foxcub, Galik, honk, jmmut, manetsus, manlio, Meysam, Naveen Mittal, Null, Peter, Richard Dally, rick112358, Savan Morya, user1336087, vdaras, VolkA, Wyzard
129	std :: opcional	Barry, diegodfrf, Jahid, Jared Payne, JVApen, Null, Yakk
130	std :: par	Ajay, Bim, demonplus, kiner_shah, Nikola Vasilev, Ravi Chandra
131	std :: set y std :: multiset	G-Man, JVApen, Mikitori
132	std :: string	1337ninja, 3442, Andrea Corbelli, Barry, Bim, caps, Christopher Oezbek, cpplearner, crea7or, Curious, drov, Edward, Emil Rowland, emlai, Fantastic Mr Fox, fbrereto, ggrr, Holt, honk, immerhart, Jack, Jahid, Jerry Coffin, Jonathan Mee, jotik,

		JPNotADragon, jpo38, Justin, JVApen, Ken Y-N, Leandros, Loki Astari, manetsus, manlio, Marc.2377, Matthew, Matthieu M., Meysam, Michael Gaskill, mpromonet, Niall, Null, Rakete1111, RamenChef, Richard Dally, SajithP, Serikov, sigalor, Skipper, Soapy, sth, T.C., Tharindu Kumara, Trevor Hickey, user1336087, user2176127, W.F., Wolf, Yakk
133	std :: variante	Yakk
134	std :: vector	2power10, A. Sarid, Aaron Stein, alain, Alex Logan, Ami Tavory, anatolyg, anderas, Andy, AndyG, arunmoezhi, Bakhtiar Hasan, Barry, Benjamin Lindley, bluefog, bone, CHess, CinCout, Cody Gray, Colin Basnett, ComicSansMS, Community, cute_ptr, Daksh Gupta, Daniel, Daniel Stradowski, Dario, David G., David Yaw, DeepCoder, diegodfrf, dkg, Dr t, Duly Kinsky, Ed Cottrell, Edward, ehudt, emlai, enrico.bacis, Falias, Fantastic Mr Fox, Fox, foxcub, gaazkam, Galik, gartenriese, granmirupa, Holt, honk, Hurkyl, iliketocode, immerhart, Isak Combrinck, Jarod42, Jason Watkins, JHBonarius, Johan Lundberg, John Slegers, jotik, jpo38, JVApen, Kevin Katzke, krOoze, Loki Astari, lordjohncena, manetsus, manlio, Marco A., Matt, Michael Gaskill, Misha Brukman, MotKohn, Motti, mtk, NageN, Niall, Nicol Bolas, Null, patmanpato, Paul Beckingham, paul-g, Ped7g, Praetorian, Pyves, R. Martinho Fernandes, Rakete1111, Randy Taylor, Richard Dally, Roddy, Romain Vincent, Rushikesh Deshpande, Ryan Hilbert, Saint-Martin, Samar Yadav, Samer Tufail, Sayakiss, Serikov, Shoe, silvergasp, Skipper, solidcell, Stephen, sth, strangeqargo, T.C., Tamarous, theo2003, Tom, towi, Trevor Hickey, TriskalJM, user1336087, user2176127, Vladimir Gamalyan, Wolf, Yakk
135	Técnicas de refactorización	asantacreu, Cody Gray, Edward, Jarod42, JVApen, RamenChef
136	Tipo de borrado	Brian, celtschk, greatwolf, Jarod42, Yakk
137	Tipo de Devolución Covarianza	Cheers and hth. - Alf, sorosh_sabz
138	Tipo de rasgos	Jarod42, silvergasp, TartanLlama
139	Tipo de retorno final	Brian, define cindy const, Torbjörn
140	Tipos atómicos	JVApen, Stephen
141	Tipos sin nombre	jotik, Roland, ThyReaper
142	Typedef y alias de tipo	Brian

143	Uniones	manlio, ThyReaper, txtechhelp
144	Usando std :: unordered_map	tulak.hord
145	Utilizando declaración	Brian
146	Valor y semántica de referencia	JVApen, Nicol Bolas
147	Variables en linea	Brian