EBook Gratis

APRENDIZAJE ada

Free unaffiliated eBook created from
Stack Overflow contributors.

#ada

Tabla de contenido

Acerca de

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

It is an unofficial and free ada 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 ada.

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 ada

Observaciones

Ada es un lenguaje de programación computarizado de alto nivel, orientado a objetos, estandarizado internacionalmente, que admite escritura tipográfica y programación estructurada. Más información se puede encontrar aquí .

Versiones

Versión	Fecha de lanzamiento
Ada 2012 (TC-1)	2016-04-01
Ada 2012	2012-12-10
Ada 2005	2007-01-01
Ada 95	1995-12-10
Ada 83	1983-01-01

Examples

Instalación o configuración

Ada es un lenguaje de programación para el que existen múltiples compiladores.

- Uno de estos compiladores, y quizás el más usado, es GNAT. Es parte de la cadena de herramientas de GCC. Se puede instalar desde varias fuentes:
    - El lanzamiento anual de GPL realizado por AdaCore, disponible de forma gratuita en el sitio libre . Esta versión ha sido sometida a todas las pruebas internas que AdaCore hace para sus lanzamientos pro, está disponible en una gran cantidad de plataformas. El compilador y su tiempo de ejecución se publican bajo la licencia GPL, y, a menos que no esté utilizando el tiempo de ejecución, cualquier licencia ejecutable que distribuya también estará cubierta por esta licencia. Para académicos y proyectos en sus etapas iniciales, esto no es un problema.
    - El FSF gcc recibe los mismos parches regularmente. La versión de GNAT puede no estar siempre actualizada, pero se pone al día con regularidad.
    - Varios colaboradores están empaquetando la versión de FSF para varias distribuciones de Linux (sistemas basados en Debian, entre otros) y binarios para Mac OS X. Usar el administrador de paquetes de su distribución puede ser la forma más

sencilla de instalar GNAT. Estas versiones vienen con la licencia GCC estándar y le permiten escribir código fuente cerrado.

- AdaCore también proporciona GNAT Pro , que viene con la licencia GCC estándar que le permite escribir código fuente cerrado. Quizás lo más importante sea que viene con soporte, si tiene preguntas sobre el uso del lenguaje, las herramientas, cómo implementar mejor algo y, por supuesto, los informes de errores y las solicitudes de mejora.

Otro número de compiladores se enumeran en el WikiBook de Ada , junto con las instrucciones de instalación. Getadanow.com presenta ediciones de FSF GNAT, preparadas para varios sistemas operativos en varios tipos de hardware o máquinas virtuales. El sitio también recopila recursos para aprender y compartir Ada.

Hola Mundo

```
with Ada.Text_IO;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("Hello World");
end Hello_World;
```

Alternativamente, después de importar el paquete Ada.Text_IO , puede decir `use Ada.Text_IO;` para poder utilizar Put_Line sin declarar explícitamente de qué paquete debe provenir, como tal:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Hello_World is
begin
    Put_Line ("Hello World");
end Hello_World;
```

Si está utilizando el compilador `gnat` , este sencillo programa puede compilarse con

```
gnatmake hello_world
```

Esto generará una serie de archivos, incluyendo un `hello_world` (o `hello_world.exe` en Windows) que puede ejecutar para ver el mensaje famoso. El nombre del ejecutable se calcula automáticamente a partir del nombre del subprograma principal de Ada. En Ada un subprograma principal puede tener cualquier nombre. Solo tiene que ser un procedimiento sin parámetros, que usted le da como argumento a `gnatmake` .

Otros compiladores tienen requisitos similares, aunque, por supuesto, el comando build es diferente.

Versión

El lenguaje de programación estándar de Ada se define en el *Manual de referencia de Ada* . Los

cambios de la versión interina y las notas de la versión se tratan en los documentos justificativos correspondientes. Las implementaciones generalmente documentan su cumplimiento con el estándar en forma de una guía del usuario y / o un manual de referencia, por ejemplo .

- Ada 2012
    - *Manual de referencia de idiomas de Ada 2012*
    - *Justificación de Ada 2012*
- Ada 2005
    - *Manual de referencia de idiomas de Ada 2005*
    - *Justificación de Ada 2005*
- Ada 95
    - *Ada 95 Language Reference Manua* l
    - *Justificación de Ada 95*
- Ada 83
    - *Ada 83 Language Reference Manua* l
    - *Justificación de Ada 83 para el diseño del lenguaje de programación Ada®*

Bibliotecas

Como para cualquier lenguaje de programación, Ada viene con extensas bibliotecas para realizar diversas tareas. Aquí hay algunos consejos para algunos de ellos, aunque la búsqueda en github llevará a algunos más.

- El propio tiempo de ejecución de Ada, distribuido con todos los compiladores, incluye un extenso conjunto de paquetes y anexos, que van desde estructuras de datos y contenedores, hasta entrada / salida, manipulación de cadenas, manipulación de tiempo, archivos, cálculos numéricos, multitarea, interruptores de línea de comando, números al azar,...
- El compilador GNAT viene con su propio tiempo de ejecución extendido, con nuevos paquetes en la jerarquía `GNAT` , que brindan soporte para expresiones regulares, clasificación, búsqueda, Unicode, CRC, entrada / salida de tiempo, ...
- gnatcoll es una biblioteca que está disponible en el sitio libre de AdaCore e incluye un amplio marco de registro, extendiendo las aplicaciones con python, mmap, un amplio marco para interactuar con los sistemas de archivos, analizando los correos electrónicos y los buzones de correo, un amplio marco para interactuar con las bases de datos en un Manera segura de tipos, interfaz con varias bibliotecas como ícono, línea de lectura, colores de terminal, soporte para tipos de recuento de referencia para la administración automática de memoria, archivos JSON, ...
- XML / Ada es una biblioteca para analizar y validar documentos XML

- GtkAda es un enlace completo a la biblioteca gtk +, que le permite escribir interfaces de usuario portátiles en Unix, Windows y OSX.
- AWS es un marco para crear servidores web en Ada, con soporte completo para varios protocolos como HTTP, Websockets, ... y su propio sistema de plantillas.

Lea Empezando con ada en línea: https://riptutorial.com/es/ada/topic/3900/empezando-con-ada

Capítulo 2: Archivos y flujos de E / S

Observaciones

La biblioteca estándar de Ada proporciona E / S de archivos tradicionales de texto o datos binarios, así como E / S de archivos transmitidos. Los archivos de datos binarios serán secuencias de valores de un tipo, mientras que los archivos de flujo pueden ser secuencias de valores de tipos posiblemente diferentes.

Para leer y escribir elementos de diferentes tipos desde / a archivos de transmisión, Ada utiliza subprogramas indicados por los atributos de los tipos, a saber, `'Read` , `'Write` , `'Input` y `'Output` . Estos dos últimos leer y escribir límites de la matriz, discriminantes de registro, y etiquetas de tipo, además de la entrada y la salida al descubierto que `Read` y `'Write` llevará a cabo.

Examples

Crear y escribir al archivo.

Los procedimientos `Create` , `Put_Line` , `Close` del paquete `Ada.Text_IO` se utilizan para crear y escribir en el archivo `file.txt` .

```
with Ada.Text_IO;

procedure Main is
   use Ada.Text_IO;
   F : File_Type;
begin
   Create (F, Out_File, "file.txt");
   Put_Line (F, "This string will be written to the file file.txt");
   Close (F);
end;
```

Archivo resultante `file.txt`

```
This string will be written to the file.txt
```

Crear y escribir en una secuencia

Los atributos orientados a la secuencia de los subtipos se llaman para escribir objetos en un archivo, desnudos y utilizando representaciones binarias predeterminadas.

```
with Ada.Streams.Stream_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   type Color_Value is range 0 .. 255;
```

```
   type Color is record
      R, G, B : Color_Value;
   end record;

   Fruit_Colors : constant array (Fruit) of Color :=
     (Banana => Color'(R => 243, G => 227, B => 18),
      Orange => Color'(R => 251, G => 130, B => 51),
      Pear   => Color'(R => 158, G => 181, B => 94));

   use Ada.Streams.Stream_IO;

   F : File_Type;

begin
   Create (F, Name => "file.bin");
   for C in Fruit_Colors'Range loop
      Fruit'Write (Stream (F), C);
      Color'Write (Stream (F), Fruit_Colors (C));
   end loop;
   Close (F);
end Main;
```

Archivo resultante

```
00000000  00 2e f3 00 e3 00 12 00  01 2e fb 00 82 00 33 00
00000010  02 2e 9e 00 b5 00 5e 00
```

Abrir y leer desde el archivo Stream

Lea los datos de Crear y escribir en una secuencia de nuevo en un programa.

```
with Ada.Streams.Stream_IO;

procedure Main is
   --
   --  ... same type definitions as in referenced example
   --
   Fruit_Colors : array (Fruit) of Color;

   use Ada.Streams.Stream_IO;

   F : File_Type;
   X : Fruit;
begin
   Open (F, Mode => In_File, Name => "file.bin");
   loop
      Fruit'Read (Stream (F), X);
      Color'Read (Stream (F), Fruit_Colors (X));
   end loop;
exception
   when End_Error =>
      Close (F);
   pragma Assert  -- check data are the same
     (Fruit_Colors (Banana) = Color'(R => 243, G => 227, B => 18) and
      Fruit_Colors (Orange) = Color'(R => 251, G => 130, B => 51) and
      Fruit_Colors (Pear)   = Color'(R => 158, G => 181, B => 94));
```

```
end Main;
```

Lea Archivos y flujos de E / S en línea: https://riptutorial.com/es/ada/topic/8865/archivos-y-flujos-de-e---s

Capítulo 3: Enumeración

Sintaxis

- function Enumeration ' Image (Argument: Enumeration'Base) return String;
- function Enumeration ' Img return String; - GNAT
- function Enumeration'Val (Argument: Universal_Integer) return Enumeration'Base;
- function Enumeration'Pos (Argument: Enumeration'Base) return Universal_Integer;
- function Enumeration'Enum_Rep (Argument: Enumeration'Base) return universal_Integer;
- function **Literal** 'Enum_Rep return Universal_Integer; - GNAT
- función **Literal** 'Dirección de retorno System.Address;
- para uso de enumeración (Literal_1 => **Universal_Integer** , Literal_n => **Universal_Integer**);
- (**Literal** in Enumeration) return Boolean;

Examples

Literales iterando

Un literal dentro de una enumeración es un tipo discreto, por lo que podemos usar la `Image` atributo para averiguar qué literal es el formulario de texto. Observe que esto imprime la misma palabra que en el código (pero en mayúsculas).

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
begin
   for I in Fruit loop
      Put (Fruit'Image (I));
      New_Line;
   end loop;
end;
```

Resultado

```
BANANA
PEAR
ORANGE
MELON
```

Usando el paquete Enumeration_IO

En lugar del atributo `Image` y `Ada.Text_IO.Put` en literales de enumeración, solo podemos usar el paquete genérico `Ada.Text_IO.Enumeration_IO` para imprimir los literales.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
   package Fruit_IO is new Enumeration_IO (Fruit); use Fruit_IO;
begin
   for I in Fruit loop
      Put (I);
      New_Line;
   end loop;
end;
```

Resultado

```
BANANA
PEAR
ORANGE
MELON
```

Primer carácter mayúscula resto minúsculas literales

`Image` atributo en mayúscula todos los caracteres de literales de enumeración. La función `Case_Rule_For_Names` aplica mayúsculas para el primer carácter y hace que el resto `Case_Rule_For_Names` minúscula.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Maps.Constants; use Ada.Strings.Maps.Constants;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

procedure Main is
   type Fruit is (Banana, Pear, Orange, Melon);
   function Case_Rule_For_Names (Item : String) return String is
   begin
      return Translate (Item (Item'First .. Item'First), Upper_Case_Map) & Translate (Item
(Item'First + 1 .. Item'Last), Lower_Case_Map);
   end;
begin
   for I in Fruit loop
      Put (Case_Rule_For_Names (Fruit'Image (I)));
      New_Line;
   end loop;
end;
```

Resultado

```
Banana
Pear
Orange
Melon
```

Título del caso, Uso de Enumeration_IO, para un subrango

Combinando el caso de cambio de caracteres con Enumeration_IO y utilizando un búfer de texto para la imagen. El primer personaje es manipulado en su lugar.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Characters.Handling; use Ada.Characters.Handling;

procedure Main is
    type Fruit is (Banana, Pear, Orange, Melon);
    package Fruit_IO is new Enumeration_IO (Fruit);
    Buffer : String (1 .. Fruit'Width);
begin
   for I in Fruit range Pear .. Fruit'Last loop
       Fruit_IO.Put (To => Buffer,
                     Item => I,
                     Set => Lower_Case);
       Buffer (Buffer'First) := To_Upper (Buffer (Buffer'First));
       Put_Line (Buffer);
   end loop;
end;
```

Resultado

```
Pear
Orange
Melon
```

Lea Enumeración en línea: https://riptutorial.com/es/ada/topic/5930/enumeracion

Capítulo 4: Genericidad en Ada

Examples

Subprogramas genéricos

Los subprogramas genéricos son útiles para crear subprogramas que tienen la misma estructura para varios tipos. Por ejemplo, para intercambiar dos objetos:

```
generic
    type A_Type is private;
procedure Swap (Left, Right : in out A_Type) is
    Temp : A_Type := Left;
begin
    Left := Right;
    Right := Temp;
end Swap;
```

Paquetes genéricos

En el paquete genérico de Ada, en la instanciación, los datos se duplican; es decir, si contienen variables globales, cada instancia tendrá su propia copia de la variable, correctamente escrita e independiente de las demás.

```
generic
    type T is private;
package Gen is
    type C is tagged record
        V : T;
    end record;
    G : Integer;
end Gen;
```

Parámetros genéricos

Ada ofrece una amplia variedad de parámetros genéricos que son difíciles de traducir a otros idiomas. Los parámetros utilizados durante la creación de instancias y, como consecuencia, aquellos en los que puede confiar la unidad genérica pueden ser variables, tipos, subprogramas o instancias de paquetes, con ciertas propiedades. Por ejemplo, lo siguiente proporciona un algoritmo de clasificación para cualquier tipo de matriz:

```
generic
    type Component is private;
    type Index is (<>);
    with function "<" (Left, Right : Component) return Boolean;
    type Array_Type is array (Index range <>) of Component;
procedure Sort (A : in out Array_Type);
```

Lea Genericidad en Ada en línea: https://riptutorial.com/es/ada/topic/9322/genericidad-en-ada

Capítulo 5: Imagen de atributo

Introducción

Los atributos de subtipo `'Image` y `'Value` tomarán, respectivamente, un valor escalar y una cadena y devolverán, respectivamente, una cadena y un valor escalar. El resultado de `'Image` se puede ingresar a `'Value` para obtener el valor original. Lo contrario también es cierto.

El `__Scalar_Object__'Image` se puede usar directamente en los objetos (desde Ada 2012-TC-1).

Sintaxis

- function Scalar'Image (Argumento: Scalar'Base) return String;
- function Discrete'Image (Argument: Discrete'Base) return String;
- function Integer'Image (Argumento: Integer'Base) return String;
- function Enumeration'Image (Argument: Enumeration'Base) return String;
- function Real'Image (Argumento: Real'Base) return String;
- function Numeric'Image (Argumento: Numeric'Base) return String;
- función Scalar'Value (Argument: String) return Scalar'Base;
- function Discrete'Value (Argument: String) return Discrete'Base;
- función Integer'Value (Argumento: String) return Integer'Base;
- function Enumeration'Value (Argument: String) return Enumeration'Base;
- function Real'Value (Argument: String) return Real'Base;
- function **Scalar_Object** 'Image return String;

Observaciones

Tenga en cuenta que `'Image` puede incurrir en resultados definidos de implementación (RM 3.5), es decir, cuando algunos caracteres gráficos necesarios para el resultado de `String` no están definidos en `Character` . Considere los repertorios más grandes de `'Wide_Image` y `'Wide_Wide_Image` .

Ada 2012 (TC-1)

El permiso para usar el atributo `__Scalar_Object__'Image` directamente en un objeto se agregó en Ada 2012-TC-1 (abril de 2016).

Examples

Imprima el flotador usando el atributo Imagen

Ada 2012 (TC-1)

```
with Ada.Text_IO;

procedure Main is
```

```
   type Some_Float digits 8 range 0.0 .. 10.0;
   X : Some_Float := 2.71;
begin
   Ada.Text_IO.Put_Line (X'Image);
end Main;
```

Resultado

```
2.71000E+00
```

Imprima entero utilizando el atributo de imagen

Ada 2012 (TC-1)

```
with Ada.Text_IO;

procedure Main is
   type Some_Integer is range -42 .. 42;
   X : Some_Integer := 17;
begin
   Ada.Text_IO.Put_Line (X'Image);
end Main;
```

Resultado

```
17
```

Imprima la enumeración usando el atributo Imagen

Ada 2012 (TC-1)

```
with Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   X : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (X'Image);
   Ada.Text_IO.Put_Line (Pear'Image);
end Main;
```

Resultado

```
ORANGE
PEAR
```

Imprima la enumeración usando el atributo Imagen

```
with Ada.Text_IO;

procedure Main is
   type Fruit is (Banana, Orange, Pear);
   X : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (Fruit'Image (X));
end Main;
```

Resultado

```
ORANGE
```

Imprimir entero con atributo de imagen

```
with Ada.Text_IO;

procedure Main is
   X : Integer := 17;
begin
   Ada.Text_IO.Put_Line (Integer'Image (X));
end Main;
```

Resultado

```
17
```

Imprime Float usando el atributo Imagen

```
with Ada.Text_IO;

procedure Main is
   X : Float := 2.71;
begin
   Ada.Text_IO.Put_Line (Float'Image (X));
end Main;
```

Resultado

```
2.71000E+00
```

Como lo inverso

```
with Ada.Text_IO;

procedure Image_And_Value is
   type Fruit is (Banana, Orange, Pear);
   X  : Fruit := Orange;
begin
   Ada.Text_IO.Put_Line (Boolean'Image
      (Fruit'Value (Fruit'Image (X)) = X
          and
       Fruit'Image (Fruit'Value ("ORANGE")) = "ORANGE"));
end Image_And_Value;
```

Resultado

```
TRUE
```

Lea Imagen de atributo en línea: https://riptutorial.com/es/ada/topic/4290/imagen-de-atributo

Capítulo 6: Implementando el patrón productor-consumidor.

Introducción

Una demostración de cómo se implementa el patrón productor-consumidor en Ada.

Sintaxis

- function Scalar ' Image (Argumento: Scalar'Base) return String;
- tarea Nombre_trabajo;
- task Task_Name es final de entradas;
- cuerpo de tarea Nombre_trabajo es Declaraciones comienzan Código final;
- entrada Entry_Name;
- aceptar Entry_Name;
- salida;

Observaciones

Todos los ejemplos *deben* garantizar la terminación correcta de la tarea.

Examples

Utilizando un buffer sincronizado

```
with Ada.Containers.Synchronized_Queue_Interfaces;
with Ada.Containers.Unbounded_Synchronized_Queues;
with Ada.Text_IO;

procedure Producer_Consumer_V1 is
   type Work_Item is range 1 .. 100;

   package Work_Item_Queue_Interfaces is
     new Ada.Containers.Synchronized_Queue_Interfaces
           (Element_Type => Work_Item);

   package Work_Item_Queues is
     new Ada.Containers.Unbounded_Synchronized_Queues
           (Queue_Interfaces => Work_Item_Queue_Interfaces);

   Queue : Work_Item_Queues.Queue;

   task type Producer;
   task type Consumer;

   Producers : array (1 .. 1)  of Producer;
   Consumers : array (1 .. 10) of Consumer;
```

```
   task body Producer is
   begin
      for Item in Work_Item loop
         Queue.Enqueue (New_Item => Item);
      end loop;
   end Producer;

   task body Consumer is
      Item : Work_Item;
   begin
      loop
         Queue.Dequeue (Element => Item);
         Ada.Text_IO.Put_Line (Work_Item'Image (Item));
      end loop;
   end Consumer;

begin
   null;
end Producer_Consumer_V1;
```

Observe que he sido perezoso aquí: no hay una terminación adecuada de las tareas del consumidor, una vez que se consumen todos los elementos de trabajo.

Patrón productor-consumidor utilizando el mecanismo Ada Rendezvous.

Una solución sincrónica productor-consumidor asegura que el consumidor lea cada elemento de datos escritos por el productor una sola vez. Las soluciones asíncronas permiten al consumidor muestrear la salida del productor. O bien el consumidor consume los datos más rápido de lo que se produce, o el consumidor consume los datos más lentamente de lo que se produce. El muestreo permite al consumidor manejar los datos disponibles actualmente. Esos datos pueden ser solo una muestra de los datos producidos o pueden ser datos ya consumidos.

```
------------------------------------------------------------------
-- synchronous PC using Rendezvous --
------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;

procedure PC_Rendezvous is
   task Producer;
   task Consumer is
      entry Buf(Item : in Integer);
   end Consumer;
   task body Producer is
   begin
      for I in 1..10 loop
         Put_Line("Producer writing" & Integer'Image(I));
         Consumer.Buf(I);
      end loop;
   end Producer;
   task body Consumer is
      Temp : Integer;
   begin
      loop
         select
            accept Buf(Item : in Integer) do
               temp := Item;
            end;
```

```
            Put_Line("Consumer read" & Integer'Image(Temp));
         or
            terminate;
         end select;
      end loop;
   end Consumer;

begin
   null;
end PC_Rendezvous;
```

Productor-Consumidor con un consumidor muestral.

Este ejemplo utiliza el procedimiento principal como la tarea del productor. En Ada, el procedimiento principal siempre se ejecuta en una tarea separada de todas las demás tareas en el programa, vea el ejemplo mínimo .

```
-------------------------------------------------------------------
-- Sampling Consumer --
-------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;

procedure Sampling_PC is
   protected Buf is
      procedure Write(Item : in Integer);
      function Read return Integer;
      procedure Set_Done;
      function Get_Done return Boolean;
   private
      Value : Integer := Integer'First;
      Is_Done : Boolean := False;
   end Buf;
   protected body Buf is
      procedure Write(Item : in Integer) is
      begin
         Value := Item;
      end Write;
      function Read return Integer is
      begin
         return Value;
      end Read;
      procedure Set_Done is
      begin
         Is_Done := True;
      end Set_Done;
      function Get_Done return Boolean is
      begin
         return Is_Done;
      end Get_Done;
   end Buf;

   task Consumer;
   task body Consumer is
   begin
      while not Buf.Get_Done loop
          Put_Line("Consumer read" & Integer'Image(Buf.Read));
      end loop;
   end Consumer;
```

```
begin
   for I in 1..10 loop
     Put_Line("Producer writing" & Integer'Image(I));
     Buf.Write(I);
   end loop;
   Buf.Set_Done;
end Sampling_PC;
```

Varios productores y consumidores compartiendo el mismo búfer

Este ejemplo muestra a varios productores y consumidores compartiendo el mismo búfer. Las entradas protegidas en Ada implementan una cola para manejar las tareas en espera. La política de cola predeterminada es Primero en entrar, primero en salir.

```
-------------------------------------------------------------------
-- Multiple producers and consumers sharing the same buffer --
-------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_Io;

procedure N_Prod_Con is
   protected Buffer is
      Entry Write(Item : in Integer);
      Entry Read(Item : Out Integer);
   private
      Value  : Integer := Integer'Last;
      Is_New : Boolean := False;
   end Buffer;
   protected body Buffer is
      Entry Write(Item : in Integer) when not Is_New is
      begin
         Value := Item;
         Is_New := True;
      end Write;
      Entry Read(Item : out Integer) when Is_New is
      begin
         Item := Value;
         Is_New := False;
      end Read;
   end Buffer;

   task type Producers(Id : Positive) is
      Entry Stop;
   end Producers;
   task body Producers is
      Num : Positive := 1;
   begin
      loop
         select
            accept Stop;
            exit;
         or
            delay 0.0001;
         end select;
         Put_Line("Producer" & Integer'Image(Id) & " writing" & Integer'Image(Num));
         Buffer.Write(Num);
         Num := Num + 1;
      end loop;
```

```
   end Producers;

   task type Consumers(Id : Positive) is
      Entry Stop;
   end Consumers;

   task body Consumers is
      Num : Integer;
   begin
      loop
         select
            accept stop;
            exit;
         or
            delay 0.0001;
         end select;
         Buffer.Read(Num);
         Put_Line("Consumer" & Integer'Image(ID) & " read" & Integer'Image(Num));
      end loop;
   end Consumers;
   P1 : Producers(1);
   P2 : Producers(2);
   P3 : Producers(3);
   C1 : Consumers(1);
   C2 : Consumers(2);
   C3 : Consumers(3);
begin
   delay 0.2;
   P1.Stop;
   P2.Stop;
   P3.Stop;
   C1.Stop;
   C2.Stop;
   C3.Stop;
end N_Prod_Con;
```

Lea Implementando el patrón productor-consumidor. en línea:
https://riptutorial.com/es/ada/topic/8632/implementando-el-patron-productor-consumidor-

Capítulo 7: Números de salida

Introducción

Los paquetes estándar de Ada proporcionan la salida de todos los tipos numéricos. El formato de salida se puede ajustar de muchas maneras.

Observaciones

Observe cómo cada vez que un paquete genérico se crea una instancia con un tipo numérico. Además, hay dos valores predeterminados que deben configurarse para toda la instancia y también formas de anular el `Width` , por ejemplo, al llamar a `Put` con este parámetro.

Examples

Imprimir enteros, generosamente usando espacio.

Las instancias de `Integer_IO` tienen una variable de configuración `Default_Width` que tomará la cantidad de caracteres que tomará cada número de salida.

```
with Ada.Text_IO;   use Ada.Text_IO;

procedure Print_Integer is
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Count_IO is new Integer_IO (Count);
    X : Count;
begin
    Count_IO.Default_Width := 12;

    X := Count'First;
    while X < Count'Last loop
        Count_IO.Put (X);
        Count_IO.Put (X + 1);
        New_Line;

        X := X + 500_000;
    end loop;
end Print_Integer;
```

Resultado

```
    -1000000
     -500000
           0
      500000
```

Imprimir números enteros, utilizando Base 16 (hexadecimal)

Una variable de configuración `Default_Base` se establece en la instancia de `Ada.Text_IO.Integer_IO` ; también, `Default_Width` se establece para que la salida no pueda tener espacio inicial.

```
with Ada.Text_IO;   use Ada.Text_IO;

procedure Print_Hex is
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Count_IO is new Integer_IO (Count);
    X : Count;
begin
    Count_IO.Default_Width := 1;
    Count_IO.Default_Base := 16;

    X := Count'First;
    while X < Count'Last loop
        Count_IO.Put (X);
        New_Line;

        X := X + 500_000;
    end loop;
end Print_Hex;
```

Resultado

```
-16#F4240#
-16#7A120#
16#0#
16#7A120#
```

Imprimir números de punto fijo decimal, también conocido como dinero

`Ada.Text_IO.Editing` ofrece el formato de valores de puntos fijos decimales utilizando "cadenas de imagen". Estos describen la salida utilizando caracteres "mágicos" para separadores, signos de moneda, etc.

```
with Ada.Text_IO.Editing;   use Ada.Text_IO;

procedure Print_Value is

    Max_Count      : constant := 1_000_000;

    type Fruit is (Banana, Orange, Pear);
    subtype Count is Integer range -Max_Count .. +Max_Count;

    type Money is delta 0.001 digits 10;

    package Fruit_IO is new Enumeration_IO (Fruit);
    package Money_IO is new Editing.Decimal_Output
      (Money,
       Default_Currency => "CHF",
```

```
       Default_Separator => ''');

    Inventory : constant array (Fruit) of Count :=
      (Banana => +27_420,
       Orange => +140_600,
       Pear   => -10_000);

    Price_List : constant array (Fruit) of Money :=
      (Banana => 0.07,
       Orange => 0.085,
       Pear   => 0.21);

    Format : constant Editing.Picture :=
      Editing.To_Picture ("<###BZ_ZZZ_ZZ9.99>");
begin
    Fruit_IO.Default_Width := 12;

    for F in Inventory'Range loop
        Fruit_IO.Put (F);
        Put          (" | ");
        Money_IO.Put (Item => Inventory (F) * Price_List (F),
                      Pic => Format);
        New_Line;
    end loop;
end Print_Value;
```

Resultado

```
BANANA       |  CHF     1'919.40
ORANGE       |  CHF    11'951.00
PEAR         | (CHF     2'100.00)
```

Imprimir varios artículos en una línea

Combine las instancias de los paquetes `_IO` , use el correcto con su tipo numérico.

```
with Ada.Text_IO;   use Ada.Text_IO;

procedure Print_Inventory is
    type Fruit is (Banana, Orange, Pear);
    subtype Count is Integer range -1_000_000 .. 1_000_000;

    package Fruit_IO is new Enumeration_IO (Fruit);
    package Count_IO is new Integer_IO (Count);

    Inventory : constant array (Fruit) of Count :=
      (Banana => 27_420,
       Orange => 140_600,
       Pear   => -10_000);

begin
    Fruit_IO.Default_Width := 12;

    for F in Inventory'Range loop
        Fruit_IO.Put (F);
        Put          (" | ");
```

```
        Count_IO.Put (Inventory (F));
        New_Line;
    end loop;
end Print_Inventory;
```

Resultado

```
BANANA       |    27420
ORANGE       |   140600
PEAR         |   -10000
```

Lea Números de salida en línea: https://riptutorial.com/es/ada/topic/8940/numeros-de-salida

Capítulo 8: paquete Ada.Text_IO

Introducción

El paquete `Ada.Text_IO` se usa para poner texto u obtener texto de archivos o consola.

Examples

Put_Line

Imprime la cadena con una nueva línea.

```
with Ada.Text_IO;

procedure Put_Text is
    use Ada.Text_IO;
    S : String := "Hello";
begin
    Put_Line ("Hello");
    Put_Line (Standard_Output, "Hello");
    Put_Line (Standard_Error, "Hello error");
    Put_Line (S & " World");
end;
```

Resultado

```
Hello
Hello
Hello error
Hello World
```

Lea paquete Ada.Text_IO en línea: https://riptutorial.com/es/ada/topic/8839/paquete-ada-text-io

Capítulo 9: Paquetes

Sintaxis

- con Package_Name_To_Include;
- package New_Package_Name cambia el nombre de Package_To_Rename;
- use Package_Name;
- paquete Parent_Name.Child_Name es

Observaciones

El paquete proporciona:

- Encapsulamiento de código
- Compilación separada
- Ocultar procedimientos, funciones, operadores en tipos privados.

Similitudes o análogos en otros idiomas:

- Espacio de nombres C ++
- Paquetes de java

Examples

Más sobre Paquetes

En Hello World , se le presentó el paquete `Ada.Text_IO` y cómo utilizarlo para realizar operaciones de E / S dentro de su programa. Los paquetes pueden ser manipulados para hacer muchas cosas diferentes.

El cambio de nombre: Para cambiar el nombre de un paquete, se utiliza la palabra clave `renames` de una declaración del paquete, tales como:

```
package IO renames Ada.Text_IO;
```

Ahora, con el nuevo nombre, puede usar la misma notación de puntos para funciones como `Put_Line` (es decir, `IO.Put_Line`), o simplemente puede `use` con `use IO` . Por supuesto, diciendo que `use IO` o `IO.Put_Line` usará las funciones del paquete `Ada.Text_IO` .

Visibilidad y aislamiento : en el ejemplo de *Hello World* , incluimos el paquete Ada.Text_IO con una cláusula `with` . Pero también `use Ada.Text_IO` que queríamos `use Ada.Text_IO` en la misma línea. El `use Ada.Text_IO` declaración `use Ada.Text_IO` podría haberse movido a la parte declarativa del procedimiento:

```
with Ada.Text_IO;

procedure hello_world is
   use Ada.Text_IO;
begin
   Put_Line ("Hello, world!");
end hello_world;
```

En esta versión, los procedimientos, funciones y tipos de `Ada.Text_IO` están directamente disponibles dentro del procedimiento. Fuera del bloque en el que se declara el uso de `Ada.Text_IO`, tendríamos que usar la notación de puntos para invocar, por ejemplo:

```
with Ada.Text_IO;

procedure hello_world is
begin
   Ada.Text_IO.Put ("Hello, ");       --  The Put function is not directly visible here
   declare
      use Ada.Text_IO;
   begin
      Put_Line ("world!");            --  But here Put_Line is, so no Ada.Text_IO. is needed
   end;
end hello_world;
```

Esto nos permite aislar las declaraciones de uso ... donde sean necesarias.

Relación padre-hijo

Como una forma de subdividir los programas de Ada, los paquetes pueden tener los llamados hijos. Estos también pueden ser paquetes. Un paquete secundario tiene un privilegio especial: puede ver las declaraciones en la parte privada del paquete principal. Un uso típico de esta visibilidad especial es cuando se forma una jerarquía de tipos derivados en la programación orientada a objetos.

```
package Orders is
   type Fruit is (Banana, Orange, Pear);
   type Money is delta 0.01 digits 6;

   type Bill is tagged private;

   procedure Add
     (Slip   : in out Bill;
      Kind   : in     Fruit;
      Amount : in     Natural);

   function How_Much (Slip : Bill) return Money;

   procedure Pay
     (Ordered : in out Bill;
      Giving  : in     Money);

private
   type Bill is tagged record
      --  ...
      Sum : Money := 0.0;
```

```
   end record;
end Orders;
```

Cualquier unidad de Ada que esté encabezada por `with Orders;` puede declarar objetos de tipo `Bill` y luego llamar a las operaciones `Add` , `How_Much` y `Pay` . Sin embargo, no ve los componentes de `Bill` , ni siquiera de `Orders.Bill` , ya que la definición de tipo completo está oculta en la parte **privada** de `Orders` . Sin embargo, la definición completa no está oculta en forma de paquetes secundarios. Esta visibilidad facilita la extensión de tipo si es necesario. Si un tipo se declara en el paquete secundario como derivado de `Bill` , entonces este tipo heredado puede manipular los componentes de `Bill` directamente.

```
package Orders.From_Home is
   type Address is new String (1 .. 120);

   type Ordered_By_Phone is new Bill with private;

   procedure Deliver
     (Ordered : in out Ordered_By_Phone;
      Place   : in     Address);

private
   type Ordered_By_Phone is new Bill with
      record
         Delivered : Boolean := False;
         To        : Address;
      end record;
end Orders.From_Home;
```

`Orders.From_Home` es un paquete secundario de `Orders` . El tipo `Ordered_By_Phone` se deriva de `Bill` e incluye su componente de registro `Sum` .

Lea Paquetes en línea: https://riptutorial.com/es/ada/topic/7322/paquetes

Capítulo 10: Tarea

Sintaxis

- tarea Nombre_trabajo;
- task Task_Name es final de entradas;
- cuerpo de tarea Nombre_trabajo es Declaraciones comienzan Código final;

Examples

Una tarea sencilla

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      Put_Line ("Hello from My_Task");
   end;
begin
   Put_Line ("Hello from Main");
end;
```

Resultado

El orden de `Put_Line` puede variar.

```
Hello from My_Task
Hello from Main
```

Una tarea sencilla y un bucle.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task");
      end loop;
   end;
begin
   Put_Line ("Hello from Main");
end;
```

Resultado

El orden de `Put_Line` puede variar.

```
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from My_Task
Hello from My_Task
```

Una tarea sencilla y dos bucles.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   task My_Task;
   task body My_Task is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task");
      end loop;
   end;
begin
   for I in 1 .. 4 loop
      Put_Line ("Hello from Main");
   end loop;
end;
```

Resultado

El orden de `Put_Line` puede variar.

```
Hello from My_Task
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from Main
Hello from My_Task
Hello from Main
Hello from Main
```

Dos tareas simples y dos bucles.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   task My_Task_1;
   task My_Task_2;

   task body My_Task_1 is
   begin
```

```
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task_1");
      end loop;
   end;

   task body My_Task_2 is
   begin
      for I in 1 .. 4 loop
         Put_Line ("Hello from My_Task_2");
      end loop;
   end;
begin
   null;
end;
```

Resultado

El orden de `Put_Line` puede variar.

```
Hello from My_Task_1
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_1
Hello from My_Task_2
Hello from My_Task_2
```

Una tarea que incrementa un número después de la entrada.

El usuario puede llamar a `Incrementor.Increment K` varias veces presionando una tecla dentro de `'0' .. '9'` y es posible llamar a `Incrementor.Increment` más rápido que la `task Incrementor` o Incrementor `I` puede `task Incrementor` .

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

procedure Main is
   use Ada.Text_IO;
   task Incrementor is
      entry Increment;
   end;
   task body Incrementor is
      use Ada.Integer_Text_IO;
      I : Integer := 0;
   begin
      loop
         accept Increment;
         I := I + 1;
         Put (I, 0);
         delay 0.1;
      end loop;
   end;
   K : Character;
begin
```

```
   loop
      Get_Immediate (K);
      if K in '0' .. '9' then
         for I in 1 .. Natural'Value (K & "") loop
            Incrementor.Increment;
         end loop;
      end if;
   end loop;
end;
```

Manejo de interrupciones

Las interrupciones son manejadas por un procedimiento protegido sin parámetros.

```
------------------------------------------------------------------
-- Interrupt Counting Package --
------------------------------------------------------------------
with Ada.Interrupts.Names; use Ada.Interrupts.Names;

package Ctl_C_Handling is

   protected CTL_C_Handler is
      procedure Handle_Int with
        Interrupt_Handler,
        Attach_Handler => SIGINT;
      entry Wait_For_Int;
   private
      Pending_Int_Count : Natural := 0;
   end Ctl_C_Handler;

   task CTL_Reporter is
      entry Stop;
   end CTL_Reporter;

end Ctl_C_Handling;
```

El cuerpo del paquete muestra cómo funciona el procedimiento protegido. En este caso, no se utiliza un valor booleano en el objeto protegido porque las interrupciones llegan más rápido de lo que se manejan. La tarea CTL_Reporter maneja las interrupciones recibidas.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ctl_C_Handling; use CTL_C_Handling;
with Ada.Calendar; use Ada.Calendar;

package body Ctl_C_Handling is

   -------------------
   -- CTL_C_Handler --
   -------------------

   protected body CTL_C_Handler is

      ----------------
      -- Handle_Int --
      ----------------

      procedure Handle_Int is
```

```
      begin
         Pending_Int_Count := Pending_Int_Count + 1;
      end Handle_Int;

      ------------------
      -- Wait_For_Int --
      ------------------

      entry Wait_For_Int when Pending_Int_Count > 0 is
      begin
         Pending_Int_Count := Pending_Int_Count - 1;
      end Wait_For_Int;

   end CTL_C_Handler;

   ------------------
   -- CTL_Reporter --
   ------------------

   task body CTL_Reporter is
      type Second_Bin is mod 10;
      type History is array(Second_Bin) of Natural;

      ---------------------
      -- Display_History --
      ---------------------

      procedure Display_History(Item : History) is
         Sum : Natural := 0;
      begin
         for I in Item'Range loop
            Put_Line("Second: " & Second_Bin'Image(I) & " :" & Natural'Image(Item(I)));
            Sum := Sum + Item(I);
         end loop;
         Put_Line("Total count: " & Natural'Image(Sum));
         New_Line(2);
      end Display_History;

      One_Second_Count : Natural := 0;
      Next_Slot : Second_Bin := 0;
      Next_Second : Time := Clock + 1.0;
      Ten_Second_History : History := (Others => 0);

   begin
      loop
         Select
            Accept Stop;
            exit;
         else
            select
               CTL_C_Handler.Wait_For_Int;
               One_Second_Count := One_Second_Count + 1;
            or
               delay until Next_Second;
               Next_Second := Next_Second + 1.0;
               Ten_Second_History(Next_Slot) := One_Second_Count;
               Display_History(Ten_Second_History);
               Next_Slot := Next_Slot + 1;
               One_Second_Count := 0;
            end Select;
         end Select;
```

```
      end loop;
   end CTL_Reporter;
end Ctl_C_Handling;
```

Un ejemplo del programa principal para ejercitar este paquete es:

```
------------------------------------------------------------------
-- Ada2012 Interrupt Handler Example --
------------------------------------------------------------------
with Ada.Text_IO; use Ada.Text_IO;
with Ctl_C_Handling; use CTL_C_Handling;

procedure Interrupt01 is
begin
   Delay 40.0;
   CTL_Reporter.Stop;
   Put_Line("Program ended.");
end Interrupt01;
```

Lea Tarea en línea: https://riptutorial.com/es/ada/topic/7345/tarea

Capítulo 11: Tipos escalares

Introducción

En la jerarquía de tipos de Ada, los tipos elementales tienen conjuntos de valores lógicamente indivisibles. Entre estos tipos se encuentran los tipos de acceso (tipos de puntero) y los tipos escalares. Los tipos escalares se pueden categorizar como *enumeración* , *carácter* y *numérico* . Estos tipos forman el tema de este tema. Además de los conjuntos de valores, los tipos tienen conjuntos de operaciones aplicables a los escalares respectivos, como *sucesor* o `"+"` .

Sintaxis

1. **tipo** ... **es** ...

Parámetros

Elipsis	Qué
... (1)	para recibir el nombre del tipo
... (2)	para recibir las características del tipo usando palabras clave: **delta** , **dígitos** , **rango**

Observaciones

Todas las definiciones de tipo escalar, excepto la enumeración y los enteros modulares, pueden incluir una restricción de **rango** .

Una restricción de rango especifica un límite inferior y un límite superior del conjunto de valores para incluir en el tipo. Para los tipos de puntos fijos, la especificación de un rango es obligatoria: los valores de estos tipos se entenderán como múltiplos de una pequeña fracción de dos, por ejemplo, de $1/2^5$. Cuanto más pequeñas se vuelven estas fracciones, más precisa es la representación, al costo del rango que se puede representar utilizando los bits disponibles.

Se pueden dar otros aspectos de las definiciones de tipo, como un `Size` deseado en bits y otros elementos de representación. Ada 2012 agrega aspectos de la programación basada en contratos como `Static_Predicate` .

Examples

Enumeración

```
type Fruit is (Banana, Orange, Pear);
```

```
Choice : Fruit := Banana;
```

Un tipo de carácter es una enumeración que incluye un literal de carácter:

```
type Roman_Numeral is
    ('I', 'V', 'X', 'L', 'C', 'D', 'M', Unknown);`
```

Entero Singado

```
type Grade is range 0 .. 15;

B   : Grade := 11;
C   : Grade := 8;
Avg : Grade := (B + C) / 2;  -- Avg = 9
```

Entero modular

Estos son los tipos de "violín de bits". También tienen operadores lógicos, como **xor** , y se "envuelven" en el límite superior, de nuevo a 0.

```
type Bits is mod 2**24;

L : Bits := 2#00001000_01010000_11001100# or 7;
```

Punto flotante

Un tipo de punto flotante se caracteriza por sus dígitos (decimales) que establecen la precisión mínima solicitada.

```
type Distance is digits 8;

Earth : Distance := 40_075.017;
```

Punto fijo (ordinario)

Una definición de tipo de punto fijo especifica un *delta* y un rango. Juntos, describen la precisión con la que se deben aproximar los valores reales, ya que están representados por potencias de dos, sin utilizar hardware de punto flotante.

```
Shoe_Ounce : constant := 2.54 / 64.0;
type Thickness is delta Shoe_Ounce range 0.00 .. 1.00;

Strop : Thickness := 0.1;  -- could actually be 0.09375
```

Punto fijo (decimal)

Los tipos de punto fijo decimal se utilizan normalmente en la contabilidad. Se caracterizan por un

delta y un número de dígitos decimales. Sus operaciones aritméticas reflejan las reglas contables.

```
type Money is delta 0.001 digits 10;

Oil_Price : Money := 56.402;
Loss      : Money := 0.002 / 3; -- is 0.000
```

Lea Tipos escalares en línea: https://riptutorial.com/es/ada/topic/9297/tipos-escalares

Capítulo 12: Tipos parametrizados

Introducción

Todos los tipos compuestos que no sean matrices pueden tener discriminantes, que son componentes con propiedades especiales. Los discriminantes pueden ser de un tipo discreto o un tipo de acceso. En este último caso, el tipo de acceso puede ser un tipo de acceso con nombre o puede ser anónimo. Un discriminante de un tipo de acceso anónimo se denomina discriminante de acceso por analogía con un parámetro de acceso.

Examples

Tipos de registros discriminados

En el caso de un tipo de registro discriminado, algunos de los componentes se conocen como discriminantes y los componentes restantes pueden depender de estos. Se puede pensar que los discriminantes parametrizan el tipo y la sintaxis revela esta analogía. En este ejemplo, creamos un tipo que proporciona una matriz cuadrada con un parámetro positivo:

```
type Square(X: Positive) is
    record
        S: Matrix(1 .. X, 1 .. X);
    end record;
```

Luego, para crear un cuadrado de 3 por 3, simplemente llámalo de la siguiente forma:

```
Sq: Square(3);
```

Variante de estructuras de registro

Un discriminante de un tipo de registro puede influir en la estructura de los objetos. Puede existir una elección de componentes en un objeto, ya que un discriminante tenía un valor particular cuando se creó el objeto. Para admitir esta variación, la definición de un tipo de registro incluye una distinción por casos que depende del discriminante:

```
type Fruit is (Banana, Orange, Pear);

type Basket (Kind : Fruit) is
   record
      case Kind is
         when Banana =>
            Bunch_Size      : Positive;
            Bunches_Per_Box : Natural;
         when Pear | Orange =>
            Fruits_Per_Box  : Natural;
      end case;
   end record;
```

A continuación, para crear una caja de plátanos,

```
Box : Basket (Banana);
```

El objeto `Box` ahora tiene dos componentes de registro además de su discriminante, `Kind` , a saber, `Bunch_Size` y `Bunches_Per_Box` .

Lea Tipos parametrizados en línea: https://riptutorial.com/es/ada/topic/9311/tipos-parametrizados

Creditos

S. No	Capítulos	Contributors
1	Empezando con ada	B98, Community, Jacob Sparre Andersen, Jaken Herman, Jossi , manuBriot, trashgod
2	Archivos y flujos de E / S	B98, Jossi
3	Enumeración	B98, Jossi, Simon Wright
4	Genericidad en Ada	Aznhar, B98
5	Imagen de atributo	B98, Jacob Sparre Andersen, Jossi
6	Implementando el patrón productor-consumidor.	Jacob Sparre Andersen, Jim Rogers, Jossi
7	Números de salida	B98
8	paquete Ada.Text_IO	Jossi
9	Paquetes	B98, Jaken Herman, Jossi
10	Tarea	Jim Rogers, Jossi
11	Tipos escalares	B98
12	Tipos parametrizados	Aznhar, B98