EBook Gratis

APRENDIZAJE
C Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#c

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con el lenguaje C ... 2

Observaciones ... 2

Compiladores comunes ... 2

Compilador C versión soporte ... 2

Estilo de código (fuera de tema aquí): ... 3

Las bibliotecas y las API no están cubiertas por el Estándar C (y, por lo tanto, están fue ... 4

Versiones ... 4

Examples ... 4

Hola Mundo ... 4

Hola C ... 4

Veamos este programa simple línea por línea. ... 4

Editando el programa ... 5

Compilando y ejecutando el programa. ... 6

Compilar utilizando GCC ... 6

Usando el compilador clang ... 6

Usando el compilador de Microsoft C desde la línea de comando ... 6

Ejecutando el programa ... 7

Original "¡Hola mundo!" en K&R C ... 7

Capítulo 2: - clasificación de caracteres y conversión ... 9

Examples ... 9

Clasificación de caracteres leídos de una secuencia ... 9

Clasificar caracteres de una cadena ... 9

Introducción ... 10

Capítulo 3: Acolchado y embalaje de estructuras ... 13

Introducción ... 13

Observaciones ... 13

Examples ... 13

Estructuras de embalaje ... 13

Estructura de embalaje 14

Acolchado de estructura 14

Capítulo 4: Afirmación 16

Introducción 16

Sintaxis 16

Parámetros 16

Observaciones 16

Examples 17

Precondición y postcondicionamiento. 17

Aserción simple 18

Afirmación estática 18

Afirmación de código inalcanzable 19

Afirmar mensajes de error 20

Capítulo 5: Aliasing y tipo efectivo. 22

Observaciones 22

Examples 23

No se puede acceder a los tipos de caracteres a través de tipos que no son de caracteres. 23

Tipo efectivo 24

Violando las estrictas reglas de alias. 25

restringir calificación 25

Cambiando bytes 26

Capítulo 6: Ámbito identificador 28

Examples 28

Alcance del bloque 28

Función de prototipo de alcance 28

Alcance del archivo 29

Alcance de la función 30

Capítulo 7: Archivos y flujos de E / S 32

Sintaxis 32

Parámetros 32

Observaciones 32

Cadenas de modo: 32

Examples 33
Abrir y escribir al archivo 33
fprintf 34
Proceso de ejecución 35
Obtener líneas de un archivo usando getline () 35
Ejemplo de archivo de example.txt 36
Salida 36
Ejemplo de implementación de getline() 37
Abrir y escribir en un archivo binario 39
fscanf () 40
Leer líneas de un archivo 41
Capítulo 8: Argumentos de línea de comando 44
Sintaxis 44
Parámetros 44
Observaciones 44
Examples 45
Imprimiendo los argumentos de la línea de comando 45
Imprima los argumentos a un programa y conviértalos a valores enteros. 46
Usando las herramientas getopt de GNU 46
Capítulo 9: Argumentos variables 50
Introducción 50
Sintaxis 50
Parámetros 50
Observaciones 51
Examples 51
Usando un argumento de conteo explícito para determinar la longitud de la lista va 51
Usando valores terminadores para determinar el final de va_list 52
Implementando funciones con una interfaz similar a `printf ()` 53
Usando una cadena de formato 55
Capítulo 10: Arrays 58
Introducción 58
Sintaxis 58

Observaciones ... 58
Examples ... 59
Declarar e inicializar una matriz ... 59
Borrar el contenido de la matriz (puesta a cero) ... 60
Longitud de la matriz ... 61
Configuración de valores en matrices ... 62
Definir matriz y elemento de matriz de acceso. ... 63
Asignar y cero inicializar una matriz con el tamaño definido por el usuario ... 64
Iterando a través de una matriz de manera eficiente y orden de fila mayor ... 64
Matrices multidimensionales ... 66
Iterando a través de una matriz utilizando punteros ... 69
Pasar matrices multidimensionales a una función. ... 69
Ver también ... 70
Capítulo 11: Atomística ... 71
Sintaxis ... 71
Observaciones ... 71
Examples ... 72
atómicos y operadores ... 72
Capítulo 12: Booleano ... 73
Observaciones ... 73
Examples ... 73
Utilizando stdbool.h ... 73
Usando #define ... 73
Usando el tipo intrínseco (incorporado) _Bool ... 74
Enteros y punteros en expresiones booleanas. ... 74
Definiendo un tipo bool usando typedef ... 75
Capítulo 13: Calificadores de tipo ... 77
Observaciones ... 77
Cualificaciones de primer nivel ... 77
Cualificaciones de subtipo de puntero ... 77
Examples ... 78
Variables no modificables (const) ... 78

Advertencia 78

Variables volátiles 79

Capítulo 14: Campos de bits 81

Introducción 81

Sintaxis 81

Parámetros 81

Observaciones 81

Examples 81

Campos de bits 81

Usando campos de bits como enteros pequeños 83

Alineación de campo de bits 83

¿Cuándo son útiles los campos de bits? 84

No hacer para campos de bits 85

Capítulo 15: Clases de almacenamiento 87

Introducción 87

Sintaxis 87

Observaciones 87

Duración del almacenamiento 89

Duración del almacenamiento estático 89

Duración de almacenamiento de hilo 89

Duración del almacenamiento automático 89

Enlace externo e interno 90

Examples 90

typedef 90

auto 90

estático 91

externo 92

registro 93

_Hilo_local 94

Capítulo 16: Comentarios 96

Introducción 96

Sintaxis 96

Examples 96

/ * * / delimitado comentarios 96

// comentarios delimitados 97

Comentando utilizando el preprocesador 97

Posible trampa debido a los trigrafos 98

Capítulo 17: Compilacion 99

Introducción 99

Observaciones 99

Examples 101

El enlazador 101

Invocación implícita del enlazador 101

Invocación explícita del enlazador 101

Opciones para el enlazador. 102

Otras opciones de compilación 102

Tipos de archivo 102

El preprocesador 104

El compilador 106

Las fases de traducción 107

Capítulo 18: Comportamiento definido por la implementación 108

Observaciones 108

Visión general 108

Programas y Procesadores 108

General 108

Fuente de traducción 109

Entorno operativo 109

Los tipos 110

Forma de fuente 111

Evaluación 111

Comportamiento en tiempo de ejecución 112

Preprocesador 112

Biblioteca estándar 113

General 113
Funciones de entorno de punto flotante 113
Funciones relacionadas con la localización 113
Funciones matematicas 113
Señales 114
Diverso 114
Funciones de manejo de archivos. 114
Funciones de E / S 115
Funciones de asignación de memoria 115
Funciones del entorno del sistema 115
Funciones de fecha y hora. 116
Funciones de E / S de caracteres anchos 116
Examples 116
Desplazamiento a la derecha de un entero negativo 116
Asignar un valor fuera de rango a un entero 117
Asignación de cero bytes 117
Representación de enteros con signo 117
Capítulo 19: Comportamiento indefinido 118
Introducción 118
Observaciones 118
Examples 120
Desreferenciación de un puntero nulo 120
Modificar cualquier objeto más de una vez entre dos puntos de secuencia. 120
Falta la declaración de retorno en la función de retorno de valor 121
Desbordamiento de entero firmado 122
Uso de una variable sin inicializar. 123
Desreferenciación de un puntero a una variable más allá de su vida útil 124
División por cero 125
Accediendo a la memoria más allá del trozo asignado 126
Copiando memoria superpuesta 126
Lectura de un objeto sin inicializar que no está respaldado por la memoria 127
Carrera de datos 128
Leer el valor del puntero que fue liberado. 129

Modificar cadena literal.....129
Liberar la memoria dos veces.....130
Usando un especificador de formato incorrecto en printf.....130
La conversión entre tipos de puntero produce un resultado alineado incorrectamente.....131
Suma o resta del puntero no correctamente delimitada......131
Modificar una variable const mediante un puntero.....132
Pasar un puntero nulo a printf% s conversión.....132
Enlace inconsistente de identificadores.....133
Usando fflush en un flujo de entrada.....134
Desplazamiento de bits utilizando recuentos negativos o más allá del ancho del tipo.....134
Modificación de la cadena devuelta por las funciones getenv, strerror y setlocale.....135
Volviendo de una función declarada con el especificador de función `_Noreturn` o` noreturn.....135
Capítulo 20: Comunicación entre procesos (IPC).....138
Introducción.....138
Examples.....138
Semáforos.....138
Ejemplo 1.1: Carreras con hilos.....139
Ejemplo 1.2: Evita competir con semáforos.....140
Capítulo 21: Conversiones implícitas y explícitas.....143
Sintaxis.....143
Observaciones.....143
Examples.....143
Conversiones enteras en llamadas a funciones.....143
Conversiones de puntero en llamadas de función.....144
Capítulo 22: Crear e incluir archivos de encabezado.....146
Introducción.....146
Examples.....146
Introducción.....146
Idempotencia.....147
Guardias de cabecera.....147
La #pragma once Directiva.....147
Autocontención.....148

Recomendación: Los archivos de encabezado deben ser autocontenidos. 148
Reglas historicas 148
Reglas modernas 148
Comprobando la autocontención. 149
Minimalidad 150
Incluye lo que usas (IWYU) 150
Notación y Miscelánea 151
Referencias cruzadas 152
Capítulo 23: Declaración vs Definición 153
Observaciones 153
Examples 153
Entendiendo la Declaración y la Definición 153
Capítulo 24: Declaraciones 155
Observaciones 155
Examples 155
Llamando a una función desde otro archivo C 155
Usando una variable global 156
Uso de constantes globales 157
Introducción 159
Typedef 162
Uso de la regla derecha-izquierda o espiral para descifrar la declaración de C 162
Capítulo 25: Declaraciones de selección 167
Examples 167
if () Declaraciones 167
if () ... else sentencias y sintaxis 167
Switch () Declaraciones 168
if () ... else Ladder Chaining dos o más if () ... else sentencias 170
Anidado si () ... else VS if () .. else Ladder 171
Capítulo 26: Efectos secundarios 173
Examples 173
Operadores Pre / Post Incremento / Decremento 173
Capítulo 27: En línea 175

Examples 175

Funciones de alineación utilizadas en más de un archivo fuente 175

C Principal: 175

fuente1.c: 175

source2.c: 175

headerfile.h: 176

Capítulo 28: Entrada / salida formateada 178

Examples 178

Impresión del valor de un puntero a un objeto 178

Usando <inttypes.h> y uintptr_t 178

Historia Pre-Estándar: 179

Impresión de la diferencia de los valores de dos punteros a un objeto 179

Especificadores de conversión para la impresión 180

La función printf () 182

Modificadores de longitud 182

Formato de impresión de banderas 184

Capítulo 29: Enumeraciones 186

Observaciones 186

Examples 186

Enumeración simple 186

Ejemplo 1 186

Ejemplo 2 187

Typedef enum 187

Enumeración con valor duplicado 189

constante de enumeración sin nombre tipográfico 189

Capítulo 30: Errores comunes 191

Introducción 191

Examples 191

Mezclar enteros con signo y sin signo en operaciones aritméticas 191

Escribir erróneamente = en lugar de == al comparar 191

Uso incauto de punto y coma 193

Olvidarse de asignar un byte extra para \ 0 193

Olvidando liberar memoria (fugas de memoria) ... 194
Copiando demasiado ... 196
Olvidando copiar el valor de retorno de realloc en un temporal ... 196
Comparando números de punto flotante ... 196
Haciendo escala adicional en aritmética de punteros. ... 198
Las macros son simples reemplazos de cadena ... 199
Errores de referencia indefinidos al enlazar ... 201
Malentendido decaimiento de la matriz ... 203
Pasar matrices no adyacentes a funciones que esperan matrices multidimensionales "reales" ... 206
Usando constantes de caracteres en lugar de cadenas literales, y viceversa ... 207
Ignorar los valores de retorno de las funciones de la biblioteca ... 208
El carácter de nueva línea no se consume en una llamada a scanf () típica ... 209
Añadiendo un punto y coma a un #define ... 210
Los comentarios multilínea no pueden ser anidados ... 211
Superando los límites de la matriz ... 212
Función recursiva - perdiendo la condición base ... 213
Comprobando la expresión lógica contra 'verdadero' ... 215
Los literales de punto flotante son de tipo doble por defecto. ... 216
Capítulo 31: Estructuras ... 217
Introducción ... 217
Examples ... 217
Estructuras de datos simples. ... 217
Estructuras Typedef ... 217
Punteros a estructuras ... 219
Miembros de la matriz flexible ... 221
Declaración de tipo ... 221
Efectos sobre el tamaño y el relleno ... 222
Uso ... 222
La 'estructura hack' ... 223
Compatibilidad ... 223
Pasando estructuras a funciones ... 224
Programación basada en objetos utilizando estructuras. ... 225

Capítulo 32: Generación de números aleatorios 228
Observaciones 228
Examples 228
Generación de números aleatorios básicos 228
Generador Congruente Permutado 229
Restringir la generación a un rango dado 230
Generación Xorshift 230
Capítulo 33: Gestión de la memoria 232
Introducción 232
Sintaxis 232
Parámetros 232
Observaciones 232
Examples 233
Liberando memoria 233
Asignación de memoria 234
Asignación estándar 234
Memoria de cero 235
Memoria alineada 236
Reasignación de memoria 236
Arreglos multidimensionales de tamaño variable. 237
realloc (ptr, 0) no es equivalente a free (ptr) 238
Gestión de memoria definida por el usuario 239
alloca: asignar memoria en la pila 240
Resumen 241
Recomendación 241
Capítulo 34: Hilos (nativos) 242
Sintaxis 242
Observaciones 242
Las bibliotecas de C que se sabe que admiten subprocesos C11 son: 242
Bibliotecas C que no admiten subprocesos C11, sin embargo: 242
Examples 243
Iniciar varios hilos 243

Inicialización por un hilo 243

Capítulo 35: Inicialización 245

Examples 245

Inicialización de variables en C 245

Inicializando estructuras y matrices de estructuras. 247

Usando inicializadores designados 247

Inicializadores designados para elementos de matriz 247

Inicializadores designados para estructuras. 248

Inicializador designado para uniones 248

Inicializadores designados para matrices de estructuras, etc. 249

Especificando rangos en inicializadores de matriz 249

Capítulo 36: Instrumentos de cuerda 251

Introducción 251

Sintaxis 251

Examples 251

Calcular la longitud: strlen () 251

Copia y concatenación: strcpy (), strcat () 252

Comparsion: strcmp (), strncmp (), strcasecmp (), strncasecmp () 253

Tokenización: strtok (), strtok_r () y strtok_s () 255

Encuentre la primera / última aparición de un carácter específico: strchr (), strrchr () 257

Iterando sobre los personajes en una cadena 259

Introducción básica a las cuerdas. 259

Creando matrices de cuerdas 260

strstr 261

Literales de cuerda 262

Poniendo a cero una cuerda 263

strspn y strcspn 264

Copiando cuerdas 265

Las asignaciones de punteros no copian cadenas 265

Copiando cadenas utilizando funciones estándar 266

strcpy() 266

snprintf() 266

strncat() 266

strncpy() 267

Convierta cadenas a números: atoi (), atof () (peligroso, no las use) 268

cadena de datos con formato de lectura / escritura 269

Convertir con seguridad cadenas a números: funciones strtoX 270

Capítulo 37: Iteraciones / bucles de repetición: for, while, do-while 272

Sintaxis 272

Observaciones 272

Declaración de iteración controlada por la cabeza / Bucles 272

Declaración de iteración controlada por el pie / bucles 272

Examples 272

En bucle 272

Mientras bucle 273

Bucle Do-While 273

Estructura y flujo de control en un bucle for. 274

Bucles infinitos 275

Loop desenrollado y dispositivo de Duff 276

Capítulo 38: Lenguajes comunes de programación C y prácticas de desarrollador. 278

Examples 278

Comparando literal y variable 278

No deje en blanco la lista de parámetros de una función - use void 278

Capítulo 39: Listas enlazadas 282

Observaciones 282

Lista enlazada individualmente 282

Estructura de datos 282

Lista doblemente enlazada 282

Estructura de datos 282

Topoliges 282

Lineal o abierto 282

Circular o anillo 283

Procedimientos 283

Enlazar 283

Hacer una lista enlazada circularmente 283

Hacer una lista enlazada linealmente 284

Inserción 284

Examples 285

Insertar un nodo al comienzo de una lista enlazada individualmente 285

Explicación para la inserción de nodos 286

Insertando un nodo en la posición n. 287

Invertir una lista enlazada 288

Explicación para el método de lista inversa 289

Una lista doblemente enlazada. 290

Capítulo 40: Literales compuestos 294

Sintaxis 294

Observaciones 294

Examples 294

Definición / Inicialización de Literales Compuestos 294

Ejemplos de la norma C, C11-§6.5.2.5 / 9: 294

Compuesto literal con designadores. 295

Literal compuesto sin especificar la longitud de la matriz 295

Literal compuesto que tiene una longitud de inicializador menor que el tamaño de matriz es 296

Literal compuesto de solo lectura 296

Literal compuesto que contiene expresiones arbitrarias 296

Capítulo 41: Literales para números, caracteres y cadenas. 297

Observaciones 297

Examples 297

Literales enteros 297

Literales de cuerda 298

Literales de punto flotante 298

Literales de personajes 299

Capítulo 42: Los operadores 301

Introducción 301

Sintaxis 301
Observaciones 301
Examples 303
Operadores relacionales 303
Es igual a "==" 303
No es igual a "! =" 303
No "!" 304
Mayor que ">" 304
Menos que "<" 304
Mayor o igual que "> =" 304
Menor o igual que "<=" 305
Operadores de Asignación 305
Operadores aritméticos 306
Aritmética básica 306
Operador de adición 307
Operador de resta 307
Operador de multiplicación 307
Operador de la división 308
Operador Modulo 308
Operadores de Incremento / Decremento 309
Operadores logicos 309
Y lógico 309
O lógico 310
Lógica NO 310
Evaluación de corto circuito 310
Incremento / Decremento 311
Operador Condicional / Operador Ternario 311
Operador de coma 312
Operador de Reparto 313
operador de tamaño 313
Con un tipo como operando. 313
Con una expresión como operando. 313

Aritmética de puntero 313
Además de puntero 314
Resta de puntero 315
Operadores de Acceso 315
Miembro de objeto 315
Miembro de objeto apuntado 315
Dirección de 316
Desreferencia 316
Indexación 316
Intercambiabilidad de la indexación. 316
Operador de llamada de función 317
Operadores de Bitwise 317
_ Alineación de 319
Comportamiento a corto circuito de operadores lógicos. 319
Capítulo 43: Macros x 322
Introducción 322
Observaciones 322
Examples 322
Uso trivial de X-macros para printfs 322
Enum valor e identificador 323
Extensión: Da la macro X como argumento 323
Codigo de GENERACION 324
Aquí usamos X-macros para declarar una enumeración que contiene 4 comandos y un mapa de su 324
De manera similar, podemos generar una tabla de salto para llamar a funciones mediante el 325
Capítulo 44: Manejo de errores 326
Sintaxis 326
Observaciones 326
Examples 326
errno 326
estridente 326
perror 327
Capítulo 45: Manejo de señales 328

Sintaxis 328

Parámetros 328

Observaciones 328

Examples 329

Manejo de señales con “señal ()” 329

Capítulo 46: Marcos de prueba 332

Introducción 332

Observaciones 332

Examples 332

CppUTest 332

Unity Test Framework 333

CMocka 334

Capítulo 47: Matemáticas estándar 336

Sintaxis 336

Observaciones 336

Examples 336

Resto de punto flotante de doble precisión: fmod () 336

Resto de punto flotante de precisión simple y doble precisión: fmodf (), fmodl () 337

Funciones de energía: pow (), powf (), powl () 338

Capítulo 48: Montaje en línea 340

Observaciones 340

Pros 340

Contras 340

Examples 340

Soporte básico de asc gcc 340

Soporte de asm gcc extendido 341

Gcc Inline Assembly en macros 342

Capítulo 49: Multihilo 344

Introducción 344

Sintaxis 344

Observaciones 344

Examples 344

C11 hilos ejemplo simple 344

Capítulo 50: Parámetros de función 346

Observaciones 346

Examples 346

Usando parámetros de puntero para devolver múltiples valores 346

Pasando en Arrays a Funciones 346

Ver también 347

Los parámetros se pasan por valor 347

Orden de ejecución de parámetros de función 348

Ejemplo de función que devuelve la estructura que contiene valores con códigos de error 348

Capítulo 51: Pasar arrays 2D a funciones 351

Examples 351

Pasar una matriz 2D a una función 351

Usando matrices planas como matrices 2D 357

Capítulo 52: Preprocesador y Macros 359

Introducción 359

Observaciones 359

Examples 359

Inclusión condicional y modificación de la firma de la función condicional. 359

Inclusión de archivo fuente 362

Reemplazo de macros 362

Directiva de error 363

#if 0 para bloquear secciones de código 363

Pegado de ficha 364

Macros predefinidas 365

Macros obligatorias predefinidas 365

Otras macros predefinidas (no obligatorio) 366

El encabezado incluye guardias 367

Implementación FOREACH 370

__cplusplus para usar C externos en el código C ++ compilado con C ++ - denominación de no 373

Macros similares a funciones 374

Argumentos variables macro 375

Capítulo 53: Punteros 378

Introducción 378

Sintaxis 378

Observaciones 378

Examples 378

Errores comunes 378

No revisar fallas en la asignación 379

Usando números literales en lugar de sizeof al solicitar memoria 379

Pérdidas de memoria 379

Errores logicos 380

Creando punteros para apilar variables. 380

Incremento / decremento y desreferenciación 381

Desreferenciación de un puntero 382

Desreferenciación de un puntero a una estructura 382

Punteros de funcion 383

Ver también 385

Inicializando los punteros 385

Precaución: 385

Precaución: 386

Dirección de Operador (&) 386

Aritmética de puntero 386

Void * punteros como argumentos y valores de retorno a las funciones estándar 386

Punteros const 387

Punteros individuales 387

Puntero a puntero 388

Mismo Asterisco, Diferentes Significados 390

Premisa 390

Ejemplo 390

Conclusión 391

Puntero a puntero 391

Introducción 392

Punteros y matrices 394

Comportamiento polimórfico con punteros vacíos. 394

Capítulo 54: Punteros a funciones 396

Introducción 396

Sintaxis 396

Examples 396

Asignar un puntero de función 396

Devolviendo punteros a funciones desde una función 397

Mejores prácticas 397

Usando typedef 397

Ejemplo: 398

Tomando punteros de contexto. 398

Ejemplo 399

Ver también 399

Introducción 399

Uso 400

Sintaxis 400

Mnemónicos para escribir punteros a funciones. 400

Lo esencial 401

Capítulo 55: Puntos de secuencia 403

Observaciones 403

Examples 403

Expresiones secuenciadas 404

Expresiones sin secuencia 404

Expresiones en secuencia indeterminada 405

Capítulo 56: Restricciones 407

Observaciones 407

Examples 407

Duplicar nombres de variables en el mismo ámbito. 407

Operadores aritméticos únicos 408

Capítulo 57: Saltar declaraciones 409
Sintaxis 409
Observaciones 409
Ver también 409
Examples 409
Usando goto para saltar fuera de los bucles anidados 409
Usando el retorno 410
Devolviendo un valor 410
Devolviendo nada 411
Usando break y continue 411
Capítulo 58: Secuencia de caracteres de múltiples caracteres 413
Observaciones 413
Examples 413
Trigrafos 413
Digraphs 414
Capítulo 59: Selección genérica 416
Sintaxis 416
Parámetros 416
Observaciones 416
Examples 416
Compruebe si una variable es de un determinado tipo calificado 416
Tipo genérico de impresión macro 417
Selección genérica basada en múltiples argumentos. 417
Capítulo 60: Tipos de datos 420
Observaciones 420
Examples 420
Tipos enteros y constantes 420
Literales de cuerda 422
Tipos de enteros de ancho fijo (desde C99) 423
Constantes de punto flotante 423
Interpretando declaraciones 424
Ejemplos 425

Declaraciones multiples ... 425
Interpretación alternativa ... 426
Capítulo 61: Typedef ... 427
Introducción ... 427
Sintaxis ... 427
Observaciones ... 427
Examples ... 428
Typedef para Estructuras y Uniones ... 428
Usos Simples de Typedef ... 429
Para dar nombres cortos a un tipo de datos ... 429
Mejora de la portabilidad ... 429
Especificar un uso o mejorar la legibilidad. ... 430
Typedef para punteros de función ... 430
Capítulo 62: Uniones ... 433
Examples ... 433
Diferencia entre estructura y unión. ... 433
Uso de uniones para reinterpretar valores. ... 433
Escribir a un miembro del sindicato y leer de otro. ... 434
Capítulo 63: Valgrind ... 436
Sintaxis ... 436
Observaciones ... 436
Examples ... 436
Corriendo valgrind ... 436
Añadiendo banderas ... 436
Bytes perdidos - Olvidando liberar ... 436
Errores más comunes encontrados al usar Valgrind ... 437
Creditos ... 439

Acerca de

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

It is an unofficial and free C Language 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 Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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 el lenguaje C

Observaciones

C es un lenguaje de programación de computadora imperativo, de propósito general, que admite programación estructurada, alcance de variables léxicas y recursión, mientras que un sistema de tipo estático evita muchas operaciones no intencionadas. Por diseño, C proporciona construcciones que se asignan de manera eficiente a las instrucciones típicas de la máquina y, por lo tanto, ha encontrado un uso duradero en aplicaciones que anteriormente se habían codificado en lenguaje ensamblador, incluidos los sistemas operativos, así como diversos programas de aplicación para computadoras que van desde supercomputadoras a sistemas integrados. .

A pesar de sus capacidades de bajo nivel, el lenguaje fue diseñado para fomentar la programación multiplataforma. Se puede compilar un programa C de escritura portátil y compatible con los estándares para una gran variedad de plataformas informáticas y sistemas operativos con pocos cambios en su código fuente. El lenguaje está disponible en una amplia gama de plataformas, desde microcontroladores integrados hasta supercomputadoras.

C fue desarrollado originalmente por Dennis Ritchie entre 1969 y 1973 en Bell Labs y se usó para volver a implementar los sistemas operativos Unix. Desde entonces, se ha convertido en uno de los lenguajes de programación más utilizados de todos los tiempos, con compiladores C de varios proveedores disponibles para la mayoría de las arquitecturas de computadoras y sistemas operativos existentes.

Compiladores comunes

El proceso para compilar un programa en C difiere entre compiladores y sistemas operativos. La mayoría de los sistemas operativos se envían sin compilador, por lo que tendrá que instalar uno. Algunas opciones comunes de compiladores son:

- GCC, la colección de compiladores de GNU
- clang: un front-end de la familia del lenguaje C para LLVM
- MSVC, herramientas de compilación de Microsoft Visual C / C ++

Los siguientes documentos le darán una buena descripción general de cómo comenzar a usar algunos de los compiladores más comunes:

- Empezando con Microsoft Visual C
- Empezando con GCC

Compilador C versión soporte

Tenga en cuenta que los compiladores tienen diferentes niveles de compatibilidad con el estándar C y muchos aún no son completamente compatibles con C99. Por ejemplo, a partir de la versión

de 2015, MSVC admite gran parte de C99, pero todavía tiene algunas excepciones importantes para el soporte del lenguaje en sí (por ejemplo, el preprocesamiento parece no conforme) y para la biblioteca C (por ejemplo, `<tgmath.h>`), ni ¿necesariamente documentan sus "opciones dependientes de la implementación"? Wikipedia tiene una tabla que muestra el soporte ofrecido por algunos compiladores populares.

Algunos compiladores (en particular GCC) han ofrecido, o continúan ofreciendo, *extensiones de compilador* que implementan características adicionales que los productores del compilador consideran necesarias, útiles o creen que pueden formar parte de una versión C futura, pero que actualmente no forman parte de ningún estándar C. Como estas extensiones son específicas del compilador, se puede considerar que no son compatibles entre sí y los desarrolladores del compilador pueden eliminarlas o alterarlas en versiones posteriores del compilador. El uso de tales extensiones generalmente puede ser controlado por banderas del compilador.

Además, muchos desarrolladores tienen compiladores que admiten solo versiones específicas de C impuestas por el entorno o la plataforma a la que se dirigen.

Si selecciona un compilador, se recomienda elegir un compilador que tenga el mejor soporte para la última versión de C permitida para el entorno de destino.

Estilo de código (fuera de tema aquí):

Debido a que el espacio en blanco es insignificante en C (es decir, no afecta el funcionamiento del código), los programadores a menudo usan espacios en blanco para que el código sea más fácil de leer y comprender, esto se denomina *estilo de código* . Es un conjunto de reglas y pautas que se utilizan al escribir el código fuente. Cubre inquietudes tales como cómo se deben sangrar las líneas, si se deben usar los espacios o tabulaciones, cómo se deben colocar los frenos, cómo se deben usar los espacios alrededor de los operadores y los corchetes, cómo deben nombrarse las variables, etc.

El estilo de código no está cubierto por el estándar y se basa principalmente en la opinión (diferentes personas encuentran que los diferentes estilos son más fáciles de leer), como tal, generalmente se considera fuera de tema en SO. El consejo primordial sobre el estilo en el propio código es que la consistencia es primordial: elija o haga un estilo y apéguese a él. Basta con explicar que hay varios estilos con nombre en el uso común que a menudo son elegidos por los programadores en lugar de crear su propio estilo.

Algunos estilos de sangría comunes son: estilo K & R, estilo Allman, estilo GNU, etc. Algunos de estos estilos tienen diferentes variantes. Allman, por ejemplo, se usa como Allman regular o como la variante popular, Allman-8. La información sobre algunos de los estilos populares se puede encontrar en Wikipedia . Dichos nombres de estilo se toman de los estándares que los autores u organizaciones publican a menudo para el uso de las muchas personas que contribuyen a su código, de modo que todos puedan leer fácilmente el código cuando conocen el estilo, como la guía de formato GNU que forma parte de El documento de normas de codificación GNU .

Algunas convenciones de nomenclatura comunes son: UpperCamelCase, lowerCamelCase, lower_case_with_underscore, ALL_CAPS, etc. Estos estilos se combinan de varias maneras para

usar con diferentes tipos y objetos (por ejemplo, las macros a menudo usan el estilo ALL_CAPS)

El estilo K & R generalmente se recomienda para su uso dentro de la documentación de SO, mientras que los estilos más esotéricos, como Pico, no se recomiendan.

Las bibliotecas y las API no están cubiertas por el Estándar C (y, por lo tanto, están fuera de tema aquí):

- API POSIX (que abarca, por ejemplo, PThreads , sockets , señales)

Versiones

Versión	Estándar	Fecha de publicación
K&R	n / A	1978-02-22
C89	ANSI X3.159-1989	1989-12-14
C90	ISO / IEC 9899: 1990	1990-12-20
C95	ISO / IEC 9899 / AMD1: 1995	1995-03-30
C99	ISO / IEC 9899: 1999	1999-12-16
C11	ISO / IEC 9899: 2011	2011-12-15

Examples

Hola Mundo

Para crear un programa en C simple que imprima *"Hola, Mundo"* en la pantalla, use un editor de texto para crear un nuevo archivo (por ejemplo, `hello.c` - la extensión del archivo debe ser `.c`) que contenga el siguiente código fuente:

Hola C

```
#include <stdio.h>

int main(void)
{
    puts("Hello, World");
    return 0;
}
```

Demo en vivo en Coliru

Veamos este programa simple línea por línea.

```
#include <stdio.h>
```

Esta línea le dice al compilador que incluya el contenido del archivo de encabezado de la biblioteca estándar `stdio.h` en el programa. Los encabezados son generalmente archivos que contienen declaraciones de funciones, macros y tipos de datos, y debe incluir el archivo de encabezado antes de usarlos. Esta línea incluye `stdio.h` para que pueda llamar a la función `puts()` .

Ver más sobre los encabezados.

```
int main(void)
```

Esta línea comienza la definición de una función. Indica el nombre de la función (`main`), el tipo y el número de argumentos que espera (`void` , lo que significa ninguno) y el tipo de valor que devuelve esta función (`int`). La ejecución del programa se inicia en la función `main()` .

```
{
    …
}
```

Las llaves se utilizan en pares para indicar dónde comienza y termina un bloque de código. Se pueden usar de muchas maneras, pero en este caso indican dónde comienza y termina la función.

```
    puts("Hello, World");
```

Esta línea llama a la función `puts()` para enviar el texto a la salida estándar (la pantalla, por defecto), seguida de una nueva línea. La cadena a emitir se incluye dentro de los paréntesis.

`"Hello, World"` es la cadena que se escribirá en la pantalla. En C, cada valor literal de cadena debe estar dentro de las comillas dobles `"…"` .

Ver más sobre cuerdas.

En los programas de C, cada declaración debe terminar con un punto y coma (es decir `;`).

```
    return 0;
```

Cuando definimos `main()` , lo declaramos como una función que devuelve un `int` , lo que significa que debe devolver un entero. En este ejemplo, estamos devolviendo el valor entero 0, que se utiliza para indicar que el programa salió correctamente. Después de la `return 0;` declaración, el proceso de ejecución terminará.

Editando el programa

Los editores de texto simples incluyen `vim` o `gedit` en Linux o `Notepad` de `Notepad` en Windows. Los editores multiplataforma también incluyen `Visual Studio Code` o `Sublime Text` .

El editor debe crear archivos de texto plano, no RTF u otro formato.

Compilando y ejecutando el programa.

Para ejecutar el programa, este archivo fuente (`hello.c`) primero debe compilarse en un archivo ejecutable (por ejemplo, `hello` en el sistema Unix / Linux o `hello.exe` en Windows). Esto se hace usando un compilador para el lenguaje C.

Ver más sobre compilar.

Compilar utilizando GCC

GCC (GNU Compiler Collection) es un compilador de C ampliamente utilizado. Para usarlo, abra un terminal, use la línea de comando para navegar a la ubicación del archivo fuente y luego ejecute:

```
gcc hello.c -o hello
```

Si no se encuentran errores en el código fuente (`hello.c`), el compilador creará un **archivo binario** , cuyo nombre viene dado por el argumento a la opción de línea de comando `-o` (`hello`). Este es el archivo ejecutable final.

También podemos usar las opciones de advertencia `-Wall -Wextra -Werror` , que ayudan a identificar problemas que pueden hacer que el programa falle o produzcan resultados inesperados. No son necesarios para este programa simple, pero esta es una forma de agregarlos:

```
gcc -Wall -Wextra -Werror -o hello hello.c
```

Usando el compilador clang

Para compilar el programa usando `clang` puedes usar:

```
clang -Wall -Wextra -Werror -o hello hello.c
```

Por diseño, las opciones de la línea de comando `clang` son similares a las de GCC.

Usando el compilador de Microsoft C desde la línea de

comando

Si utiliza el compilador `cl.exe` Microsoft en un sistema Windows que admite Visual Studio y todas las variables de entorno están configuradas, este ejemplo de C se puede compilar con el siguiente comando que producirá un `hello.exe` ejecutable dentro del directorio en el que se ejecuta el comando (Hay opciones de advertencia como `/W3` para `cl` , aproximadamente análogas a `-Wall` etc. para GCC o clang).

```
cl hello.c
```

Ejecutando el programa

Una vez compilado, el archivo binario puede ejecutarse escribiendo `./hello` en el terminal. Una vez ejecutado, el programa compilado imprimirá `Hello, World` , seguido de una nueva línea, en el símbolo del sistema.

Original "¡Hola mundo!" en K&R C

El siguiente es el original "¡Hola mundo!" programa del libro The C Programming Language de Brian Kernighan y Dennis Ritchie (Ritchie fue el desarrollador original del lenguaje de programación C en Bell Labs), conocido como "K&R":

K&R

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Observe que el lenguaje de programación C no estaba estandarizado al momento de escribir la primera edición de este libro (1978), y que este programa probablemente no se compilará en la mayoría de los compiladores modernos a menos que se les indique que acepten el código C90.

Este primer ejemplo en el libro de K&R ahora se considera de baja calidad, en parte porque carece de un tipo de retorno explícito para `main()` y en parte porque carece de una declaración de `return` . La 2ª edición del libro fue escrita para el antiguo estándar C89. En C89, el tipo de `main` defecto sería `int` , pero el ejemplo de K&R no devuelve un valor definido al entorno. En C99 y estándares posteriores, se requiere el tipo de devolución, pero es seguro omitir la declaración de `return` de `main` (y solo `main`), debido a un caso especial introducido con C99 5.1.2.2.3: es equivalente a devolver 0 , lo que indica éxito.

La forma recomendada y más portátil de `main` para sistemas alojados es `int main (void)` cuando el programa no usa argumentos de línea de comando, o `int main(int argc, char **argv)` cuando el programa sí usa los argumentos de línea de comando.

C90 §5.1.2.2.3 **Terminación del programa**

> Un retorno de la llamada inicial a la función `main` es equivalente a llamar a la función de `exit` con el valor devuelto por la función `main` como su argumento. Si la función `main` ejecuta una devolución que no especifica ningún valor, el estado de terminación devuelto al entorno del host no está definido.

C90 §6.6.6.4 **La declaración de `return`**

> Si se ejecuta una declaración de `return` sin una expresión, y el llamante utiliza el valor de la llamada a la función, el comportamiento no está definido. Llegar al `}` que termina una función es equivalente a ejecutar una declaración de `return` sin una expresión.

C99 §5.1.2.2.3 **Terminación del programa**

> Si el tipo de retorno de la función `main` es un tipo compatible con `int` , un retorno de la llamada inicial a la función `main` es equivalente a llamar a la función de `exit` con el valor devuelto por la función `main` como su argumento; alcanzar el `}` que termina la función `main` devuelve un valor de 0. Si el tipo de retorno no es compatible con `int` , el estado de terminación devuelto al entorno de host no está especificado.

Lea Empezando con el lenguaje C en línea: https://riptutorial.com/es/c/topic/213/empezando-con-el-lenguaje-c

Capítulo 2: - clasificación de caracteres y conversión

Examples

Clasificación de caracteres leídos de una secuencia

```
#include <ctype.h>
#include <stdio.h>

typedef struct {
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(FILE *f) {
  chartypes types = { 0, 0, 0 };
  int ch;

  while ((ch = fgetc(f)) != EOF) {
    types.space += !!isspace(ch);
    types.alnum += !!isalnum(ch);
    types.punct += !!ispunct(ch);
  }

  return types;
}
```

La función de `classify` lee caracteres de una secuencia y cuenta el número de espacios, caracteres alfanuméricos y puntuación. Evita varias trampas.

- Cuando se lee un carácter de un flujo, el resultado se guarda como un `int` , ya que de lo contrario habría una ambigüedad entre la lectura de `EOF` (el marcador de fin de archivo) y un carácter que tiene el mismo patrón de bits.
- Las funciones de clasificación de caracteres (por ejemplo, `isspace`) esperan que su argumento se pueda *representar como un `unsigned char` , o el valor de la macro `EOF`* . Dado que esto es exactamente lo que devuelve el `fgetc` , no hay necesidad de conversión aquí.
- El valor de retorno de las funciones de clasificación de caracteres solo distingue entre cero (que significa `false`) y distinto de cero (que significa `true`). Para contar el número de ocurrencias, este valor se debe convertir a 1 o 0, lo que se hace con la negación doble, `!!` .

Clasificar caracteres de una cadena

```
#include <ctype.h>
#include <stddef.h>

typedef struct {
  size_t space;
```

```
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(const char *s) {
  chartypes types = { 0, 0, 0 };
  const char *p;
  for (p= s; p != '\0'; p++) {
    types.space += !!isspace((unsigned char)*p);
    types.alnum += !!isalnum((unsigned char)*p);
    types.punct += !!ispunct((unsigned char)*p);
  }

  return types;
}
```

La función de `classify` examina todos los caracteres de una cadena y cuenta el número de espacios, caracteres alfanuméricos y signos de puntuación. Evita varias trampas.

- Las funciones de clasificación de caracteres (por ejemplo, `isspace`) esperan que su argumento se pueda *representar como un `unsigned char` , o el valor de la macro `EOF`* .
- La expresión `*p` es de tipo `char` y, por lo tanto, debe convertirse para que coincida con el texto anterior.
- El tipo `char` se define como equivalente a `signed char` o `unsigned char` .
- Cuando `char` es equivalente a un `unsigned char` , no hay problema, ya que cada valor posible del tipo `char` se puede representar como un `unsigned char` .
- Cuando `char` es equivalente a `signed char` , se debe convertir a `unsigned char` antes de pasar a las funciones de clasificación de caracteres. Y aunque el valor del personaje puede cambiar debido a esta conversión, esto es exactamente lo que estas funciones esperan.
- El valor de retorno de las funciones de clasificación de caracteres solo distingue entre cero (que significa `false`) y distinto de cero (que significa `true`). Para contar el número de ocurrencias, este valor se debe convertir a 1 o 0, lo que se hace con la negación doble, `!!` .

Introducción

El encabezado `ctype.h` es una parte de la biblioteca C estándar. Proporciona funciones para clasificar y convertir personajes.

Todas estas funciones toman un parámetro, un `int` que debe ser *EOF* o representable como un carácter sin signo.

Los nombres de las funciones de clasificación tienen el prefijo 'es'. Cada uno devuelve un valor entero no nulo (VERDADERO) si el carácter que se le pasa satisface la condición relacionada. Si la condición no se cumple, la función devuelve un valor cero (FALSO).

Estas funciones de clasificación funcionan como se muestra, asumiendo la configuración regional C predeterminada:

```
int a;
int c = 'A';
a = isalpha(c); /* Checks if c is alphabetic (A-Z, a-z), returns non-zero here. */
```

```
a = isalnum(c); /* Checks if c  is alphanumeric (A-Z, a-z, 0-9), returns non-zero here. */
a = iscntrl(c); /* Checks is c is a control character (0x00-0x1F, 0x7F), returns zero here. */
a = isdigit(c); /* Checks if c is a digit (0-9), returns zero here. */
a = isgraph(c); /* Checks if c has a graphical representation (any printing character except
space), returns non-zero here. */
a = islower(c); /* Checks if c is a lower-case letter (a-z), returns zero here. */
a = isprint(c); /* Checks if c is any printable character (including space), returns non-zero
here. */
a = isupper(c); /* Checks if c is a upper-case letter (a-z), returns zero here. */
a = ispunct(c); /* Checks if c is a punctuation character, returns zero here. */
a = isspace(c); /* Checks if c is a white-space character, returns zero here. */
a = isupper(c); /* Checks if c is an upper-case letter (A-Z), returns non-zero here. */
a = isxdigit(c); /* Checks if c is a hexadecimal digit (A-F, a-f, 0-9), returns non-zero here.
*/
```

C99

```
a = isblank(c); /* Checks if c is a blank character (space or tab), returns non-zero here. */
```

Hay dos funciones de conversión. Estos se nombran usando el prefijo 'to'. Estas funciones toman el mismo argumento que las anteriores. Sin embargo, el valor de retorno no es un simple cero o no cero, pero el argumento pasado cambió de alguna manera.

Estas funciones de conversión funcionan como se muestra, asumiendo la configuración regional C predeterminada:

```
int a;
int c = 'A';

/* Converts c to a lower-case letter (a-z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'a' here.
 */
a = tolower(c);

/* Converts c to an upper-case letter (A-Z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'A' here.
 */
a = toupper(c);
```

La información a continuación se cita en cplusplus.com, que muestra cómo el conjunto ASCII de 127 caracteres original es considerado por cada una de las funciones de tipo de clasificación (a • indica que la función no es cero para ese carácter)

Valores ASCII	caracteres	iscntrl	está en blanco	espacio de emisión	isupper	es bajo	isalfa	isigi
0x00 .. 0x08	NUL, (otros códigos de control)	•						
0x09	pestaña ('\ t')	•	•	•				

Valores ASCII	caracteres	iscntrl	está en blanco	espacio de emisión	isupper	es bajo	isalfa	isigi
0x0A .. 0x0D	(códigos de control de espacios en blanco: '\ f', '\ v', '\ n', '\ r')	•		•				
0x0E .. 0x1F	(otros códigos de control)	•						
0x20	espacio (' ')		•	•				
0x21 .. 0x2F	! "# $% & '() * +, -. /							
0x30 .. 0x39	0123456789							•
0x3a .. 0x40	:; <=>? @							
0x41 .. 0x46	A B C D E F				•		•	
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ				•		•	
0x5B .. 0x60	[] ^ _ `							
0x61 .. 0x66	a B C D e F					•	•	
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					•	•	
0x7B .. 0x7E	{} ~ bar							
0x7F	(DEL)	•						

Lea - clasificación de caracteres y conversión en línea: https://riptutorial.com/es/c/topic/6846/-ctype-h----clasificacion-de-caracteres-y-conversion

Capítulo 3: Acolchado y embalaje de estructuras

Introducción

De forma predeterminada, los compiladores de C diseñan estructuras para que se pueda acceder rápidamente a cada miembro, sin incurrir en penalizaciones para el acceso no alineado, un problema con máquinas RISC como el DEC Alpha y algunas CPU ARM.

Dependiendo de la arquitectura de la CPU y el compilador, una estructura puede ocupar más espacio en la memoria que la suma de los tamaños de sus miembros componentes. El compilador puede agregar relleno entre los miembros o al final de la estructura, pero no al principio.

El embalaje anula el relleno predeterminado.

Observaciones

Eric Raymond tiene un artículo en The Lost Art of C Structure Packing que es útil para leer.

Examples

Estructuras de embalaje

Por defecto, las estructuras se rellenan en C. Si desea evitar este comportamiento, debe solicitarlo explícitamente. Bajo GCC es `__attribute__((__packed__))` . Considere este ejemplo en una máquina de 64 bits:

```
struct foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

La estructura se rellenará automáticamente para tener `8-byte` alineación de `8-byte` y se verá así:

```
struct foo {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte  */

    char pad[7]; /* 7 bytes added by compiler */

    long x;      /* 8 bytes */
};
```

Entonces `sizeof(struct foo)` nos dará `24` lugar de `17` . Esto sucedió debido a un compilador de 64

bits de lectura / escritura de / a Memoria en 8 bytes de palabra en cada paso y es obvio cuando se intenta escribir el `char c;` un byte en la memoria, un total de 8 bytes (es decir, palabra) recuperado y consume solo el primer byte de la misma, y sus siete bytes sucesivos permanecen vacíos y no son accesibles para ninguna operación de lectura y escritura para el relleno de la estructura.

Estructura de embalaje

Pero si agrega el atributo `packed` , el compilador no agregará relleno:

```
struct __attribute__((__packed__)) foo {
    char *p;   /* 8 bytes */
    char c;    /* 1 byte  */
    long x;    /* 8 bytes */
};
```

Ahora `sizeof(struct foo)` devolverá `17` .

Generalmente se utilizan estructuras compactas:

- Para ahorrar espacio.
- Para formatear una estructura de datos para transmitir a través de la red sin depender de cada alineación de la arquitectura de cada nodo de la red.

Se debe tener en cuenta que algunos procesadores como el ARM Cortex-M0 no permiten el acceso a la memoria no alineada; en tales casos, el empaquetado de la estructura puede llevar a *un comportamiento indefinido* y puede bloquear la CPU.

Acolchado de estructura

Supongamos que esta `struct` está definida y compilada con un compilador de 32 bits:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

Podríamos esperar que esta `struct` ocupe solo 10 bytes de memoria, pero al imprimir `sizeof(str_32)` vemos que usa 12 bytes.

Esto sucedió porque el compilador alinea las variables para un acceso rápido. Un patrón común es que cuando el tipo de base ocupa N bytes (donde N es una potencia de 2 como 1, 2, 4, 8, 16 y rara vez más grande), la variable debe alinearse en un límite de N bytes (un múltiplo de N bytes).

Para la estructura mostrada con `sizeof(int) == 4` y `sizeof(short) == 2` , un diseño común es:

- `int a;` almacenado en offset 0; talla 4.
- `short b;` almacenado en el desplazamiento 4; talla 2

- relleno sin nombre en el desplazamiento 6; talla 2
- `int c;` almacenado en el desplazamiento 8; talla 4.

Así, `struct test_32` ocupa 12 bytes de memoria. En este ejemplo, no hay relleno final.

El compilador se asegurará de que cualquier variable `struct test_32` se almacene comenzando en un límite de 4 bytes, de modo que los miembros dentro de la estructura se alinearán correctamente para un acceso rápido. Se requieren funciones de asignación de memoria como `malloc()` , `calloc()` y `realloc()` para garantizar que el puntero devuelto esté lo suficientemente bien alineado para su uso con cualquier tipo de datos, por lo que las estructuras asignadas dinámicamente también se alinearán correctamente.

Puede terminar con situaciones extrañas, como en un procesador Intel x86_64 de 64 bits (por ejemplo, Intel Core i7 - una Mac que ejecuta macOS Sierra o Mac OS X), donde al compilar en modo de 32 bits, los compiladores colocan `double` alineación en un Límite de 4 bytes; pero, en el mismo hardware, al compilar en modo de 64 bits, los compiladores colocan `double` alineación en un límite de 8 bytes.

Lea Acolchado y embalaje de estructuras en línea:
https://riptutorial.com/es/c/topic/4590/acolchado-y-embalaje-de-estructuras

Capítulo 4: Afirmación

Introducción

Una ***aserción*** es un predicado de que la condición presentada debe ser verdadera en el momento en que el software encuentra la aserción. Las más comunes son ***las aserciones simples*** , que se validan en el momento de la ejecución. Sin embargo, las ***afirmaciones estáticas*** se verifican en el momento de la compilación.

Sintaxis

- afirmar (expresión)
- static_assert (expresión, mensaje)
- _Static_assert (expresión, mensaje)

Parámetros

Parámetro	Detalles
expresión	Expresión de tipo escalar.
mensaje	cadena literal que se incluirá en el mensaje de diagnóstico.

Observaciones

Tanto `assert` como `static_assert` son macros definidas en `assert.h` .

La definición de `assert` depende de la macro `NDEBUG` que no se define por la biblioteca estándar. Si se define `NDEBUG` , `assert` es un no-op:

```
#ifdef NDEBUG
#  define assert(condition) ((void) 0)
#else
#  define assert(condition) /* implementation defined */
#endif
```

La opinión varía sobre si `NDEBUG` siempre debe usarse para compilaciones de producción.

- El pro-campo argumenta que `assert` llamadas `abort` y los mensajes de aserción no son útiles para los usuarios finales, por lo que el resultado no es útil para el usuario. Si tiene condiciones fatales para verificar el código de producción, debe usar las condiciones ordinarias `if/else` y `exit` o `quick_exit` para finalizar el programa. En contraste con la `abort` , estos permiten que el programa realice una limpieza (a través de las funciones registradas con `atexit` o `at_quick_exit`).

- La con-campamento sostiene que `assert` llamadas nunca deben disparar en el código de producción, pero si lo hacen, la condición que se comprueba significa que hay algo radicalmente equivocado y el programa se comportará mal peor si la ejecución continúa. Por lo tanto, es mejor tener las afirmaciones activas en el código de producción porque si se disparan, el infierno ya se ha desatado.
- Otra opción es usar un sistema casero de aserciones que siempre realice la verificación pero que maneje los errores de manera diferente entre el desarrollo (cuando sea apropiado `abort`) y la producción (donde un "error interno inesperado (contacte al Soporte Técnico" puede ser más apropiado).

`static_assert` expande a `_Static_assert` que es una palabra clave. La condición se verifica en el momento de la compilación, por lo que la `condition` debe ser una expresión constante. No hay necesidad de que esto se maneje de manera diferente entre el desarrollo y la producción.

Examples

Precondición y postcondicionamiento.

Un caso de uso para la afirmación es la condición previa y la condición posterior. Esto puede ser muy útil para mantener invariable y diseño por contrato . Para un ejemplo, una longitud siempre es cero o positiva, por lo que esta función debe devolver un valor cero o positivo.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* Precondition: */
    /* NULL is an invalid vector */
    assert (a != NULL);
    /* Number of dimensions can not be negative.*/
    assert (count >= 0);

    /* Calculation */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* Postcondition: */
    /* Resulting length can not be negative. */
    assert (result >= 0);
    return result;
}

#define COUNT 3

int main (void)
{
    int a[COUNT] = {1, 2, 3};
```

```
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);
    printf ("r = %i\n", r);
    r = length2 (b, COUNT);
    printf ("r = %i\n", r);
    return 0;
}
```

Aserción simple

Una afirmación es una declaración utilizada para afirmar que un hecho debe ser verdadero cuando se alcanza esa línea de código. Las afirmaciones son útiles para asegurar que se cumplan las condiciones esperadas. Cuando la condición pasada a una aserción es verdadera, no hay acción. El comportamiento en condiciones falsas depende de las banderas del compilador. Cuando las aserciones están habilitadas, una entrada falsa provoca una detención inmediata del programa. Cuando están desactivados, no se realiza ninguna acción. Es una práctica común habilitar aserciones en compilaciones internas y de depuración, y deshabilitarlas en compilaciones de versión, aunque las aserciones a menudo se habilitan en la versión. (El hecho de que la terminación sea mejor o peor que los errores depende del programa). Las afirmaciones se deben usar solo para detectar errores de programación internos, lo que generalmente significa que se pasan parámetros incorrectos.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);

    printf("x = %d\n", x);
    return 0;
}
```

Posible salida con `NDEBUG` indefinido:

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

Posible salida con `NDEBUG` definido:

```
x = -1
```

Es una buena práctica definir `NDEBUG` globalmente, para que pueda compilar fácilmente su código con todas las aserciones `NDEBUG` o desactivadas. Una forma fácil de hacer esto es definir `NDEBUG` como una opción para el compilador, o definirlo en un encabezado de configuración compartido (por ejemplo, `config.h`).

Afirmación estática

C11

Las aserciones estáticas se utilizan para verificar si una condición es verdadera cuando se compila el código. Si no es así, el compilador debe emitir un mensaje de error y detener el proceso de compilación.

Una aserción estática se verifica en el momento de la compilación, no en el tiempo de ejecución. La condición debe ser una expresión constante, y si es falso, se producirá un error del compilador. El primer argumento, la condición que se comprueba, debe ser una expresión constante, y el segundo es un literal de cadena.

A diferencia de aseverar, `_Static_assert` es una palabra clave. Una macro de conveniencia `static_assert` se define en `<assert.h>` .

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10");  /* compiler error */
```

C99

Antes de C11, no había soporte directo para afirmaciones estáticas. Sin embargo, en C99, las afirmaciones estáticas se pueden emular con macros que desencadenan un error de compilación si la condición de tiempo de compilación es falsa. A diferencia de `_Static_assert` , el segundo parámetro debe ser un nombre de token adecuado para que se pueda crear un nombre de variable con él. Si la afirmación falla, el nombre de la variable se ve en el error del compilador, ya que esa variable se usó en una declaración de matriz sintácticamente incorrecta.

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##__##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* compile error */
```

Antes de C99, no podía declarar variables en ubicaciones arbitrarias en un bloque, por lo que tendría que ser extremadamente cuidadoso al usar esta macro, asegurándose de que solo aparezca donde una declaración de variable sería válida.

Afirmación de código inalcanzable

Durante el desarrollo, cuando ciertas rutas de código deben evitarse del alcance del flujo de control, puede usar `assert(0)` para indicar que tal condición es errónea:

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;
```

```
    default:
        assert(0);
}
```

Cuando el argumento de la macro `assert()` evalúa como falso, la macro escribirá información de diagnóstico en la secuencia de error estándar y luego abortará el programa. Esta información incluye el archivo y el número de línea de la declaración `assert()` y puede ser muy útil para la depuración. Las afirmaciones se pueden desactivar definiendo la macro `NDEBUG` .

Otra forma de terminar un programa cuando se produce un error es con las funciones de biblioteca estándar `exit` , `quick_exit` o `abort` . `exit` y `quick_exit` toman un argumento que puede ser devuelto a su entorno. `abort()` (y, por lo tanto, `assert`) puede ser una terminación realmente severa de su programa, y ciertas limpiezas que de otra forma se realizarían al final de la ejecución, pueden no realizarse.

La principal ventaja de `assert()` es que imprime automáticamente la información de depuración. Llamar a `abort()` tiene la ventaja de que no se puede deshabilitar como una aserción, pero puede que no muestre ninguna información de depuración. En algunas situaciones, usar ambas construcciones juntas puede ser beneficioso:

```
if (color == COLOR_RED || color == COLOR_GREEN) {
   ...
} else if (color == COLOR_BLUE) {
   ...
} else {
   assert(0), abort();
}
```

Cuando las `assert()` están *habilitadas* , la llamada a `assert()` imprimirá información de depuración y terminará el programa. La ejecución nunca llega a la llamada `abort()` . Cuando las `assert()` están *deshabilitadas* , la llamada `assert()` no hace nada y se llama `abort()` . Esto asegura que el programa *siempre* termina por esta condición de error; habilitar y deshabilitar las afirmaciones solo afecta si se imprime o no la salida de depuración.

Nunca se debe dejar un tal `assert` en el código de producción, debido a que la información de depuración no es útil para los usuarios finales y debido a `abort` es generalmente una terminación demasiado severa que inhiben los manipuladores de limpieza que se instalan para `exit` o `quick_exit` para funcionar.

Afirmar mensajes de error

Existe un truco que puede mostrar un mensaje de error junto con una afirmación. Normalmente, escribirías código como este

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

Si la afirmación falla, un mensaje de error sería similar

> Fallo en la afirmación: p! = NULL, archivo main.c, línea 5

Sin embargo, también puede usar AND (`&&`) lógico para dar un mensaje de error

```
void f(void *p)
{
    assert(p != NULL && "function f: p cannot be NULL");
    /* more code */
}
```

Ahora, si la afirmación falla, un mensaje de error leerá algo como esto

> Falló la aserción: p! = NULL && "la función f: p no puede ser NULL", archivo main.c, línea 5

La razón por la que esto funciona es que un literal de cadena siempre se evalúa como distinto de cero (verdadero). Agregar `&& 1` a una expresión booleana no tiene efecto. Por lo tanto, agregar `&& "error message"` tampoco tiene ningún efecto, excepto que el compilador mostrará la expresión completa que falló.

Lea Afirmación en línea: https://riptutorial.com/es/c/topic/555/afirmacion

Capítulo 5: Aliasing y tipo efectivo.

Observaciones

Las violaciones de las reglas de creación de alias y de violar el tipo efectivo de un objeto son dos cosas diferentes y no deben confundirse.

- *El aliasing* es la propiedad de dos punteros `a` y `b` que se refieren al mismo objeto, es decir, `a == b` .
- C utiliza el *tipo efectivo* de un objeto de datos para determinar qué operaciones se pueden realizar en ese objeto. En particular, el tipo efectivo se utiliza para determinar si dos punteros pueden alias entre sí.

El aliasing puede ser un problema para la optimización, porque cambiar el objeto a través de un puntero, `a` ejemplo, puede cambiar el objeto que es visible a través del otro puntero, `b` . Si su compilador de C tuviera que asumir que los punteros siempre se podrían aliar entre sí, independientemente de su tipo y procedencia, se perderían muchas oportunidades de optimización y muchos programas se ejecutarían más lentamente.

Las reglas estrictas de alias de C se refieren a los casos en el compilador que *puede asumir* qué objetos se hacen (o no) alias entre sí. Hay dos reglas básicas que siempre debe tener en cuenta para los punteros de datos.

> A menos que se indique lo contrario, dos punteros con el mismo tipo de base pueden ser alias.
>
> Dos punteros con un tipo de base diferente no pueden ser alias, a menos que al menos uno de los dos tipos sea un tipo de carácter.

Aquí *, tipo base* significa que dejamos de lado las calificaciones de tipo como `const` , por ejemplo, si `a` es `double*` y `b` es `const double*` , el compilador generalmente *debe* asumir que un cambio de `*a` puede cambiar `*b` .

Violar la segunda regla puede tener resultados catastróficos. En este caso, violar la regla de alias estricta significa que presenta dos punteros `a` y `b` de diferente tipo al compilador que en realidad apuntan al mismo objeto. Entonces, el compilador siempre puede suponer que los dos apuntan a objetos diferentes, y no actualizará su idea de `*b` si cambia el objeto a través de `*a` .

Si lo haces, el comportamiento de tu programa se vuelve indefinido. Por lo tanto, C pone restricciones muy severas en las conversiones de punteros para ayudarlo a evitar que esa situación ocurra accidentalmente.

> A menos que el tipo de origen o destino sea `void` , todas las conversiones de punteros entre punteros con un tipo de base diferente deben ser *explícitas* .

O en otras palabras, necesitan un *molde,* a menos que haga una conversión que sólo añade un

calificativo como `const` al tipo de destino.

Evitar las conversiones de punteros en general y las conversiones en particular lo protege de los problemas de aliasing. A menos que realmente los necesite, y estos casos son muy especiales, debe evitarlos como pueda.

Examples

No se puede acceder a los tipos de caracteres a través de tipos que no son de caracteres.

Si un objeto se define con una duración de almacenamiento estático, de subproceso o automático y tiene un tipo de carácter, ya sea: `char` , `unsigned char` , o `signed char` , no se puede acceder a él por un tipo que no sea de carácter. En el ejemplo siguiente un `char` array se reinterpreta como el tipo `int` , y el comportamiento es indefinido en cada dereference de la `int` puntero `b` .

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

    _Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

Esto no está definido porque viola la regla del "tipo efectivo", no se puede acceder a ningún objeto de datos que tenga un tipo efectivo a través de otro tipo que no sea un tipo de carácter. Dado que el otro tipo aquí es `int` , esto no está permitido.

Incluso si se sabe que la alineación y el tamaño de los punteros se ajustan, esto no estaría exento de esta regla, el comportamiento aún sería indefinido.

Esto significa, en particular, que no hay forma en el estándar C de reservar un objeto de tipo de carácter de búfer que pueda usarse a través de punteros con diferentes tipos, ya que usaría un búfer recibido por `malloc` o una función similar.

Una forma correcta de lograr el mismo objetivo que en el ejemplo anterior sería utilizar una `union` .

```
typedef union bufType bufType;
union bufType {
   char c[sizeof(int[25])];
   int i[25];
};

int main( void )
{
```

```
    bufType a = { .c = { 0 } }; // reserve a buffer and initialize
    int* b = a.i;      // no cast necessary
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

    _Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

Aquí, la `union` garantiza que el compilador sepa desde el principio que se puede acceder al búfer a través de diferentes vistas. Esto también tiene la ventaja de que ahora el búfer tiene una "vista" `ai` que ya es de tipo `int` y no se necesita conversión de puntero.

Tipo efectivo

El *tipo efectivo* de un objeto de datos es el último tipo de información que se asoció con él, en su caso.

```
// a normal variable, effective type uint32_t, and this type never changes
uint32_t a = 0.0;

// effective type of *pa is uint32_t, too, simply
// because *pa is the object a
uint32_t* pa = &a;

// the object pointed to by q has no effective type, yet
void* q = malloc(sizeof uint32_t);
// the object pointed to by q still has no effective type,
// because nobody has written to it
uint32_t* qb = q;
// *qb now has effective type uint32_t because a uint32_t value was written
*qb = 37;

// the object pointed to by r has no effective type, yet, although
// it is initialized
void* r = calloc(1, sizeof uint32_t);
// the object pointed to by r still has no effective type,
// because nobody has written to or read from it
uint32_t* rc = r;
// *rc now has effective type uint32_t because a value is read
// from it with that type. The read operation is valid because we used calloc.
// Now the object pointed to by r (which is the same as *rc) has
// gained an effective type, although we didn't change its value.
uint32_t c = *rc;

// the object pointed to by s has no effective type, yet.
void* s = malloc(sizeof uint32_t);
// the object pointed to by s now has effective type uint32_t
// because an uint32_t value is copied into it.
memcpy(s, r, sizeof uint32_t);
```

Observe que para este último, no fue necesario que tengamos un puntero `uint32_t*` para ese objeto. El hecho de que hayamos copiado otro objeto `uint32_t` es suficiente.

Violando las estrictas reglas de alias.

En el siguiente código, asumamos por simplicidad que `float` y `uint32_t` tienen el mismo tamaño.

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g should equal %g\n", a, b);
}
```

`u` y `f` tienen un tipo de base diferente, y por lo tanto el compilador puede asumir que apuntan a objetos diferentes. No hay posibilidad de que `*f` haya cambiado entre las dos inicializaciones de `a` y `b` , por lo que el compilador puede optimizar el código a algo equivalente a

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g should equal %g\n", a, a);
}
```

Es decir, la segunda operación de carga de `*f` puede optimizarse completamente.

Si llamamos a esta función "normalmente"

```
float fval = 4;
uint32_t uval = 77;
fun(&uval, &fval);
```

todo va bien y algo así

> 4 debería ser igual a 4

está impreso. Pero si hacemos trampa y pasamos el mismo puntero, después de convertirlo,

```
float fval = 4;
uint32_t* up = (uint32_t*)&fval;
fun(up, &fval);
```

violamos la estricta regla de aliasing. Entonces el comportamiento se vuelve indefinido. La salida podría ser la anterior, si el compilador hubiera optimizado el segundo acceso, o algo completamente diferente, y así su programa termine en un estado completamente no confiable.

restringir calificación

Si tenemos dos argumentos de puntero del mismo tipo, el compilador no puede hacer ninguna suposición y siempre tendremos que asumir que el cambio a `*e` puede cambiar `*f` :

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
```

```
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}

float fval = 4;
float eval = 77;
 fun(&eval, &fval);
```

todo va bien y algo así

> ¿Es 4 igual a 4?

está impreso. Si pasamos el mismo puntero, el programa seguirá haciendo lo correcto e imprimirá

> ¿Es 4 igual a 22?

Esto puede resultar ineficiente, si *sabemos* por alguna información externa que `e` y `f` nunca apuntarán al mismo objeto de datos. Podemos reflejar ese conocimiento al agregar calificadores de `restrict` a los parámetros del puntero:

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}
```

Entonces, el compilador siempre puede suponer que `e` y `f` apuntan a objetos diferentes.

Cambiando bytes

Una vez que un objeto tiene un tipo efectivo, no debe intentar modificarlo a través de un puntero de otro tipo, a menos que ese otro tipo sea un tipo de carácter, `char` , `signed char` o `unsigned char` .

```
#include <inttypes.h>
#include <stdio.h>

int main(void) {
  uint32_t a = 57;
  // conversion from incompatible types needs a cast !
  unsigned char* ap = (unsigned char*)&a;
  for (size_t i = 0; i < sizeof a; ++i) {
    /* set each byte of a to 42 */
    ap[i] = 42;
  }
  printf("a now has value %" PRIu32 "\n", a);
}
```

Este es un programa válido que imprime

> un ahora tiene valor 707406378

Esto funciona porque:

- El acceso se realiza a los bytes individuales que se ven con el tipo `unsigned char` para que cada modificación esté bien definida.
- Las dos vistas al objeto, a través de `a` y a través de `*ap` , alias, pero como `ap` es un puntero a un tipo de carácter, la regla de alias estricta no se aplica. Por lo tanto, el compilador debe asumir que el valor de `a` puede haber sido cambiado en el bucle `for` . El valor modificado de `a` debe construirse a partir de los bytes que se han cambiado.
- El tipo de `a` , `uint32_t` no tiene bits de relleno. Todos sus bits de la representación cuentan para el valor, aquí `707406378` , y no puede haber representación de captura.

Lea Aliasing y tipo efectivo. en línea: https://riptutorial.com/es/c/topic/1301/aliasing-y-tipo-efectivo-

Capítulo 6: Ámbito identificador

Examples

Alcance del bloque

Un identificador tiene un ámbito de bloque si su declaración correspondiente aparece dentro de un bloque (se aplica la declaración de parámetros en la definición de función). El alcance finaliza al final del bloque correspondiente.

Ninguna entidad diferente con el mismo identificador puede tener el mismo alcance, pero los ámbitos pueden superponerse. En el caso de superposición de ámbitos, el único visible es el declarado en el ámbito más interno.

```
#include <stdio.h>

void test(int bar)                   // bar has scope test function block
{
    int foo = 5;                     // foo has scope test function block
    {
        int bar = 10;                // bar has scope inner block, this overlaps with previous
test:bar declaration, and it hides test:bar
        printf("%d %d\n", foo, bar); // 5 10
    }                                // end of scope for inner bar
    printf("%d %d\n", foo, bar);     // 5 5, here bar is test:bar
}                                    // end of scope for test:foo and test:bar

int main(void)
{
    int foo = 3;         // foo has scope main function block

    printf("%d\n", foo); // 3
    test(5);
    printf("%d\n", foo); // 3
    return 0;
}                        // end of scope for main:foo
```

Función de prototipo de alcance

```
#include <stdio.h>

/* The parameter name, apple, has function prototype scope.  These names
   are not significant outside the prototype itself.  This is demonstrated
   below. */

int test_function(int apple);

int main(void)
{
    int orange = 5;

    orange = test_function(orange);
    printf("%d\r\n", orange); //orange = 6
```

```
    return 0;
}

int test_function(int fruit)
{
    fruit += 1;
    return fruit;
}
```

Tenga en cuenta que recibe mensajes de error desconcertantes si introduce un nombre de tipo en un prototipo:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

Con GCC 6.3.0, este código (archivo fuente `dc11.c`) produce:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: error: 'struct whatever' declared inside parameter list will not be visible
outside of this definition or declaration [-Werror]
     int function(struct whatever *arg);
                         ^~~~~~~~
dc11.c:9:9: error: conflicting types for 'function'
     int function(struct whatever *arg)
         ^~~~~~~~
dc11.c:1:9: note: previous declaration of 'function' was here
     int function(struct whatever *arg);
         ^~~~~~~~
cc1: all warnings being treated as errors
$
```

Coloque la definición de la estructura antes de la declaración de la función, o agregue la `struct whatever;` como una línea antes de la declaración de función, y no hay problema. No debe introducir nuevos nombres de tipo en un prototipo de función porque no hay manera de usar ese tipo y, por lo tanto, no hay manera de definir o usar esa función.

Alcance del archivo

```
#include <stdio.h>

/* The identifier, foo, is declared outside all blocks.
   It can be used anywhere after the declaration until the end of
   the translation unit. */
static int foo;
```

```
void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();
    printf("%d\r\n", foo); //foo = 3;

    return 0;
}
```

Alcance de la función

El alcance de la función es el ámbito especial para las **etiquetas** . Esto se debe a su propiedad inusual. Una **etiqueta** es visible a través de toda la función que está definida y se puede saltar (usando la instrucción `goto label`) desde cualquier punto en la misma función. Si bien no es útil, el siguiente ejemplo ilustra el punto:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
  OUTSIDE:
    if (a!=0) {
        int i=0;
      INSIDE:
        printf("a=%d\n",a);
        goto OUTSIDE;
    }
}
```

`INSIDE` puede parecer definido *dentro* del bloque `if` , como es el caso para `i` cuyo alcance es el bloque, pero no lo es. Es visible en toda la función como la instrucción `goto INSIDE;` ilustra Por lo tanto, no puede haber dos etiquetas con el mismo identificador en una sola función.

Un posible uso es el siguiente patrón para realizar limpiezas complejas correctas de recursos asignados:

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
   double* a = malloc(sizeof(double[34]));
   if (!a) {
      fprintf(stderr,"can't allocate\n");
      return;                 /* No point in freeing a if it is null */
   }
   FILE* b = fopen("some_file","r");
   if (!b) {
```

```
        fprintf(stderr,"can't open\n");
        goto CLEANUP1;          /* Free a; no point in closing b */
    }
    /* do something reasonable */
    if (error) {
        fprintf(stderr,"something's wrong\n");
        goto CLEANUP2;       /* Free a and close b to prevent leaks */
    }
    /* do yet something else */
CLEANUP2:
    close(b);
CLEANUP1:
    free(a);
}
```

Las etiquetas como `CLEANUP1` y `CLEANUP2` son identificadores especiales que se comportan de manera diferente a todos los demás identificadores. Son visibles desde cualquier lugar dentro de la función, incluso en lugares que se ejecutan antes de la declaración etiquetada, o incluso en lugares a los que nunca se podría llegar si no se ejecuta ninguno de los `goto` . Las etiquetas a menudo se escriben en minúsculas en lugar de mayúsculas.

Lea Ámbito identificador en línea: https://riptutorial.com/es/c/topic/1804/ambito-identificador

Capítulo 7: Archivos y flujos de E / S

Sintaxis

- #include <stdio.h> / * Incluya esto para usar cualquiera de las siguientes secciones * /
- FILE * fopen (const char * path, const char * mode); / * Abra una secuencia en el archivo en la *ruta* con el *modo* especificado * /
- FILE * freopen (const char * path, const char * mode, FILE * stream); / * Vuelva a abrir una secuencia existente en el archivo en la *ruta* con el *modo* especificado * /
- int fclose (FILE * stream); / * Cerrar un flujo abierto * /
- size_t fread (void * ptr, size_t size, size_t nmemb, FILE * stream); / * Lea a la mayoría de los elementos *nmemb* de bytes de *tamaño* cada uno del *flujo* y escríbalos en *ptr* . Devuelve el número de elementos leídos. * /
- size_t fwrite (const void * ptr, size_t size, size_t nmemb, FILE * stream); / * Escribir *nmiemb* elementos de *size* bytes cada uno de *ptr* a la *corriente.* Devuelve el número de elementos escritos. * /
- int fseek (FILE * stream, offset largo, int whence); / * Establezca el cursor del flujo en *offset* , relativo al desplazamiento indicado por *whence* , y devuelve 0 si tuvo éxito. * /
- largo ftell (FILE * stream); / * Devuelve el desplazamiento de la posición actual del cursor desde el principio de la secuencia. * /
- rebobinado vacío (secuencia de ARCHIVO *); / * Establece la posición del cursor al principio del archivo. * /
- int fprintf (FILE * fout, const char * fmt, ...); / * Escribe la cadena de formato printf en fout * /
- ARCHIVO * stdin; / * Flujo de entrada estándar * /
- ARCHIVO * stdout; / * Flujo de salida estándar * /
- ARCHIVO * stderr; / * Corriente de error estándar * /

Parámetros

Parámetro	Detalles
modo const char *	Una cadena que describe el modo de apertura de la secuencia respaldada por archivos. Ver comentarios para los posibles valores.
de dónde viene	Puede ser `SEEK_SET` para establecer desde el principio del archivo, `SEEK_END` para establecer desde su final, o `SEEK_CUR` para establecer en relación con el valor actual del cursor. Nota: `SEEK_END` no es portátil.

Observaciones

Cadenas de modo:

Las cadenas de modo en `fopen()` y `freopen()` pueden ser uno de esos valores:

- `"r"` : abre el archivo en modo de solo lectura, con el cursor colocado al principio del archivo.
- `"r+"` : abre el archivo en modo de lectura y escritura, con el cursor ajustado al principio del archivo.
- `"w"` : abra o cree el archivo en modo de solo escritura, con su contenido truncado a 0 bytes. El cursor se establece al principio del archivo.
- `"w+"` : abra o cree el archivo en modo de lectura / escritura, con su contenido truncado a 0 bytes. El cursor se establece al principio del archivo.
- `"a"` : abre o crea el archivo en modo de solo escritura, con el cursor colocado al final del archivo.
- `"a+"` : abre o crea el archivo en modo de lectura y escritura, con el cursor de lectura configurado al principio del archivo. La salida, sin embargo, *siempre* se agregará al final del archivo.

Cada uno de estos modos de archivo puede tener una `b` agregada después de la letra inicial (por ejemplo, `"rb"` o `"a+b"` o `"ab+"`). La `b` significa que el archivo debe tratarse como un archivo binario en lugar de un archivo de texto en aquellos sistemas donde hay una diferencia. No hace una diferencia en sistemas similares a Unix; Es importante en los sistemas Windows. (Además, Windows `fopen` permite una `t` explícita en lugar de `b` para indicar 'archivo de texto' y muchas otras opciones específicas de la plataforma).

C11

- `"wx"` : crea un archivo de texto en modo de solo escritura. *El archivo puede no existir* .
- `"wbx"` : `"wbx"` un archivo binario en modo de solo escritura. *El archivo puede no existir* .

La `x` , si está presente, debe ser el último carácter en la cadena de modo.

Examples

Abrir y escribir al archivo

```
#include <stdio.h>   /* for perror(), fopen(), fputs() and fclose() */
#include <stdlib.h>  /* for the EXIT_* macros */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* Get path from argument to main else default to output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";

    /* Open file for writing and obtain file pointer */
    FILE *file = fopen(path, "w");

    /* Print error message and exit if fopen() failed */
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Writes text to file. Unlike puts(), fputs() does not add a new-line. */
```

```
    if (fputs("Output in file.\n", file) == EOF)
    {
        perror(path);
        e = EXIT_FAILURE;
    }

    /* Close file */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

Este programa abre el archivo con el nombre dado en el argumento a main, por defecto a `output.txt` si no se da ningún argumento. Si ya existe un archivo con el mismo nombre, su contenido se descarta y el archivo se trata como un nuevo archivo vacío. Si los archivos no existen, la llamada `fopen()` crea.

Si la llamada `fopen()` falla por algún motivo, devuelve un valor `NULL` y establece el valor de la variable `errno` global. Esto significa que el programa puede probar el valor devuelto después de la llamada `fopen()` y usar `perror()` si `fopen()` falla.

Si la llamada `fopen()` tiene éxito, devuelve un puntero de `FILE` válido. Este puntero se puede usar para hacer referencia a este archivo hasta que se `fclose()` .

La función `fputs()` escribe el texto dado en el archivo abierto, reemplazando cualquier contenido anterior del archivo. De manera similar a `fopen()` , la función `fputs()` también establece el valor `errno` si falla, aunque en este caso la función devuelve `EOF` para indicar el error (de lo contrario, devuelve un valor no negativo).

La función `fclose()` cualquier búfer, cierra el archivo y libera la memoria apuntada por `FILE *` . El valor de retorno indica que se completó tal como lo hace `fputs()` (aunque devuelve '0' si es exitoso), de nuevo, también establece el valor `errno` en el caso de un error.

fprintf

Puede usar `fprintf` en un archivo como lo `fprintf` en una consola con `printf` . Por ejemplo, para hacer un seguimiento de las victorias del juego, las pérdidas y los empates que pueda escribir

```
/* saves wins, losses and, ties */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
    fprintf(fout, "Wins: %d\nTies: %d\nLosses: %d\n", wins, ties, losses);
}
```

Una nota al margen: algunos sistemas (infame, Windows) no usan lo que la mayoría de los programadores llamarían finales de línea "normales". Mientras que los sistemas similares a UNIX usan \ n para terminar líneas, Windows usa un par de caracteres: \ r (retorno de carro) y \ n (avance de línea). Esta secuencia se llama comúnmente CRLF. Sin embargo, siempre que use C,

no necesita preocuparse por estos detalles altamente dependientes de la plataforma. Se requiere el compilador de CA para convertir cada instancia de \ n al final de línea de la plataforma correcta. Entonces, un compilador de Windows convertiría \ n a \ r \ n, pero un compilador de UNIX lo mantendría tal como está.

Proceso de ejecución

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}
int main(void)
{
    FILE *stream;

    /* call netstat command. netstat is available for Windows and Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

Este programa ejecuta un proceso (`netstat`) a través de `popen()` y lee toda la salida estándar del proceso y hace eco de eso a la salida estándar.

Nota: `popen()` no existe en la biblioteca estándar de C , pero es más bien una parte de POSIX C)

Obtener líneas de un archivo usando getline ()

La biblioteca POSIX C define la función `getline()` . Esta función asigna un búfer para mantener el contenido de la línea y devuelve la nueva línea, el número de caracteres en la línea y el tamaño del búfer.

Programa de ejemplo que obtiene cada línea de `example.txt` :

```
#include <stdlib.h>
#include <stdio.h>


#define FILENAME "example.txt"

int main(void)
{
  /* Open the file for reading */
  char *line_buf = NULL;
  size_t line_buf_size = 0;
  int line_count = 0;
  ssize_t line_size;
  FILE *fp = fopen(FILENAME, "r");
```

```
  if (!fp)
  {
    fprintf(stderr, "Error opening file '%s'\n", FILENAME);
    return EXIT_FAILURE;
  }

  /* Get the first line of the file. */
  line_size = getline(&line_buf, &line_buf_size, fp);

  /* Loop through until we are done with the file. */
  while (line_size >= 0)
  {
    /* Increment our line count */
    line_count++;

    /* Show the line details */
    printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
        line_size, line_buf_size, line_buf);

    /* Get the next line */
    line_size = getline(&line_buf, &line_buf_size, fp);
  }

  /* Free the allocated line buffer */
  free(line_buf);
  line_buf = NULL;

  /* Close the file now that we are done with it */
  fclose(fp);

  return EXIT_SUCCESS;
}
```

Ejemplo de archivo de `example.txt`

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines



a really long line to show that getline() will reallocate the line buffer if the length of a
line is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Salida

```
line[000001]: chars=000015, buf size=000016, contents: This is a file
line[000002]: chars=000012, buf size=000016, contents:   which has
line[000003]: chars=000015, buf size=000016, contents: multiple lines
line[000004]: chars=000030, buf size=000032, contents:     with various indentation,
line[000005]: chars=000012, buf size=000032, contents: blank lines
line[000006]: chars=000001, buf size=000032, contents:
line[000007]: chars=000001, buf size=000032, contents:
line[000008]: chars=000001, buf size=000032, contents:
```

```
line[000009]: chars=000150, buf size=000160, contents: a really long line to show that
getline() will reallocate the line buffer if the length of a line is too long to fit in the
buffer it has been given,
line[000010]: chars=000042, buf size=000160, contents:  and punctuation at the end of the
lines.
line[000011]: chars=000001, buf size=000160, contents:
```

En el ejemplo, inicialmente se llama a `getline()` sin un búfer asignado. Durante esta primera llamada, `getline()` asigna un búfer, lee la primera línea y coloca el contenido de la línea en el nuevo búfer. En las llamadas subsiguientes, `getline()` actualiza el mismo búfer y solo reasigna el búfer cuando ya no es lo suficientemente grande como para que quepa toda la línea. El búfer temporal se libera cuando terminamos con el archivo.

Otra opción es `getdelim()` . Esto es lo mismo que `getline()` excepto que especifique el carácter de final de línea. Esto solo es necesario si el último carácter de la línea para su tipo de archivo no es '\ n'. `getline()` funciona incluso con archivos de texto de Windows porque con la línea multibyte el final (`"\r\n")` '\ n'` sigue siendo el último carácter de la línea.

Ejemplo de implementación de `getline()`

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <stdint.h>


#if !(defined _POSIX_C_SOURCE)
typedef long int ssize_t;
#endif

/* Only include our version of getline() if the POSIX version isn't available. */

#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L

#if !(defined SSIZE_MAX)
#define SSIZE_MAX (SIZE_MAX >> 1)
#endif

ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)
{
  const size_t INITALLOC = 16;
  const size_t ALLOCSTEP = 16;
  size_t num_read = 0;

  /* First check that none of our input pointers are NULL. */
  if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))
  {
    errno = EINVAL;
    return -1;
  }

  /* If output buffer is NULL, then allocate a buffer. */
  if (NULL == *pline_buf)
  {
    *pline_buf = malloc(INITALLOC);
    if (NULL == *pline_buf)
```

```
  {
    /* Can't allocate memory. */
    return -1;
  }
  else
  {
    /* Note how big the buffer is at this time. */
    *pn = INITALLOC;
  }
}

/* Step through the file, pulling characters until either a newline or EOF. */

{
  int c;
  while (EOF != (c = getc(fin)))
  {
    /* Note we read a character. */
    num_read++;

    /* Reallocate the buffer if we need more room */
    if (num_read >= *pn)
    {
      size_t n_realloc = *pn + ALLOCSTEP;
      char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 for the trailing NUL. */
      if (NULL != tmp)
      {
        /* Use the new buffer and note the new buffer size. */
        *pline_buf = tmp;
        *pn = n_realloc;
      }
      else
      {
        /* Exit with error and let the caller free the buffer. */
        return -1;
      }

      /* Test for overflow. */
      if (SSIZE_MAX < *pn)
      {
        errno = ERANGE;
        return -1;
      }
    }

    /* Add the character to the buffer. */
    (*pline_buf)[num_read - 1] = (char) c;

    /* Break from the loop if we hit the ending character. */
    if (c == '\n')
    {
      break;
    }
  }

  /* Note if we hit EOF. */
  if (EOF == c)
  {
    errno = 0;
    return -1;
  }
```

```
  }

  /* Terminate the string by suffixing NUL. */
  (*pline_buf)[num_read] = '\0';

  return (ssize_t) num_read;
}

#endif
```

Abrir y escribir en un archivo binario

```
#include <stdlib.h>
#include <stdio.h>


int main(void)
{
   result = EXIT_SUCCESS;

   char file_name[] = "outbut.bin";
   char str[] = "This is a binary file example";
   FILE * fp = fopen(file_name, "wb");

   if (fp == NULL)  /* If an error occurs during the file creation */
   {
     result = EXIT_FAILURE;
     fprintf(stderr, "fopen() failed for '%s'\n", file_name);
   }
   else
   {
     size_t element_size = sizeof *str;
     size_t elements_to_write = sizeof str;

     /* Writes str (_including_ the NUL-terminator) to the binary file. */
     size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
     if (elements_written != elements_to_write)
     {
       result = EXIT_FAILURE;
       /* This works for >=c99 only, else the z length modifier is unknown. */
       fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",
         elements_written, elements_to_write);
       /* Use this for <c99: *
       fprintf(stderr, "fwrite() failed: wrote only %lu out of %lu elements.\n",
         (unsigned long) elements_written, (unsigned long) elements_to_write);
        */
     }

     fclose(fp);
   }

   return result;
}
```

Este programa crea y escribe texto en forma binaria a través de la función `fwrite` en el archivo `output.bin` .

Si ya existe un archivo con el mismo nombre, su contenido se descarta y el archivo se trata como

un nuevo archivo vacío.

Un flujo binario es una secuencia ordenada de caracteres que puede grabar datos internos de forma transparente. En este modo, los bytes se escriben entre el programa y el archivo sin ninguna interpretación.

Para escribir de forma portátil números enteros, se debe saber si el formato de archivo los espera en formato big o little-endian, y el tamaño (generalmente 16, 32 o 64 bits). El desplazamiento de bits y el enmascaramiento se pueden usar para escribir los bytes en el orden correcto. No se garantiza que los enteros en C tengan representación de complemento a dos (aunque casi todas las implementaciones sí). Afortunadamente, una conversión a unsigned *está* garantizada para utilizar dos complementos. El código para escribir un entero con signo en un archivo binario es, por lo tanto, un poco sorprendente.

```
/* write a 16-bit little endian integer */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}
```

Las otras funciones siguen el mismo patrón con modificaciones menores para el tamaño y el orden de los bytes.

fscanf ()

Digamos que tenemos un archivo de texto y queremos leer todas las palabras en ese archivo para cumplir con algunos requisitos.

archivo.txt :

```
This is just
a test file
to be used by fscanf()
```

Esta es la función principal:

```
#include <stdlib.h>
#include <stdio.h>

void printAllWords(FILE *);

int main(void)
{
  FILE *fp;
```

```
  if ((fp = fopen("file.txt", "r")) == NULL) {
      perror("Error opening file");
      exit(EXIT_FAILURE);
  }

  printAllWords(fp);

  fclose(fp);

  return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
    char tmp[20];
    int i = 1;

    while (fscanf(fp, "%19s", tmp) != EOF) {
        printf("Word %d: %s\n", i, tmp);
        i++;
    }
}
```

La salida será:

```
Word 1: This
Word 2: is
Word 3: just
Word 4: a
Word 5: test
Word 6: file
Word 7: to
Word 8: be
Word 9: used
Word 10: by
Word 11: fscanf()
```

Leer líneas de un archivo

El encabezado `stdio.h` define la `fgets()` . Esta función lee una línea de una secuencia y la almacena en una cadena específica. La función deja de leer el texto de la secuencia cuando se leen `n - 1` caracteres, se lee el carácter de nueva línea (`'\n'`) o se alcanza el final del archivo (EOF).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;
```

```
    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* Open file */
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Get each line until there are none left */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* Print each line */
        printf("line[%06d]: %s", ++line_count, line);

        /* Add a trailing newline to lines that don't already have one */
        if (line[strlen(line) - 1] != '\n')
            printf("\n");
    }

    /* Close file */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
    }
}
```

Llamar al programa con un argumento que es una ruta a un archivo que contiene el siguiente texto:

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines



a really long line to show that the line will be counted as two lines if the length of a line
is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Resultará en el siguiente resultado:

```
line[000001]: This is a file
line[000002]:   which has
line[000003]: multiple lines
line[000004]:     with various indentation,
line[000005]: blank lines
line[000006]:
line[000007]:
line[000008]:
line[000009]: a really long line to show that the line will be counted as two lines if the le
```

```
line[000010]: ngth of a line is too long to fit in the buffer it has been given,
line[000011]:  and punctuation at the end of the lines.
line[000012]:
```

Este ejemplo muy simple permite una longitud de línea máxima fija, de modo que las líneas más largas se cuenten efectivamente como dos líneas. La `fgets()` requiere que el código de llamada proporcione la memoria que se utilizará como destino para la línea que se lee.

POSIX pone a disposición la función `getline()` que, en su lugar, asigna internamente memoria para ampliar el búfer según sea necesario para una línea de cualquier longitud (siempre que haya suficiente memoria).

Lea Archivos y flujos de E / S en línea: https://riptutorial.com/es/c/topic/507/archivos-y-flujos-de-e---s

Capítulo 8: Argumentos de línea de comando

Sintaxis

- int main (int argc, char * argv [])

Parámetros

Parámetro	Detalles
argc	recuento de argumentos: se inicializa con el número de argumentos separados por espacios que se asignan al programa desde la línea de comandos, así como el nombre del programa.
argv	vector argumento - inicializado a un conjunto de `char` -pointers (cadenas) que contiene los argumentos (y el nombre del programa) que se le dio en la línea de comandos.

Observaciones

El programa de CA que se ejecuta en un "entorno alojado" (el tipo normal, a diferencia de un "entorno independiente") debe tener una función `main` . Tradicionalmente se define como:

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

Tenga en cuenta que `argv` también puede ser, y muy a menudo es, definido como `char **argv` ; El comportamiento es el mismo. Además, los nombres de los parámetros se pueden cambiar porque son solo variables locales dentro de la función, pero `argc` y `argv` son convencionales y debe usar esos nombres.

Para las funciones `main` donde el código no usa ningún argumento, use `int main(void)` .

Ambos parámetros se inicializan cuando se inicia el programa:

- `argc` se inicializa a la cantidad de argumentos separados por espacios que se le dan al programa desde la línea de comandos, así como al nombre del programa.
- `argv` es una matriz de `char` -pointers (cadenas) que contienen los argumentos (y el nombre del programa) que se le dio en la línea de comandos.
- algunos sistemas expanden los argumentos de la línea de comandos "en el shell", otros no. En Unix, si el usuario escribe `myprogram *.txt` el programa recibirá una lista de archivos de texto; en Windows recibirá la cadena " `*.txt` ".

Nota: Antes de usar `argv` , es posible que deba verificar el valor de `argc` . En teoría, `argc` podría ser `0` , y si `argc` es cero, entonces no hay argumentos y `argv[0]` (equivalente a `argv[argc]`) es un

puntero nulo. Sería un sistema inusual con un entorno alojado si se encontrara con este problema. Del mismo modo, es posible, aunque muy inusual, que no haya información sobre el nombre del programa. En ese caso, `argv[0][0] == '\0'` - el nombre del programa puede estar vacío.

Supongamos que iniciamos el programa así:

```
./some_program abba banana mamajam
```

Entonces `argc` es igual a `4` , y los argumentos de la línea de comando:

- `argv[0]` apunta a `"./some_program"` (el nombre del programa) si el nombre del programa está disponible desde el entorno host. De lo contrario, una cadena vacía `""` .
- `argv[1]` apunta a `"abba"` ,
- `argv[2]` apunta a `"banana"` ,
- `argv[3]` apunta a `"mamajam"` ,
- `argv[4]` contiene el valor `NULL` .

Vea también Qué debería devolver `main()` en C y C ++ para obtener citas completas del estándar.

Examples

Imprimiendo los argumentos de la línea de comando

Después de recibir los argumentos, puede imprimirlos de la siguiente manera:

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
    }
}
```

Notas

1. El parámetro `argv` también se puede definir como `char *argv[]` .
2. `argv[0]` *puede* contener el nombre del programa (dependiendo de cómo se ejecutó el programa). El primer argumento de línea de comando "real" está en `argv[1]` , y esta es la razón por la que la variable de bucle `i` se inicializa en 1.
3. En la declaración de impresión, puede usar `*(argv + i)` lugar de `argv[i]` : se evalúa para la misma cosa, pero es más detallado.
4. Los corchetes alrededor del valor del argumento ayudan a identificar el inicio y el final. Esto puede ser invaluable si hay espacios en blanco, nuevas líneas, retornos de carro u otros caracteres extraños en el argumento. Alguna variante de este programa es una herramienta útil para depurar scripts de shell donde necesita comprender lo que realmente contiene la lista de argumentos (aunque existen alternativas de shell simples que son casi equivalentes).

Imprima los argumentos a un programa y conviértalos a valores enteros.

El siguiente código imprimirá los argumentos en el programa, y el código intentará convertir cada argumento en un número (a un `long`):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>

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

    for (int i = 1; i < argc; i++) {
        printf("Argument %d is: %s\n", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "Argument %d is not a number.\n", i);
        }
        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno ==
ERANGE) {
            fprintf(stderr, "Argument %d is out of range.\n", i);
        }
        else {
            printf("Argument %d is a number, and the value is: %ld\n",
                   i, argument_numValue);
        }
    }
    return 0;
}
```

Referencias

- strtol () devuelve un valor incorrecto
- Uso correcto de strtol

Usando las herramientas getopt de GNU

Las opciones de línea de comandos para aplicaciones no se tratan de manera diferente de los argumentos de línea de comandos por el lenguaje C. Son solo argumentos que, en un entorno Linux o Unix, tradicionalmente comienzan con un guión (`-`).

Con glibc en un entorno Linux o Unix, puede usar las herramientas getopt para definir, validar y analizar fácilmente las opciones de línea de comandos del resto de sus argumentos.

Estas herramientas esperan que sus opciones sean formateadas de acuerdo con los Estándares de Codificación GNU , que es una extensión de lo que POSIX especifica para el formato de las opciones de la línea de comando.

El siguiente ejemplo muestra cómo manejar las opciones de línea de comandos con las herramientas getopt de GNU.

```
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* print a description of all supported options */
void usage (FILE *fp, const char *path)
{
    /* take only the last portion of the path */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;

    fprintf (fp, "usage: %s [OPTION]\n", basename);
    fprintf (fp, "  -h, --help\t\t"
                 "Print this help and exit.\n");
    fprintf (fp, "  -f, --file[=FILENAME]\t"
                 "Write all output to a file (defaults to out.txt).\n");
    fprintf (fp, "  -m, --msg=STRING\t"
                 "Output a particular message rather than 'Hello world'.\n");
}

/* parse command-line options and print message */
int main(int argc, char *argv[])
{
    /* for code brevity this example just uses fixed buffer sizes for strings */
    char filename[256] = { 0 };
    char message[256] = "Hello world";
    FILE *fp;
    int help_flag = 0;
    int opt;

    /* table of all supported options in their long form.
     * fields: name, has_arg, flag, val
     * `has_arg` specifies whether the associated long-form option can (or, in
     * some cases, must) have an argument. the valid values for `has_arg` are
     * `no_argument`, `optional_argument`, and `required_argument`.
     * if `flag` points to a variable, then the variable will be given a value
     * of `val` when the associated long-form option is present at the command
     * line.
     * if `flag` is NULL, then `val` is returned by `getopt_long` (see below)
     * when the associated long-form option is found amongst the command-line
     * arguments.
     */
    struct option longopts[] = {
        { "help", no_argument, &help_flag, 1 },
        { "file", optional_argument, NULL, 'f' },
        { "msg", required_argument, NULL, 'm' },
        { 0 }
    };

    /* infinite loop, to be broken when we are done parsing options */
    while (1) {
        /* getopt_long supports GNU-style full-word "long" options in addition
         * to the single-character "short" options which are supported by
         * getopt.
         * the third argument is a collection of supported short-form options.
         * these do not necessarily have to correlate to the long-form options.
         * one colon after an option indicates that it has an argument, two
         * indicates that the argument is optional. order is unimportant.
         */
        opt = getopt_long (argc, argv, "hf::m:", longopts, 0);
```

```
        if (opt == -1) {
            /* a return value of -1 indicates that there are no more options */
            break;
        }

        switch (opt) {
        case 'h':
            /* the help_flag and value are specified in the longopts table,
             * which means that when the --help option is specified (in its long
             * form), the help_flag variable will be automatically set.
             * however, the parser for short-form options does not support the
             * automatic setting of flags, so we still need this code to set the
             * help_flag manually when the -h option is specified.
             */
            help_flag = 1;
            break;
        case 'f':
            /* optarg is a global variable in getopt.h. it contains the argument
             * for this option. it is null if there was no argument.
             */
            printf ("outarg: '%s'\n", optarg);
            strncpy (filename, optarg ? optarg : "out.txt", sizeof (filename));
            /* strncpy does not fully guarantee null-termination */
            filename[sizeof (filename) - 1] = '\0';
            break;
        case 'm':
            /* since the argument for this option is required, getopt guarantees
             * that aptarg is non-null.
             */
            strncpy (message, optarg, sizeof (message));
            message[sizeof (message) - 1] = '\0';
            break;
        case '?':
            /* a return value of '?' indicates that an option was malformed.
             * this could mean that an unrecognized option was given, or that an
             * option which requires an argument did not include an argument.
             */
            usage (stderr, argv[0]);
            return 1;
        default:
            break;
        }
    }

    if (help_flag) {
        usage (stdout, argv[0]);
        return 0;
    }

    if (filename[0]) {
        fp = fopen (filename, "w");
    } else {
        fp = stdout;
    }

    if (!fp) {
        fprintf(stderr, "Failed to open file.\n");
        return 1;
    }

    fprintf (fp, "%s\n", message);
```

```
    fclose (fp);
    return 0;
}
```

Se puede compilar con `gcc` :

```
gcc example.c -o example
```

Admite tres opciones de línea de comandos (`--help` , `--file` y `--msg`). Todos tienen también una "forma corta" (`-h` , `-f` y `-m`). Las opciones "archivo" y "msg" aceptan argumentos. Si especifica la opción "msg", se requiere su argumento.

Los argumentos para las opciones se formatean como:

- `--option=value` (para opciones de formato largo)
- `-ovalue` o `-o"value"` (para opciones de formato corto)

Lea Argumentos de línea de comando en línea: https://riptutorial.com/es/c/topic/1285/argumentos-de-linea-de-comando

Capítulo 9: Argumentos variables

Introducción

Los argumentos variables son utilizados por las funciones de la familia printf (`printf` , `fprintf` , etc.) y otras para permitir que se llame a una función con un número diferente de argumentos cada vez, de ahí el nombre de *varargs* .

Para implementar funciones usando la característica de argumentos variables, use `#include <stdarg.h>` .

Para llamar a funciones que toman un número variable de argumentos, asegúrese de que haya un prototipo completo con los puntos suspensivos finales en el alcance: `void err_exit(const char *format, ...);` por ejemplo.

Sintaxis

- void va_start (va_list ap, *última*); / * Iniciar el procesamiento de argumentos variadic; *el último* es el último parámetro de la función antes de los puntos suspensivos ("...") * /
- *escriba* va_arg (va_list ap, *type*); / * Obtener el siguiente argumento variad en la lista; asegúrese de pasar el tipo correcto *promovido* * /
- void va_end (va_list ap); / * Procesamiento de argumento final * /
- void va_copy (va_list dst, va_list src); / * C99 o posterior: copie la lista de argumentos, es decir, la posición actual en el procesamiento de argumentos, en otra lista (por ejemplo, para pasar los argumentos varias veces) * /

Parámetros

Parámetro	Detalles
`va_list` ap	indicador de argumento, posición actual en la lista de argumentos variadic
último	nombre del último argumento de función no variada, por lo que el compilador encuentra el lugar correcto para comenzar a procesar los argumentos variadic; no se puede declarar como una variable de `register` , una función o un tipo de matriz
tipo	tipo **promovido** del argumento variadic para leer (por ejemplo, `int` para un `short int` argumento `short int`)
`va_list` src	argumento actual para copiar
`va_list` dst	nueva lista de argumentos para completar

Observaciones

Las `va_start` , `va_arg` , `va_end` y `va_copy` son en realidad macros.

Asegúrese de llamar *siempre* a `va_start` primero, y solo una vez, y para llamar a `va_end` último, y solo una vez, y en cada punto de salida de la función. No hacerlo *puede* funcionar en *su* sistema, pero seguramente **no** es portátil y, por lo tanto, invita a errores.

Tenga cuidado de declarar su función correctamente, es decir, con un prototipo, y tenga en cuenta las restricciones del *último* argumento no variado (no `register` , no es una función o tipo de matriz). No es posible declarar una función que solo tome argumentos variables, ya que se necesita al menos un argumento no variable para poder iniciar el procesamiento de argumentos.

Al llamar a `va_arg` , debe solicitar el tipo de argumento **promovido** , es decir:

- `short` se promueve a `int` (y `unsigned short` también se promueve a `int` menos que `sizeof(unsigned short) == sizeof(int)` , en cuyo caso se promueve a `unsigned int`).
- Se promueve `float` al `double` .
- `signed char` se promueve a `int` ; `unsigned char` también se promueve a `int` menos que `sizeof(unsigned char) == sizeof(int)` , que rara vez es el caso.
- `char` se suele ascendido a `int` .
- Los tipos C99 como `uint8_t` o `int16_t` se promueven de manera similar.

El procesamiento del argumento variad histórico (es decir, K&R) se declara en `<varargs.h>` pero no se debe usar porque está obsoleto. El procesamiento de argumentos variadic estándar (el descrito aquí y declarado en `<stdarg.h>`) se introdujo en C89; la macro `va_copy` se introdujo en C99 pero fue proporcionada por muchos compiladores antes de eso.

Examples

Usando un argumento de conteo explícito para determinar la longitud de la lista va

Con cualquier función variable, la función debe saber cómo interpretar la lista de argumentos variables. Con las funciones `printf()` o `scanf()` , la cadena de formato le dice a la función qué esperar.

La técnica más simple es pasar un conteo explícito de los otros argumentos (que normalmente son todos del mismo tipo). Esto se demuestra en la función variadic en el código siguiente, que calcula la suma de una serie de enteros, donde puede haber cualquier número de enteros, pero ese recuento se especifica como un argumento antes de la lista de argumentos variables.

```
#include <stdio.h>
#include <stdarg.h>

/* first arg is the number of following int args to sum. */
int sum(int n, ...) {
    int sum = 0;
```

```
    va_list it; /* hold information about the variadic argument list. */

    va_start(it, n); /* start variadic argument processing */
    while (n--)
      sum += va_arg(it, int); /* get and sum the next variadic argument */
    va_end(it); /* end variadic argument processing */

    return sum;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /* prints 15 */
    printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* prints 666 */
    return 0;
}
```

Usando valores terminadores para determinar el final de va_list

Con cualquier función variable, la función debe saber cómo interpretar la lista de argumentos variables. El enfoque "tradicional" (ejemplificado por `printf`) es especificar el número de argumentos por adelantado. Sin embargo, esto no siempre es una buena idea:

```
/* First argument specifies the number of parameters; the remainder are also int */
extern int sum(int n, ...);

/* But it's far from obvious from the code. */
sum(5, 2, 1, 4, 3, 6)

/* What happens if i.e. one argument is removed later on? */
sum(5, 2, 1, 3, 6)  /* Disaster */
```

A veces es más robusto agregar un terminador explícito, ejemplificado por la función `execlp()` POSIX. Aquí hay otra función para calcular la suma de una serie de números `double` :

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* Sums args up until the terminator NAN */
double sum (double x, ...) {
  double sum = 0;
  va_list va;

  va_start(va, x);
  for (; !isnan(x); x = va_arg(va, double)) {
    sum += x;
  }
  va_end(va);

  return sum;
}

int main (void) {
  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));
  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

Buenos valores de terminador:

- entero (se supone que es todo positivo o no negativo) - `0` o `-1`
- tipos de punto flotante - `NAN`
- tipos de punteros - `NULL`
- tipos de enumerador - algún valor especial

Implementando funciones con una interfaz similar a `printf ()`

Un uso común de las listas de argumentos de longitud variable es implementar funciones que son un envoltorio delgado alrededor de la familia de funciones `printf()` . Un ejemplo de ello es un conjunto de funciones de informe de errores.

errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif
```

Este es un ejemplo escueto; Tales paquetes pueden ser mucho más elaborados. Normalmente, los programadores usarán `errmsg()` o `warnmsg()` , que a su vez usan `verrmsg()` internamente. Sin embargo, si alguien tiene la necesidad de hacer más, entonces la función `verrmsg()` expuesta será útil. Se podría evitar la exposición hasta que haya una necesidad de ella (YAGNI - yagni), pero la necesidad va a surgir con el tiempo (que *está* a necesitarlo - YAGNI).

errmsg.c

Este código solo necesita reenviar los argumentos variadic a la función `vfprintf()` para generar un error estándar. También informa el mensaje de error del sistema correspondiente al número de error del sistema (`errno`) pasado a las funciones.

```
#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    if (fmt)
        vfprintf(stderr, fmt, ap);
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));
    putc('\n', stderr);
}
```

```
void
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

void
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}
```

Utilizando `errmsg.h`

Ahora puedes usar esas funciones de la siguiente manera:

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer),
filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* continue the program */
    return 0;
}
```

Si las llamadas del sistema `open()` o `read()` fallan, el error se escribe en el error estándar y el programa sale con el código de salida 1. Si falla la llamada al sistema `close()` , el error simplemente se imprime como un mensaje de advertencia, y el programa continúa

Comprobando el uso correcto de los formatos `printf()`

Si está utilizando GCC (el compilador GNU C, que forma parte de la colección del compilador GNU), o está usando Clang, puede hacer que el compilador verifique que los argumentos que pasa a las funciones de mensaje de error coinciden con lo que espera `printf()` . Dado que no todos los compiladores admiten la extensión, se debe compilar de forma condicional, lo cual es un poco complicado. Sin embargo, la protección que da merece el esfuerzo.

Primero, necesitamos saber cómo detectar que el compilador es GCC o Clang emulando GCC. La respuesta es que GCC define `__GNUC__` para indicar eso.

Consulte los atributos de funciones comunes para obtener información sobre los atributos, específicamente el atributo de `format` .

Reescrito `errmsg.h`

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format(printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* If only */
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
        PRINTFLIKE(3, 4);
void warnmsg(int errnum, const char *fmt, ...)
        PRINTFLIKE(2, 3);

#endif
```

Ahora, si cometes un error como:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(donde `%d` debería ser `%s`), entonces el compilador se quejará:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: In function 'main':
erruse.c:20:64: error: format '%d' expects argument of type 'int', but argument 4 has type
'const char *' [-Werror=format=]
         errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
                                                          ~^
                                                          %s
cc1: all warnings being treated as errors
$
```

Usando una cadena de formato

El uso de una cadena de formato proporciona información sobre el número esperado y el tipo de los argumentos variables posteriores para evitar la necesidad de un argumento de recuento explícito o un valor de terminación.

El siguiente ejemplo muestra una función que envuelve la `printf()` estándar `printf()` , solo permite el uso de argumentos variables del tipo `char` , `int` y `double` (en formato de coma flotante decimal). Aquí, como con `printf()` , el primer argumento de la función de ajuste es la cadena de formato. A medida que se analiza la cadena de formato, la función puede determinar si se espera otro argumento variadic y cuál debería ser su tipo.

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* hold information about the variadic argument list. */
    int printed = 0; /* count of printed characters */

    va_start(ap, format); /* start variadic argument processing */

    while (*format != '\0') /* read format string until string terminator */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
            switch(*format)
            {
                case 'c' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument, note
type promotion from char to int */
                    break;
                case 'd' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument */
                    break;

                case 'f' :
                    f = printf("%f", va_arg(ap, double)); /* print next variadic argument */
                    break;
                default :
                    f = -1; /* invalid format specifier */
                    break;
            }
        }
        else
        {
            f = printf("%c", *format); /* print any other characters */
        }

        if (f < 0) /* check for errors */
        {
            printed = f;
            break;
        }
        else
        {
            printed += f;
```

```
        }
        ++format; /* move on to next character in string */
    }

    va_end(ap); /* end variadic argument processing */

    return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

    y = simple_printf("There are %d characters in this sentence", x);
    simple_printf("\n%d were printed\n", y);
}
```

Lea Argumentos variables en línea: https://riptutorial.com/es/c/topic/455/argumentos-variables

Capítulo 10: Arrays

Introducción

Las matrices son tipos de datos derivados, que representan una colección ordenada de valores ("elementos") de otro tipo. La mayoría de las matrices en C tienen un número fijo de elementos de cualquier tipo, y su representación almacena los elementos de forma contigua en la memoria sin espacios ni rellenos. C permite matrices multidimensionales cuyos elementos son otras matrices y también matrices de punteros.

C admite matrices asignadas dinámicamente cuyo tamaño se determina en el tiempo de ejecución. C99 y posteriores admiten matrices de longitud variable o VLA.

Sintaxis

- escriba nombre [longitud]; / * Definir matriz de 'tipo' con nombre 'nombre' y longitud 'longitud'. * /
- int arr [10] = {0}; / * Definir una matriz e inicializar TODOS los elementos a 0. * /
- int arr [10] = {42}; / * Definir una matriz e inicializar los primeros elementos a 42 y el resto a 0. * /
- int arr [] = {4, 2, 3, 1}; / * Definir e inicializar una matriz de longitud 4. * /
- arr [n] = valor; / * Establecer valor en el índice n. * /
- valor = arr [n]; / * Obtener valor en el índice n. * /

Observaciones

¿Por qué necesitamos matrices?

Las matrices proporcionan una forma de organizar los objetos en un agregado con su propio significado. Por ejemplo, cadenas de C son matrices de caracteres (`char` s), y una cadena como "Hola, mundo!" tiene un significado como un agregado que no es inherente a los caracteres individualmente. De manera similar, las matrices se usan comúnmente para representar vectores matemáticos y matrices, así como listas de muchos tipos. Además, sin una forma de agrupar los elementos, uno tendría que abordar cada uno individualmente, como a través de variables separadas. No solo es difícil de manejar, sino que no se adapta fácilmente a colecciones de diferentes longitudes.

Las matrices se convierten implícitamente a punteros en la mayoría de los contextos .

Excepto cuando aparece como el operando de la `sizeof` operador, el `_Alignof` operador (C2011), o el unario `&` operador (dirección-de), o como un literal de cadena utilizado para inicializar un (otro) de matriz, una matriz se convierte implícitamente en ("decae a") un puntero a su primer elemento. Esta conversión implícita está estrechamente relacionada con la definición del operador de subíndice de matriz (`[]`): la expresión `arr[idx]` se define como equivalente a `*(arr + idx)` . Además, dado que la aritmética de punteros es conmutativa, `*(arr + idx)` también es equivalente

a `*(idx + arr)` , que a su vez es equivalente a `idx[arr]` . Todas esas expresiones son válidas y se evalúan con el mismo valor, siempre que `idx` o `arr` sean un puntero (o una matriz, que decaiga en un puntero), la otra es un número entero, y el entero es un índice válido en la matriz a la que apunta el puntero.

Como caso especial, observe que `&(arr[0])` es equivalente a `&*(arr + 0)` , lo que simplifica a `arr` . Todas esas expresiones son intercambiables siempre que la última se desintegre a un puntero. Esto simplemente expresa nuevamente que una matriz se desintegra en un puntero a su primer elemento.

Por el contrario, si la dirección del operador se aplica a una matriz de tipo `T[N]` (*es decir,* `&arr`), el resultado tiene el tipo `T (*)[N]` y apunta a toda la matriz. Esto es distinto de un puntero al primer elemento de la matriz al menos con respecto a la aritmética de punteros, que se define en términos del tamaño del tipo apuntado.

Los parámetros de función no son matrices .

```
void foo(int a[], int n);
void foo(int *a, int n);
```

Aunque la primera declaración de `foo` usa una sintaxis similar a `a` matriz para el parámetro `a` , dicha sintaxis se usa para declarar que un parámetro de función declara ese parámetro como un *puntero* al tipo de elemento de la matriz. Por lo tanto, la segunda firma para `foo()` es semánticamente idéntica a la primera. Esto corresponde a la caída de los valores de matriz a los punteros donde aparecen como argumentos a una *llamada de* función, de modo que si una variable y un parámetro de función se declaran con el mismo tipo de matriz, el valor de esa variable es adecuado para su uso en una llamada de función como el Argumento asociado al parámetro.

Examples

Declarar e inicializar una matriz

La sintaxis general para declarar una matriz unidimensional es

```
type arrName[size];
```

donde `type` podría ser cualquier tipo incorporado o tipos definidos por el usuario, como las estructuras, `arrName` es un identificador definido por el usuario, y `size` es una constante entera.

La declaración de una matriz (una matriz de 10 variables int en este caso) se realiza de la siguiente manera:

```
int array[10];
```

ahora tiene valores indeterminados. Para asegurarse de que mantiene valores cero al declarar, puede hacer esto:

```
int array[10] = {0};
```

Las matrices también pueden tener inicializadores, en este ejemplo se declara una matriz de 10 `int` , donde las primeras 3 `int` contendrán los valores `1` , `2` , `3` , todos los demás valores serán cero:

```
int array[10] = {1, 2, 3};
```

En el método de inicialización anterior, el primer valor de la lista se asignará al primer miembro de la matriz, el segundo valor se asignará al segundo miembro de la matriz y así sucesivamente. Si el tamaño de la lista es más pequeño que el tamaño de la matriz, entonces, como en el ejemplo anterior, los miembros restantes de la matriz se inicializarán a ceros. Con la inicialización de lista designada (ISO C99), es posible la inicialización explícita de los miembros de la matriz. Por ejemplo,

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* array is {0, 2, 5, 0, 9} */
```

En la mayoría de los casos, el compilador puede deducir la longitud de la matriz para usted, esto se puede lograr dejando los corchetes vacíos:

```
int array[] = {1, 2, 3}; /* an array of 3 int's */
int array[] = {[3] = 8, [0] = 9}; /* size is 4 */
```

No se permite declarar una matriz de longitud cero.

C99 C11

Las matrices de longitud variable (VLA para abreviar) se agregaron en C99 y se hicieron opcionales en C11. Son iguales a las matrices normales, con una diferencia importante: la longitud no debe conocerse en el momento de la compilación. Los VLA tienen duración de almacenamiento automático. Solo los punteros a VLA pueden tener una duración de almacenamiento estático.

```
size_t m = calc_length(); /* calculate array length at runtime */
int vla[m];               /* create array with calculated length */
```

Importante:

Los VLA son potencialmente peligrosos. Si la matriz `vla` en el ejemplo anterior requiere más espacio en la pila que el disponible, la pila se desbordará. Por lo tanto, el uso de VLA se suele desaconsejar en las guías de estilo y en los libros y ejercicios.

Borrar el contenido de la matriz (puesta a cero)

A veces es necesario establecer una matriz en cero, después de que se haya realizado la inicialización.

```
#include <stdlib.h> /* for EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
  int array[ARRLEN]; /* Allocated but not initialised, as not defined static or global. */

  size_t i;
  for(i = 0; i < ARRLEN; ++i)
  {
    array[i] = 0;
  }

  return EXIT_SUCCESS;
}
```

Un atajo común al bucle anterior es usar `memset()` de `<string.h>` . Al pasar la `array` como se muestra a continuación, se descompone en un puntero a su primer elemento.

```
memset(array, 0, ARRLEN * sizeof (int)); /* Use size explicitly provided type (int here). */
```

o

```
memset(array, 0, ARRLEN * sizeof *array); /* Use size of type the pointer is pointing to. */
```

Como en este ejemplo, la `array` *es* una matriz y no solo un puntero al primer elemento de una matriz (ver longitud de la matriz sobre por qué esto es importante) una tercera opción para eliminar la matriz es 0:

```
 memset(array, 0, sizeof array); /* Use size of the array itself. */
```

Longitud de la matriz

Las matrices tienen longitudes fijas que se conocen dentro del alcance de sus declaraciones. Sin embargo, es posible y, a veces, conveniente calcular las longitudes de matriz. En particular, esto puede hacer que el código sea más flexible cuando la longitud de la matriz se determina automáticamente a partir de un inicializador:

```
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

/* size of `array` in bytes */
size_t size = sizeof(array);

/* number of elements in `array` */
size_t length = sizeof(array) / sizeof(array[0]);
```

Sin embargo, en la mayoría de los contextos donde aparece una matriz en una expresión, se convierte automáticamente en ("decae a") un puntero a su primer elemento. El caso en el que una matriz es el operando del operador `sizeof` es una de las pocas excepciones. El puntero resultante no es en sí mismo una matriz y no transporta ninguna información sobre la longitud de la matriz

de la que se derivó. Por lo tanto, si esa longitud se necesita junto con el puntero, como cuando el puntero se pasa a una función, debe transmitirse por separado.

Por ejemplo, supongamos que queremos escribir una función para devolver el último elemento de una matriz de `int` . Continuando con lo anterior, podríamos llamarlo así:

```
/* array will decay to a pointer, so the length must be passed separately */
int last = get_last(array, length);
```

La función se podría implementar así:

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

Tenga en cuenta en particular que aunque la declaración de `input` de parámetros se parece a la de una matriz, **de hecho declara la `input` como un puntero** (a `int`). Es exactamente equivalente a declarar `input` como `input int *input` . Lo mismo sería cierto incluso si se diera una dimensión. Esto es posible porque las matrices nunca pueden ser argumentos reales de las funciones (se descomponen en los punteros cuando aparecen en expresiones de llamada de función), y se pueden ver como mnemotécnicas.

Es un error muy común intentar determinar el tamaño de la matriz desde un puntero, que no puede funcionar. NO HAGAS ESTO:

```
int BAD_get_last(int input[]) {
    /* INCORRECTLY COMPUTES THE LENGTH OF THE ARRAY INTO WHICH input POINTS: */
    size_t length = sizeof(input) / sizeof(input[0]));

    return input[length - 1];  /* Oops -- not the droid we are looking for */
}
```

De hecho, ese error en particular es tan común que algunos compiladores lo reconocen y lo advierten. `clang` , por ejemplo, emitirá la siguiente advertencia:

```
warning: sizeof on array function parameter will return size of 'int *' instead of 'int []' [-
Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
                            ^
note: declared here
int BAD_get_last(int input[])
                     ^
```

Configuración de valores en matrices

El acceso a los valores de la matriz se realiza generalmente entre corchetes:

```
int val;
int array[10];

/* Setting the value of the fifth element to 5: */
```

```
array[4] = 5;

/* The above is equal to: */
*(array + 4) = 5;

/* Reading the value of the fifth element: */
val = array[4];
```

Como efecto secundario de los operandos al operador `+` que es intercambiable (-> ley conmutativa), lo siguiente es equivalente:

```
*(array + 4) = 5;
*(4 + array) = 5;
```

así también las siguientes afirmaciones son equivalentes:

```
array[4] = 5;
4[array] = 5; /* Weird but valid C ... */
```

y esos dos también:

```
val = array[4];
val = 4[array]; /* Weird but valid C ... */
```

C no realiza ninguna verificación de límites, el acceso al contenido fuera de la matriz declarada no está definido (se accede a la memoria más allá del fragmento asignado):

```
int val;
int array[10];

array[4] = 5;    /* ok */
val = array[4];  /* ok */
array[19] = 20;  /* undefined behavior */
val = array[15]; /* undefined behavior */
```

Definir matriz y elemento de matriz de acceso.

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{

   int n[ ARRLEN ]; /* n is an array of 10 integers */
   size_t i, j; /* Use size_t to address memory, that is to index arrays, as its guaranteed to

                   be wide enough to address all of the possible available memory.
                   Using signed integers to do so should be considered a special use case,
                   and should be restricted to the uncommon case of being in the need of
                   negative indexes. */

   /* Initialize elements of array n. */
```

```
    for ( i = 0; i < ARRLEN ; i++ )
    {
       n[ i ] = i + 100; /* Set element at location i to i + 100. */
    }

    /* Output each array element's value. */
    for (j = 0; j < ARRLEN ; j++ )
    {
       printf("Element[%zu] = %d\n", j, n[j] );
    }

    return 0;
}
```

Asignar y cero inicializar una matriz con el tamaño definido por el usuario

```
#include <stdio.h>
#include <stdlib.h>


int main (void)
{
  int * pdata;
  size_t n;

  printf ("Enter the size of the array: ");
  fflush(stdout); /* Make sure the prompt gets printed to buffered stdout. */

  if (1 != scanf("%zu", &n)) /* If zu is not supported (Windows?) use lu. */
  {
    fprintf("scanf() did not read a in proper value.\n");
    exit(EXIT_FAILURE);
  }

  pdata = calloc(n, sizeof *pdata);
  if (NULL == pdata)
  {
    perror("calloc() failed"); /* Print error. */
    exit(EXIT_FAILURE);
  }

  free(pdata); /* Clean up. */

  return EXIT_SUCCESS;
}
```

Este programa intenta escanear en un valor entero sin signo de la entrada estándar, asignar un bloque de memoria para una matriz de `n` elementos de tipo `int` llamando a la función `calloc()` . La memoria se inicializa a todos los ceros por este último.

En caso de éxito la memoria es `free()` por la llamada a `free()` .

Iterando a través de una matriz de manera eficiente y orden de fila mayor

Las matrices en C se pueden ver como una porción contigua de memoria. Más precisamente, la última dimensión de la matriz es la parte contigua. Llamamos a esto el *orden de* la *fila principal* .

Al comprender esto y el hecho de que un error de caché carga una línea de caché completa en el caché cuando se accede a datos no almacenados en caché para evitar fallas de caché subsiguientes, podemos ver por qué el acceso a una matriz de dimensión 10000x10000 con `array[0][0]` podría **potencialmente** cargar `array[0][1]` en caché, pero acceder a la `array[1][0]` justo después generaría un segundo error de caché, ya que está a `sizeof(type)*10000` bytes de la `array[0][0]` , y por lo tanto no en la misma línea de caché. Es por eso que iterar así es ineficiente:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

E iterar así es más eficiente:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

En la misma línea, esta es la razón por la que cuando se trata de una matriz con una dimensión y múltiples índices (digamos 2 dimensiones aquí por simplicidad con los índices i y j), es importante recorrer la matriz de esta manera:

```
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}
```

O con 3 dimensiones e índices i, j y k:

```
#define DIM_X 10
```

```
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}
```

O de una manera más genérica, cuando tenemos una matriz con **N1 x N2 x ... x Nd** elementos, **d** dimensiones e índices anotados como **n1, n2, ..., nd** el desplazamiento se calcula de esta manera

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1) \cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

Imagen / fórmula tomada de: https://en.wikipedia.org/wiki/Row-major_order

Matrices multidimensionales

El lenguaje de programación en C permite matrices multidimensionales . Aquí está la forma general de una declaración de matriz multidimensional:

```
type name[size1][size2]...[sizeN];
```

Por ejemplo, la siguiente declaración crea una matriz de enteros tridimensional (5 x 10 x 4):

```
int arr[5][10][4];
```

Arrays bidimensionales

La forma más simple de matriz multidimensional es la matriz bidimensional. Una matriz bidimensional es, en esencia, una lista de matrices unidimensionales. Para declarar una matriz entera bidimensional de dimensiones mxn, podemos escribir de la siguiente manera:

```
type arrayName[m][n];
```

Donde `type` puede ser cualquier tipo de datos de C válido (`int` , `float` , etc.) y `arrayName` puede ser cualquier identificador de C válido. Una matriz bidimensional se puede visualizar como una tabla con `m` filas `n` columnas. **Nota** : el orden *sí* importa en C. La matriz `int a[4][3]` no es lo mismo que la matriz `int a[3][4]` . El número de filas es lo primero, ya que C es un idioma principal de la *fila* .

Una matriz bidimensional `a` , que contiene tres filas y cuatro columnas se puede mostrar de la siguiente manera:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Por lo tanto, cada elemento de la matriz `a` se identifica por un nombre de elemento de la forma `a[i][j]` , donde `a` es el nombre de la matriz, `i` representa qué fila y `j` representa qué columna. Recuerde que las filas y columnas están indexadas en cero. Esto es muy similar a la notación matemática para las matrices 2D de subíndices.

Inicializando matrices bidimensionales

Los arreglos multidimensionales pueden inicializarse especificando valores entre corchetes para cada fila. Lo siguiente define una matriz con 3 filas donde cada fila tiene 4 columnas.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Las llaves anidadas, que indican la fila deseada, son opcionales. La siguiente inicialización es equivalente al ejemplo anterior:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Si bien el método para crear matrices con llaves anidadas es opcional, se recomienda encarecidamente ya que es más legible y más claro.

Acceso a elementos de matriz bidimensional

Se accede a un elemento de una matriz bidimensional utilizando los subíndices, es decir, el índice de fila y el índice de columna de la matriz. Por ejemplo

```
int val = a[2][3];
```

La declaración anterior tomará el cuarto elemento de la tercera fila de la matriz. Revisemos el siguiente programa en el que hemos utilizado un bucle anidado para manejar una matriz bidimensional:

```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
```

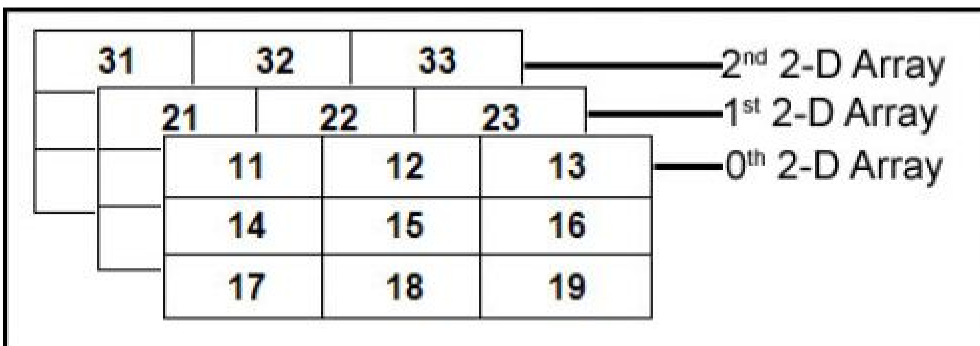
```
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {

      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

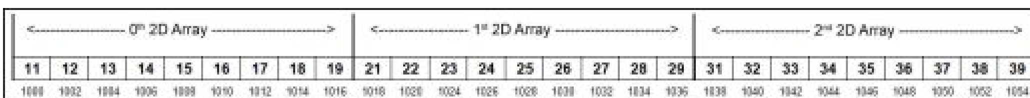
```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Array tridimensional:

Una matriz 3D es esencialmente una matriz de matrices de matrices: es una matriz o colección de matrices 2D, y una matriz 2D es una matriz de matrices 1D.



Mapa de memoria de matriz 3D:



Inicializando una matriz 3D:

```
double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

Podemos tener matrices con cualquier número de dimensiones, aunque es probable que la mayoría de las matrices que se crean sean de una o dos dimensiones.

Iterando a través de una matriz utilizando punteros

```
#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be i*i */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}
```

Aquí, en la inicialización de `p` en la primera condición de bucle `for` , la matriz `a` *decae* a un puntero a su primer elemento, como lo haría en casi todos los lugares donde se usa dicha variable de matriz.

Luego, `++p` realiza aritmética de punteros en el puntero `p` y recorre uno por uno los elementos de la matriz, y se refiere a ellos desreferiéndolos con `*p` .

Pasar matrices multidimensionales a una función.

Las matrices multidimensionales siguen las mismas reglas que las matrices unidimensionales al pasarlas a una función. Sin embargo, la combinación de decaimiento a puntero, la precedencia del operador y las dos formas diferentes de declarar una matriz multidimensional (matriz de matrices frente a matriz de punteros) puede hacer que la declaración de tales funciones no sea intuitiva. El siguiente ejemplo muestra las formas correctas de pasar matrices multidimensionales.

```
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual.  But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}
```

```
/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int
   *(x[4]), i.e. an array of 4 pointers to int.  But if it's declared as a
   function parameter, it decays into a pointer and becomes int **x,
   which is not compatable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* Here we're dynamically creating an array of pointers.  Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}
```

Ver también

Pasando en Arrays a Funciones

Lea Arrays en línea: https://riptutorial.com/es/c/topic/322/arrays

Capítulo 11: Atomística

Sintaxis

- `#ifdef __STDC_NO_ATOMICS__`
- `# error this implementation needs atomics`
- `#endif`
- `#include <stdatomic.h>`
- sin firma _Atomic counter = ATOMIC_VAR_INIT (0);

Observaciones

La atómica como parte del lenguaje C es una característica opcional que está disponible desde C11.

Su propósito es garantizar el acceso sin carreras a las variables que se comparten entre diferentes subprocesos. Sin la calificación atómica, el estado de una variable compartida sería indefinido si dos subprocesos acceden a ella simultáneamente. Por ejemplo, una operación de incremento (`++`) podría dividirse en varias instrucciones del ensamblador, una lectura, la adición en sí y una instrucción de almacenamiento. Si otro hilo estuviera realizando la misma operación, sus dos secuencias de instrucciones podrían estar entrelazadas y dar lugar a un resultado inconsistente.

- **Tipos:** Todos los tipos de objetos, con excepción de los tipos de matriz, pueden calificarse con `_Atomic` .

- **Operadores:** Se garantiza que todos los **operadores de** lectura-modificación-escritura (por ejemplo, `++` o `*=`) en estos son atómicos.

- **Operaciones:** hay algunas otras operaciones que se especifican como funciones genéricas de tipo, por ejemplo, `atomic_compare_exchange` .

- **Subprocesos:** se garantiza que el acceso a ellos no producirá una carrera de datos cuando se acceda a ellos mediante diferentes subprocesos.

- **Manejadores de señales: los** tipos atómicos se denominan sin *bloqueo* si todas las operaciones en ellos son sin estado. En ese caso, también pueden usarse para tratar cambios de estado entre el flujo de control normal y un manejador de señales.

- Solo hay un tipo de datos que está garantizado como libre de bloqueo: `atomic_flag` . Este es un tipo mínimo en el que las operaciones están diseñadas para asignarse a instrucciones de hardware de prueba y configuración eficientes.

Otros medios para evitar las condiciones de carrera están disponibles en la interfaz de subprocesos de C11, en particular un tipo de mutex `mtx_t` para excluir mutuamente que los subprocesos accedan a datos críticos o secciones críticas de código. Si no se dispone de atomics, estos deben usarse para prevenir razas.

Examples

atómicos y operadores

Se puede acceder a las variables atómicas simultáneamente entre diferentes hilos sin crear condiciones de carrera.

```
/* a global static variable that is visible by all threads */
static unsigned _Atomic active = ATOMIC_VAR_INIT(0);


int myThread(void* a) {
  ++active;         // increment active race free
  // do something
  --active;         // decrement active race free
  return 0;
}
```

Todas las operaciones de valor l (operaciones que modifican el objeto) que están permitidas para el tipo base están permitidas y no conducirán a condiciones de carrera entre los diferentes subprocesos que acceden a ellas.

- Las operaciones en objetos atómicos son generalmente órdenes de magnitud más lentas que las operaciones aritméticas normales. Esto también incluye operaciones simples de carga o almacenamiento. Así que solo deberías usarlos para tareas críticas.
- Operaciones aritméticas habituales y asignación como `a = a+1;` son, de hecho, tres operaciones en `a` : primero una carga, luego adición y finalmente una tienda. Esto *no* es raza libre. Sólo la operación `a += 1;` y `a++;` son.

Lea Atomística en línea: https://riptutorial.com/es/c/topic/4924/atomistica

Capítulo 12: Booleano

Observaciones

Para usar el tipo predefinido `_Bool` y el encabezado `<stdbool.h>` , debe usar las versiones C99 / C11 de C.

Para evitar advertencias de compilación y posibles errores, solo debe usar el ejemplo `typedef` / `define` si está usando C89 y versiones anteriores del idioma.

Examples

Utilizando stdbool.h

C99

El uso del archivo de cabecera del sistema `stdbool.h` permite usar `bool` como un tipo de datos booleano. `true` evalúa como `1` y `false` evalúa como `0` .

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;  /* equivalent to bool x = 1; */
    bool y = false; /* equivalent to bool y = 0; */
    if (x)  /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

`bool` es solo una buena ortografía para el tipo de datos `_Bool` . Tiene reglas especiales cuando los números o punteros se convierten a él.

Usando #define

C de todas las versiones, tratará efectivamente cualquier valor entero distinto de `0` como `true` para los operadores de comparación y el valor entero `0` como `false` . Si no tiene disponible `_Bool` o `bool` partir de C99, puede simular un tipo de datos Boolean en C usando macros `#define` , y aún puede encontrar esas cosas en el código heredado.

```
#include <stdio.h>

#define bool int
#define true 1
```

```
#define false 0

int main(void) {
    bool x = true;  /* Equivalent to int x = 1; */
    bool y = false; /* Equivalent to int y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

No introduzca esto en el nuevo código ya que la definición de estas macros podría chocar con los usos modernos de `<stdbool.h>` .

Usando el tipo intrínseco (incorporado) _Bool

C99

Agregado en la versión C99 estándar de C, `_Bool` también es un tipo de datos C nativo. Es capaz de mantener los valores `0` (para *falso*) y `1` (para *verdadero*).

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* Equivalent to if (x == 1) */
    {
        puts("This will print!");
    }
    if (!y) /* Equivalent to if (y == 0) */
    {
        puts("This will also print!");
    }
}
```

`_Bool` es un tipo entero pero tiene reglas especiales para conversiones de otros tipos. El resultado es análogo al uso de otros tipos en expresiones `if` . En el siguiente

```
_Bool z = X;
```

- Si `X` tiene un tipo aritmético (es cualquier tipo de número), `z` convierte en `0` si `X == 0` . De lo contrario `z` convierte en `1` .
- Si `X` tiene un tipo de puntero, `z` convierte en `0` si `X` es un puntero nulo y `1` contrario.

Para utilizar más agradables ortografía `bool` , `false` y `true` es necesario utilizar `<stdbool.h>` .

Enteros y punteros en expresiones booleanas.

Todos los enteros o punteros se pueden usar en una expresión que se interpreta como "valor de verdad".

```
int main(int argc, char* argv[]) {
  if (argc % 4) {
    puts("arguments number is not divisible by 4");
  } else {
    puts("argument number is divisible by 4");
  }
...
```

La expresión `argc % 4` se evalúa y conduce a uno de los valores `0` , `1` , `2` o `3` . El primero, `0` es el único valor que es "falso" y lleva la ejecución a la parte `else` . Todos los demás valores son "verdaderos" y entran en la parte `if` .

```
double* A = malloc(n*sizeof *A);
if (!A) {
   perror("allocation problems");
   exit(EXIT_FAILURE);
}
```

Aquí se evalúa el puntero `A` y, si es un puntero nulo, se detecta un error y el programa sale.

Muchas personas prefieren escribir algo como `A == NULL` , en cambio, pero si tiene comparaciones de punteros como parte de otras expresiones complicadas, las cosas se vuelven difíciles de leer rápidamente.

```
char const* s = ....;   /* some pointer that we receive */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

Para verificar esto, tendría que escanear un código complicado en la expresión y estar seguro de las preferencias del operador.

```
char const* s = ....;   /* some pointer that we receive */
if (s && s[0] && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

es relativamente fácil de capturar: si el puntero es válido, verificamos si el primer carácter es distinto de cero y luego verificamos si es una letra.

Definiendo un tipo bool usando typedef

Teniendo en cuenta que la mayoría de los depuradores no conocen las macros de `#define` , pero pueden verificar las constantes de `enum` , puede ser conveniente hacer algo como esto:

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
/* Modern C code might expect these to be macros. */
```

```
# ifndef bool
#  define bool bool
# endif
# ifndef true
#  define true true
# endif
# ifndef false
#  define false false
# endif
#else
# include <stdbool.h>
#endif

/* Somewhere later in the code ... */
bool b = true;
```

Esto permite que los compiladores funcionen con versiones históricas de C, pero sigue siendo compatible con versiones posteriores si el código se compila con un compilador de C moderno.

Para obtener más información sobre `typedef` , consulte Typedef , para obtener más información sobre `enum` consulte Enumeraciones

Lea Booleano en línea: https://riptutorial.com/es/c/topic/3336/booleano

Capítulo 13: Calificadores de tipo

Observaciones

Los calificadores de tipo son las palabras clave que describen semánticas adicionales sobre un tipo. Son una parte integral de las firmas de tipo. Pueden aparecer tanto en el nivel superior de una declaración (que afecta directamente al identificador) como en los subniveles (relevantes solo para los indicadores, que afectan los valores apuntados a):

Palabra clave	Observaciones
`const`	Evita la mutación del objeto declarado (al aparecer en el nivel superior) o evita la mutación del valor apuntado (al lado de un subtipo de puntero).
`volatile`	Informa al compilador que el objeto declarado (en el nivel superior) o el valor apuntado (en los subtipos de puntero) pueden cambiar su valor como resultado de condiciones externas, no solo como resultado del flujo de control del programa.
`restrict`	Una sugerencia de optimización, relevante solo para los punteros. Declara la intención de que durante la vida útil del puntero, no se utilizarán otros punteros para acceder al mismo objeto apuntado.

El ordenamiento de los calificadores de tipo con respecto a los especificadores de clase de almacenamiento (`static` , `extern` , `auto` , `register`), modificadores de tipo (con `signed` , `unsigned` , `short` , `long`) y los especificadores de tipo (`int` , `char` , `double` , etc.) no se aplica, pero La buena práctica es ponerlos en el orden mencionado:

```
static const volatile unsigned long int a = 5; /* good practice */
unsigned volatile long static int const b = 5; /* bad practice */
```

Cualificaciones de primer nivel

```
/* "a" cannot be mutated by the program but can change as a result of external conditions */
const volatile int a = 5;

/* the const applies to array elements, i.e. "a[0]" cannot be mutated */
const int arr[] = { 1, 2, 3 };

/* for the lifetime of "ptr", no other pointer could point to the same "int" object */
int *restrict ptr;
```

Cualificaciones de subtipo de puntero

```
/* "s1" can be mutated, but "*s1" cannot */
const char *s1 = "Hello";

/* neither "s2" (because of top-level const) nor "*s2" can be mutated */
const char *const s2 = "World";

/* "*p" may change its value as a result of external conditions, "**p" and "p" cannot */
char *volatile *p;

/* "q", "*q" and "**q" may change their values as a result of external conditions */
volatile char *volatile *volatile q;
```

Examples

Variables no modificables (const)

```
const int a = 0; /* This variable is "unmodifiable", the compiler
                    should throw an error when this variable is changed */
int b = 0; /* This variable is modifiable */

b += 10; /* Changes the value of 'b' */
a += 10; /* Throws a compiler error */
```

La calificación `const` solo significa que no tenemos el derecho de cambiar los datos. No significa que el valor no pueda cambiar a nuestras espaldas.

```
_Bool doIt(double const* a) {
   double rememberA = *a;
   // do something long and complicated that calls other functions

   return rememberA == *a;
}
```

Durante la ejecución de las otras llamadas `*a` podría haber cambiado, por lo que esta función puede devolver `false` o `true` .

Advertencia

Las variables con calificación `const` aún podrían cambiarse utilizando los punteros:

```
const int a = 0;

int *a_ptr = (int*)&a; /* This conversion must be explicitly done with a cast */
*a_ptr += 10;          /* This has undefined behavior */

printf("a = %d\n", a); /* May print: "a = 10" */
```

Pero hacerlo es un error que conduce a un comportamiento indefinido. La dificultad aquí es que esto puede comportarse como se espera en ejemplos simples como este, pero luego salen mal cuando el código crece.

Variables volátiles

La palabra clave `volatile` le dice al compilador que el valor de la variable puede cambiar en cualquier momento como resultado de condiciones externas, no solo como resultado del flujo de control del programa.

El compilador no optimizará nada que tenga que ver con la variable volátil.

```
volatile int foo; /* Different ways to declare a volatile variable */
int volatile foo;

volatile uint8_t * pReg; /* Pointers to volatile variable */
uint8_t volatile * pReg;
```

Hay dos razones principales para usar variables volátiles:

- Para interactuar con el hardware que tiene registros de E / S asignados en memoria.
- Cuando se usan variables que se modifican fuera del flujo de control del programa (por ejemplo, en una rutina de servicio de interrupción)

Veamos este ejemplo:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
      // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
  quit = true;
}
```

Al compilador se le permite notar que el bucle while no modifica la variable `quit` y convierte el bucle en un bucle `while (true)` sin fin. Incluso si la variable `quit` se establece en el controlador de señal para `SIGINT` y `SIGTERM` , el compilador no lo sabe.

Declarar `quit` como `volatile` indicará al compilador que no optimice el bucle y el problema se resolverá.

El mismo problema ocurre al acceder al hardware, como vemos en este ejemplo:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

El comportamiento del optimizador es leer el valor de la variable una vez, no hay necesidad de

volver a leerlo, ya que el valor siempre será el mismo. Así terminamos con un bucle infinito. Para forzar al compilador a hacer lo que queremos, modificamos la declaración a:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Lea Calificadores de tipo en línea: https://riptutorial.com/es/c/topic/2588/calificadores-de-tipo

Capítulo 14: Campos de bits

Introducción

La mayoría de las variables en C tienen un tamaño que es un número entero de bytes. Los campos de bits son parte de una estructura que no necesariamente ocupa un número entero de bytes; Pueden cualquier número de bits. Se pueden empaquetar múltiples campos de bits en una sola unidad de almacenamiento. Forman parte del estándar C, pero hay muchos aspectos que están definidos por la implementación. Son una de las partes menos portátiles de C.

Sintaxis

- identificador de especificador de tipo: tamaño;

Parámetros

Parámetro	Descripción
especificador de tipo	`signed` , `unsigned` , `int` o `_Bool`
identificador	El nombre de este campo en la estructura.
tamaño	El número de bits a utilizar para este campo.

Observaciones

Los únicos tipos portátiles para campos de bits son `signed` , `unsigned` o `_Bool` . Se puede usar el tipo `int` simple, pero el estándar dice (§6.7.2¶5) *... para campos de bits, está definido por la implementación si el especificador `int` designa el mismo tipo que `signed int` o el mismo tipo que `unsigned int` .*

Una implementación específica puede permitir otros tipos de enteros, pero su uso no es portátil.

Examples

Campos de bits

Se puede usar un campo de bits simple para describir cosas que pueden tener un número específico de bits involucrados.

```
struct encoderPosition {
   unsigned int encoderCounts : 23;
   unsigned int encoderTurns  : 4;
   unsigned int _reserved     : 5;
```

```
};
```

En este ejemplo, consideramos un codificador con 23 bits de precisión simple y 4 bits para describir el giro múltiple. Los campos de bits se utilizan a menudo al interactuar con hardware que genera datos asociados con un número específico de bits. Otro ejemplo podría ser la comunicación con un FPGA, donde el FPGA escribe datos en su memoria en secciones de 32 bits permitiendo lecturas de hardware:

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb1On  : 1;
            unsigned int bulb2On  : 1;
            unsigned int bulb1Off : 1;
            unsigned int bulb2Off : 1;
            unsigned int jetOn    : 1;
        };
        unsigned int data;
   };
};
```

Para este ejemplo, hemos mostrado un constructo comúnmente utilizado para poder acceder a los datos en sus bits individuales, o para escribir el paquete de datos en su totalidad (emulando lo que podría hacer el FPGA). Entonces podríamos acceder a los bits como este:

```
FPGAInfo fInfo;
fInfo.data = 0xFF34F;
if (fInfo.bits.bulb1On) {
    printf("Bulb 1 is on\n");
}
```

Esto es válido, pero según el estándar C99 6.7.2.1, artículo 10:

> El orden de asignación de los campos de bits dentro de una unidad (orden alto a orden bajo o orden bajo a orden alto) está definido por la implementación.

Debe ser consciente de la endianidad al definir campos de bits de esta manera. Como tal, puede ser necesario usar una directiva de preprocesador para verificar la endianness de la máquina. Un ejemplo de esto sigue:

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
    uint8_t commFailure :1;
    uint8_t hardwareFailure :1;
    uint8_t _reserved :6;
#else
    uint8_t _reserved :6;
    uint8_t hardwareFailure :1;
    uint8_t commFailure :1;
#endif
    };
    uint8_t data;
} hardwareStatus;
```

Usando campos de bits como enteros pequeños

```
#include <stdio.h>

int main(void)
{
    /* define a small bit-field that can hold values from 0 .. 7 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* extract the right 3 bits from a value */
    unsigned int value = 255 - 2; /* Binary 11111101 */
    small.uint3 = value;          /* Binary      101 */
    printf("%d", small.uint3);

    /* This is in effect an infinite loop */
    for (small.uint3 = 0; small.uint3 < 8; small.uint3++)
    {
        printf("%d\n", small.uint3);
    }

    return 0;
}
```

Alineación de campo de bits

Los campos de bits permiten declarar campos de estructura que son más pequeños que el ancho de caracteres. Los campos de bits se implementan con una máscara de nivel de palabra o nivel de byte. El siguiente ejemplo da como resultado una estructura de 8 bytes.

```
struct C
{
    short s;            /* 2 bytes */
    char  c;            /* 1 byte */
    int   bit1 : 1;     /* 1 bit */
    int   nib  : 4;     /* 4 bits padded up to boundary of 8 bits. Thus 3 bits are padded */
    int   sept : 7;     /* 7 Bits septet, padded up to boundary of 32 bits. */
};
```

Los comentarios describen un posible diseño, pero como el estándar dice que *la alineación de la unidad de almacenamiento direccionable no está especificada* , también son posibles otros diseños.

Un campo de bits sin nombre puede ser de cualquier tamaño, pero no pueden ser inicializados o referenciados.

A un campo de bits de ancho cero no se le puede dar un nombre y alinea el siguiente campo con el límite definido por el tipo de datos del campo de bits. Esto se logra rellenando bits entre los campos de bits.

El tamaño de la estructura 'A' es de 1 byte.

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

En la estructura B, el primer campo de bits sin nombre salta 2 bits; el campo de bits de ancho cero después de `c2` hace que `c3` inicie desde el límite char (de modo que se omiten 3 bits entre `c2` y `c3` . Hay 3 bits de relleno después de `c4` . Por lo tanto, el tamaño de la estructura es de 2 bytes.

```
struct B
{
    unsigned char c1 : 1;
    unsigned char    : 2;    /* Skips 2 bits in the layout */
    unsigned char c2 : 2;
    unsigned char    : 0;    /* Causes padding up to next container boundary */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

¿Cuándo son útiles los campos de bits?

Un campo de bits se utiliza para agrupar muchas variables en un objeto, similar a una estructura. Esto permite un uso reducido de la memoria y es especialmente útil en un entorno integrado.

```
 e.g. consider the following variables having the ranges as given below.
 a --> range 0 - 3
 b --> range 0 - 1
 c --> range 0 - 7
 d --> range 0 - 1
 e --> range 0 - 1
```

Si declaramos estas variables por separado, cada una debe ser al menos un entero de 8 bits y el espacio total requerido será de 5 bytes. Además, las variables no utilizarán el rango completo de un entero sin signo de 8 bits (0-255). Aquí podemos utilizar los campos de bits.

```
typedef struct {
   unsigned int a:2;
   unsigned int b:1;
   unsigned int c:3;
   unsigned int d:1;
   unsigned int e:1;
} bit_a;
```

Se accede a los campos de bits de la estructura de la misma manera que a cualquier otra estructura. El programador debe tener cuidado de que las variables estén escritas en rango. Si está fuera de rango el comportamiento es indefinido.

```
int main(void)
{
   bit_a bita_var;
   bita_var.a = 2;              // to write into element a
```

```
   printf ("%d",bita_var.a);    // to read from element a.
   return 0;
}
```

A menudo, el programador quiere poner a cero el conjunto de campos de bits. Esto se puede hacer elemento por elemento, pero hay un segundo método. Simplemente cree una unión de la estructura anterior con un tipo sin signo que sea mayor o igual que el tamaño de la estructura. Luego, todo el conjunto de campos de bits puede ponerse a cero poniendo a cero este entero sin signo.

```
typedef union {
    struct {
       unsigned int a:2;
       unsigned int b:1;
       unsigned int c:3;
       unsigned int d:1;
       unsigned int e:1;
    };
    uint8_t data;
} union_bit;
```

El uso es el siguiente

```
int main(void)
{
   union_bit un_bit;
   un_bit.data = 0x00;        // clear the whole bit-field
   un_bit.a = 2;              // write into element a
   printf ("%d",un_bit.a);    // read from element a.
   return 0;
}
```

En conclusión, los campos de bits se usan comúnmente en situaciones de memoria restringida donde tiene muchas variables que pueden tomar rangos limitados.

No hacer para campos de bits

1. Las matrices de campos de bits, punteros a campos de bits y funciones que devuelven campos de bits no están permitidas.
2. El operador de dirección (&) no se puede aplicar a los miembros de campo de bits.
3. El tipo de datos de un campo de bits debe ser lo suficientemente ancho para contener el tamaño del campo.
4. El operador `sizeof()` no se puede aplicar a un campo de bits.
5. No hay forma de crear un `typedef` para un campo de bits aislado (aunque ciertamente puede crear un `typedef` para una estructura que contiene campos de bits).

```
typedef struct mybitfield
{
    unsigned char c1 : 20;   /* incorrect, see point 3 */
    unsigned char c2 : 4;    /* correct */
    unsigned char c3 : 1;
    unsigned int x[10]: 5;   /* incorrect, see point 1 */
```

```
} A;

int SomeFunction(void)
{
    // Somewhere in the code
    A a = { … };
    printf("Address of a.c2 is %p\n", &a.c2);      /* incorrect, see point 2 */
    printf("Size of a.c2 is %zu\n", sizeof(a.c2)); /* incorrect, see point 4 */
}
```

Lea Campos de bits en línea: https://riptutorial.com/es/c/topic/1930/campos-de-bits

Capítulo 15: Clases de almacenamiento

Introducción

Una clase de almacenamiento se usa para establecer el alcance de una variable o función. Al conocer la clase de almacenamiento de una variable, podemos determinar el tiempo de vida de esa variable durante el tiempo de ejecución del programa.

Sintaxis

- [auto | register | static | extern] <Tipo de datos> <Nombre de variable> [= <Valor>];
- [static _Thread_local | extern _Thread_local | _Thread_local] <Tipo de datos> <Nombre de variable> [= <Valor>]; / * desde> = C11 * /
- Ejemplos:
- typedef int *foo* ;
- extern int *foo* [2];

Observaciones

Los especificadores de clase de almacenamiento son las palabras clave que pueden aparecer junto al tipo de declaración de nivel superior. El uso de estas palabras clave afecta la duración del almacenamiento y la vinculación del objeto declarado, dependiendo de si se declara en el alcance del archivo o en el alcance del bloque:

Palabra clave	Duración del almacenamiento	Enlace	Observaciones
`static`	Estático	Interno	Establece el enlace interno para los objetos en el alcance del archivo; establece la duración del almacenamiento estático para objetos en el ámbito de bloque.
`extern`	Estático	Externo	Implícito y, por lo tanto, redundante para los objetos definidos en el alcance del archivo que también tienen un inicializador. Cuando se usa en una declaración en el alcance del archivo sin un inicializador, insinúa que la definición se encuentra en otra unidad de traducción y se resolverá en el momento del enlace.

Palabra clave	Duración del almacenamiento	Enlace	Observaciones
`auto`	Automático	Irrelevante	Implícito y por lo tanto redundante para objetos declarados en el ámbito de bloque.
`register`	Automático	Irrelevante	Relevante solo para objetos con duración de almacenamiento automático. Proporciona una pista de que la variable debe almacenarse en un registro. Una restricción impuesta es que uno no puede usar el operador unario `&` "dirección de" en dicho objeto, y por lo tanto el objeto no puede tener un alias.
`typedef`	Irrelevante	Irrelevante	No es un especificador de clase de almacenamiento en la práctica, pero funciona como uno desde un punto de vista sintáctico. La única diferencia es que el identificador declarado es un tipo, en lugar de un objeto.
`_Thread_local`	Hilo	Interno externo	Introducido en C11, para representar *la duración del almacenamiento de hilos* . Si se usa en el alcance del bloque, también incluirá `extern` o `static` .

Cada objeto tiene una duración de almacenamiento asociada (independientemente del alcance) y la vinculación (relevante solo para las declaraciones en el alcance del archivo), incluso cuando se omiten estas palabras clave.

El ordenamiento de los especificadores de clase de almacenamiento con respecto a los especificadores de tipo de nivel superior (`int` , `unsigned` , `short` , etc.) y los calificadores de tipo de nivel superior (`const` , `volatile`) no se aplica, por lo que ambas declaraciones son válidas:

```
int static const unsigned a = 5; /* bad practice */
static const unsigned int b = 5; /* good practice */
```

Sin embargo, se considera una buena práctica poner primero los especificadores de clase de almacenamiento, luego los calificadores de cualquier tipo, luego el especificador de tipo (`void` , `char` , `int` , `signed long` , `unsigned long long` , `long double` ...).

No todos los especificadores de clase de almacenamiento son legales en un determinado ámbito:

```
register int x; /* legal at block scope, illegal at file scope */
auto int y; /* same */

static int z; /* legal at both file and block scope */
```

```
extern int a; /* same */

extern int b = 5; /* legal and redundant at file scope, illegal at block scope */

/* legal because typedef is treated like a storage class specifier syntactically */
int typedef new_type_name;
```

Duración del almacenamiento

La duración del almacenamiento puede ser estática o automática. Para un objeto declarado, se determina según su alcance y los especificadores de clase de almacenamiento.

Duración del almacenamiento estático

Las variables con duración de almacenamiento estático se mantienen activas durante toda la ejecución del programa y se pueden declarar tanto en el alcance del archivo (con o sin `static`) como en el alcance del bloque (poniendo la `static` explícitamente). Por lo general, son asignados e inicializados por el sistema operativo al inicio del programa y se reclaman cuando finaliza el proceso. En la práctica, los formatos ejecutables tienen secciones dedicadas para dichas variables (`data` , `bss` y `rodata`) y estas secciones completas del archivo se asignan a la memoria en ciertos rangos.

Duración de almacenamiento de hilo

C11

Esta duración de almacenamiento se introdujo en C11. Esto no estaba disponible en los estándares C anteriores. Algunos compiladores proporcionan una extensión no estándar con semántica similar. Por ejemplo, gcc admite el especificador `__thread` que se puede usar en estándares C anteriores que no tenían `_Thread_local` .

Las variables con duración de almacenamiento de subprocesos se pueden declarar tanto en el alcance del archivo como en el alcance del bloque. Si se declara en el ámbito del bloque, también utilizará un especificador de almacenamiento `static` o `extern` . Su duración es la ejecución completa del *subproceso* en el que se crea. Este es el único especificador de almacenamiento que puede aparecer junto a otro especificador de almacenamiento.

Duración del almacenamiento automático

Las variables con duración de almacenamiento automático solo se pueden declarar en el alcance del bloque (directamente dentro de una función o dentro de un bloque en esa función). Solo se pueden utilizar en el período entre la entrada y la salida de la función o el bloque. Una vez que la variable queda fuera del alcance (ya sea volviendo de la función o dejando el bloque), su almacenamiento se desasigna automáticamente. Cualquier referencia adicional a la misma variable desde los punteros no es válida y conduce a un comportamiento indefinido.

En implementaciones típicas, las variables automáticas se ubican en ciertas compensaciones en el marco de pila de una función o en registros.

Enlace externo e interno

La vinculación solo es relevante para los objetos (funciones y variables) declarados en el alcance del archivo y afecta su visibilidad en diferentes unidades de traducción. Los objetos con enlace externo son visibles en todas las demás unidades de traducción (siempre que se incluya la declaración correspondiente). Los objetos con enlace interno no están expuestos a otras unidades de traducción y solo se pueden usar en la unidad de traducción donde están definidos.

Examples

typedef

Define un nuevo tipo basado en un tipo existente. Su sintaxis refleja la de una declaración de variable.

```
/* Byte can be used wherever `unsigned char` is needed */
typedef unsigned char Byte;

/* Integer is the type used to declare an array consisting of a single int */
typedef int Integer[1];

/* NodeRef is a type used for pointers to a structure type with the tag "node" */
typedef struct node *NodeRef;

/* SigHandler is the function pointer type that gets passed to the signal function. */
typedef void (*SigHandler)(int);
```

Aunque técnicamente no es una clase de almacenamiento, un compilador lo tratará como uno ya que ninguna de las otras clases de almacenamiento está permitida si se usa la palabra clave `typedef` .

Los `typedef` s son importantes y no deben sustituirse con la macro `#define` .

```
typedef int newType;
newType *ptr;        // ptr is pointer to variable of type 'newType' aka int
```

Sin embargo,

```
#define int newType
newType *ptr;        // Even though macros are exact replacements to words, this doesn't
result to a pointer to variable of type 'newType' aka int
```

auto

Esta clase de almacenamiento denota que un identificador tiene una duración de almacenamiento

automática. Esto significa que una vez que el ámbito en el que se definió el identificador termina, el objeto denotado por el identificador ya no es válido.

Dado que todos los objetos, que no viven en el ámbito global o que se declaran `static` , tienen una duración de almacenamiento automático por defecto cuando se definen, esta palabra clave es en su mayoría de interés histórico y no debe usarse:

```
int foo(void)
{
    /* An integer with automatic storage duration. */
    auto int i = 3;

    /* Same */
    int j = 5;

    return 0;
} /* The values of i and j are no longer able to be used. */
```

estático

La clase de almacenamiento `static` tiene diferentes propósitos, dependiendo de la ubicación de la declaración en el archivo:

1. Para limitar el identificador a esa unidad de traducción solamente (scope = archivo).

```
/* No other translation unit can use this variable. */
static int i;

/* Same; static is attached to the function type of f, not the return type int. */
static int f(int n);
```

2. Para guardar datos para usar con la próxima llamada de una función (alcance = bloque):

```
void foo()
{
    static int a = 0; /* has static storage duration and its lifetime is the
                       * entire execution of the program; initialized to 0 on
                       * first function call */
    int b = 0; /* b has block scope and has automatic storage duration and
                * only "exists" within function */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d\n", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }
```

```
    return 0;
}
```

Este código imprime:

```
static int a = 10, int b = 10
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10
```

Las variables estáticas conservan su valor incluso cuando se las llama desde varios subprocesos diferentes.

C99

3. Se utiliza en los parámetros de función para denotar que una matriz tiene un número mínimo constante de elementos y un parámetro no nulo:

```
/* a is expected to have at least 512 elements. */
void printInts(int a[static 512])
{
    size_t i;
    for (i = 0; i < 512; ++i)
        printf("%d\n", a[i]);
}
```

El compilador no comprueba necesariamente el número requerido de elementos (o incluso un puntero que no sea nulo), y los compiladores no tienen la obligación de notificarle de ninguna manera si no tiene suficientes elementos. Si un programador pasa menos de 512 elementos o un puntero nulo, el resultado es un comportamiento indefinido. Dado que es imposible hacer cumplir esto, se debe tener mucho cuidado al pasar un valor para ese parámetro a dicha función.

externo

Se utiliza para **declarar un objeto o función** que se define en otro lugar (y que tiene *un enlace externo*). En general, se utiliza para declarar que un objeto o una función se utiliza en un módulo que no es aquel en el que se define el objeto o función correspondiente:

```
/* file1.c */
int foo = 2;  /* Has external linkage since it is declared at file scope. */
```

```
/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` keyword refers to external definition of `foo`. */
    extern int foo;
    printf("%d\n", foo);
    return 0;
```

```
}
```

C99

Las cosas se ponen un poco más interesantes con la introducción de la palabra clave en `inline` en C99:

```
/* Should usually be place in a header file such that all users see the definition */
/* Hints to the compiler that the function `bar` might be inlined */
/* and suppresses the generation of an external symbol, unless stated otherwise. */
inline void bar(int drink)
{
    printf("You ordered drink no.%d\n", drink);
}

/* To be found in just one .c file.
   Creates an external function definition of `bar` for use by other files.
   The compiler is allowed to choose between the inline version and the external
   definition when `bar` is called. Without this line, `bar` would only be an inline
   function, and other files would not be able to call it. */
extern void bar(int);
```

registro

Indica al compilador que el acceso a un objeto debe ser lo más rápido posible. Si el compilador realmente usa la sugerencia está definido por la implementación; simplemente puede tratarlo como equivalente a `auto` .

La única propiedad que es definitivamente diferente para todos los objetos que se declaran con `register` es que no pueden calcularse sus direcciones. Por lo tanto, `register` puede ser una buena herramienta para asegurar ciertas optimizaciones:

```
register size_t size = 467;
```

es un objeto que nunca puede ser *alias* porque ningún código puede pasar su dirección a otra función en la que podría cambiarse inesperadamente.

Esta propiedad también implica que una matriz

```
register int array[5];
```

no puede descomponerse en un puntero a su primer elemento (es decir, la `array` convierte en `&array[0]`). Esto significa que no se puede acceder a los elementos de dicha matriz y que la matriz en sí no se puede pasar a una función.

De hecho, el único uso legal de una matriz declarada con una clase de almacenamiento de `register` es el operador `sizeof` ; cualquier otro operador requeriría la dirección del primer elemento de la matriz. Por esa razón, los arreglos generalmente no deben declararse con la palabra clave de `register` ya que los hace inútiles para cualquier otra cosa que no sea el cálculo de tamaño de la matriz completa, lo que puede hacerse con la misma facilidad sin la palabra clave de `register` .

La clase de almacenamiento de `register` es más apropiada para las variables que se definen dentro de un bloque y se accede a ellas con alta frecuencia. Por ejemplo,

```
/* prints the sum of the first 5 integers*/
/* code assumed to be part of a function body*/
{
    register int k, sum;
    for(k = 1, sum = 0; k < 6; sum += k, k++);
        printf("\t%d\n",sum);
}
```

C11

El operador `_Alignof` también se puede usar con matrices de `register` .

_Hilo_local

C11

Este fue un nuevo especificador de almacenamiento introducido en C11 junto con subprocesos múltiples. Esto no está disponible en los estándares C anteriores.

Indica *la duración del almacenamiento de hilos* . Una variable declarada con el especificador de almacenamiento `_Thread_local` denota que el objeto es *local para ese hilo* y su duración es la ejecución completa del hilo en el que se creó. También puede aparecer junto con `static` o `extern` .

```
#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* thread local variable i. */
    static _Thread_local int i;

    /* Prints the ID passed from main() and the address of the i.
     * Running this program will print different addresses for i, showing
     * that they are all distinct objects. */
    printf("From thread:[%d], Address of i (thread local): %p\n", *(int*)id, (void*)&i);

    return 0;
}

int main(void)
{
    thrd_t id[SIZE];
    int arr[SIZE] = {1, 2, 3, 4, 5};

    /* create 5 threads. */
    for(int i = 0; i < SIZE; i++) {
        thrd_create(&id[i], thread_func, &arr[i]);
    }

    /* wait for threads to complete. */
    for(int i = 0; i < SIZE; i++) {
        thrd_join(id[i], NULL);
```

```
        }
}
```

Lea Clases de almacenamiento en línea: https://riptutorial.com/es/c/topic/3597/clases-de-almacenamiento

Capítulo 16: Comentarios

Introducción

Los comentarios se utilizan para indicar algo a la persona que lee el código. Los comentarios son tratados como un espacio en blanco por el compilador y no cambian nada en el significado real del código. Hay dos sintaxis utilizadas para los comentarios en C, el original `/* */` y el ligeramente más nuevo `//` . Algunos sistemas de documentación utilizan comentarios con formato especial para ayudar a producir la documentación para el código.

Sintaxis

- `/*...*/`
- `//...` (solo C99 y posteriores)

Examples

/ * * / delimitado comentarios

Un comentario comienza con una barra inclinada seguida inmediatamente por un asterisco (`/*`), y termina tan pronto como se encuentra un asterisco seguido inmediatamente por una barra inclinada (`*/`). Todo lo que se encuentra entre estas combinaciones de caracteres es un comentario y el compilador lo trata como un espacio en blanco (básicamente ignorado).

```
/* this is a comment */
```

El comentario de arriba es un comentario de una sola línea. Los comentarios de este tipo `/*` pueden abarcar varias líneas, como:

```
/* this is a
multi-line
comment */
```

Aunque no es estrictamente necesario, una convención de estilo común con comentarios de varias líneas es colocar los espacios y asteriscos iniciales en las líneas posteriores a la primera, y `/*` y `*/` en las nuevas líneas, de manera que todas se alineen:

```
/*
 * this is a
 * multi-line
 * comment
 */
```

Los asteriscos adicionales no tienen ningún efecto funcional en el comentario, ya que ninguno de ellos tiene una barra inclinada relacionada.

Estos `/*` tipo de comentarios se pueden usar en su propia línea, al final de una línea de código, o incluso dentro de líneas de código:

```
/* this comment is on its own line */
if (x && y) { /*this comment is at the end of a line */
    if ((complexCondition1) /* this comment is within a line of code */
            && (complexCondition2)) {
        /* this comment is within an if, on its own line */
    }
}
```

Los comentarios no pueden ser anidados. Esto se debe a que cualquier `/*` posterior se ignorará (como parte del comentario) y la primera `*/` alcanzada se considerará como finalización del comentario. El comentario en el siguiente ejemplo *no funcionará* :

```
/* outer comment, means this is ignored => /* attempted inner comment */ <= ends the comment,
not this one => */
```

Para comentar bloques de código que contienen comentarios de este tipo, que de lo contrario se anidarían, consulte el Ejemplo de comentarios utilizando el preprocesador a continuación.

// comentarios delimitados

C99

C99 introdujo el uso de comentarios de una sola línea de estilo C ++. Este tipo de comentario comienza con dos barras diagonales y se ejecuta hasta el final de una línea:

```
// this is a comment
```

Este tipo de comentario no permite comentarios de varias líneas, aunque es posible hacer un bloque de comentarios agregando varios comentarios de una sola línea uno después de otro:

```
// each of these lines are a single-line comment
// note how each must start with
// the double forward-slash
```

Este tipo de comentario se puede usar en su propia línea o al final de una línea de código. Sin embargo, debido a que se ejecutan *hasta el final de la línea* , *no se* pueden usar dentro de una línea de código

```
// this comment is on its own line
if (x && y) { // this comment is at the end of a line
    // this comment is within an if, on its own line
}
```

Comentando utilizando el preprocesador

Los grandes fragmentos de código también se pueden "comentar" usando las directivas de

preprocesador `#if 0` y `#endif` . Esto es útil cuando el código contiene comentarios de varias líneas que de lo contrario no se anidarían.

```
#if 0 /* Starts the "comment", anything from here on is removed by preprocessor */

/* A large amount of code with multi-line comments */
int foo()
{
    /* lots of code */
    ...

    /* ... some comment describing the if statement ... */
    if (someTest) {
        /* some more comments */
        return 1;
    }

    return 0;
}

#endif /* 0 */

/* code from here on is "uncommented" (included in compiled executable) */
...
```

Posible trampa debido a los trigrafos

C99

Mientras escribe `//` comentarios delimitados, es posible cometer un error tipográfico que afecte la operación esperada. Si uno escribe:

```
int x = 20;  // Why did I do this??/
```

El `/` al final fue un error tipográfico pero ahora se interpretará en `\` . Esto es porque el `??/` forma un trigraph .

El `??/` Trigraph es en realidad una notación a mano para `\` , que es el símbolo de continuación de línea. Esto significa que el compilador piensa que la siguiente línea es una continuación de la línea actual, es decir, una continuación del comentario, que puede no ser lo que se pretende.

```
int foo = 20; // Start at 20 ??/
int bar = 0;

// The following will cause a compilation error (undeclared variable 'bar')
// because 'int bar = 0;' is part of the comment on the preceding line
bar += foo;
```

Lea Comentarios en línea: https://riptutorial.com/es/c/topic/10670/comentarios

Capítulo 17: Compilacion

Introducción

El lenguaje C es tradicionalmente un lenguaje compilado (en lugar de interpretado). El Estándar C define las ***fases de traducción*** , y el producto de su aplicación es una imagen del programa (o un programa compilado). En c11 , las fases se enumeran en §5.1.1.2.

Observaciones

Extensión de nombre de archivo	Descripción
`.c`	Archivo fuente. Por lo general contiene definiciones y código.
`.h`	Archivo de cabecera. Suele contener declaraciones.
`.o`	Archivo de objeto Código compilado en lenguaje máquina.
`.obj`	Extensión alternativa para archivos de objetos.
`.a`	Archivo de la biblioteca. Paquete de archivos de objeto.
`.dll`	Biblioteca de enlace dinámico en Windows.
`.so`	Objeto compartido (biblioteca) en muchos sistemas similares a Unix.
`.dylib`	Dynamic-Link Library en OSX (variante Unix).
`.exe` , `.com`	Archivo ejecutable de Windows. Formado por la vinculación de archivos de objetos y archivos de la biblioteca. En sistemas similares a Unix, no hay una extensión de nombre de archivo especial para el archivo ejecutable.

POSIX c99 banderas del compilador	Descripción
`-o filename`	Nombre del archivo de salida, por ejemplo. (`bin/program.exe` , `program`)
`-I directory`	búsqueda de encabezados en `direrctory` .
`-D name`	definir `name` macro
`-L directory`	búsqueda de bibliotecas en el `directory` .

POSIX c99 banderas del compilador	Descripción
`-l name`	biblioteca de enlaces `libname` .

Los compiladores en plataformas POSIX (Linux, mainframes, Mac) generalmente aceptan estas opciones, incluso si no se llaman `c99` .

- Ver también c99 - compilar programas estándar de C

Banderas GCC (Colección de compiladores GNU)	Descripción
`-Wall`	Permite que todos los mensajes de advertencia que se aceptan comúnmente sean útiles.
`-Wextra`	Habilita más mensajes de advertencia, puede ser demasiado ruidoso.
`-pedantic`	Forzar advertencias donde el código viole el estándar elegido.
`-Wconversion`	Habilite las advertencias sobre la conversión implícita, use con cuidado.
`-c`	Compila archivos fuente sin vincularlos.
`-v`	Imprime la información de la compilación.

- `gcc` acepta los indicadores POSIX y muchos otros.
- Muchos otros compiladores en plataformas POSIX (`clang` , compiladores específicos del proveedor) también usan los indicadores que se enumeran anteriormente.
- Vea también Invocar GCC para muchas más opciones.

Banderas TCC (Compilador C)	Descripción
`-Wimplicit-function-declaration`	Avisar sobre la declaración de función implícita.
`-Wunsupported`	Avisar sobre las funciones de GCC no compatibles que TCC ignora.
`-Wwrite-strings`	Haga que las constantes de cadena sean de tipo const char * en lugar de char *.
`-Werror`	Abortar la compilación si se emiten advertencias.
`-Wall`	Active todas las advertencias, excepto las `-Werror` , `-Wunusupported` y `-Wwrite strings` .

Examples

El enlazador

El trabajo del enlazador es vincular un grupo de archivos de objetos (archivos `.o`) en un ejecutable binario. El proceso de *vinculación* implica principalmente *resolver direcciones simbólicas a direcciones numéricas* . El resultado del proceso de enlace es normalmente un programa ejecutable.

Durante el proceso de enlace, el vinculador recogerá todos los módulos de objeto especificados en la línea de comandos, agregará un *código de inicio* específico del sistema al frente e intentará resolver todas *las* referencias *externas* en el módulo de objeto con *definiciones externas* en otros archivos de objeto (archivos de objeto se puede especificar directamente en la línea de comandos o se puede agregar implícitamente a través de bibliotecas). A continuación, asignará *direcciones de carga* para los archivos de objeto, es decir, especifica dónde terminarán el código y los datos en el espacio de direcciones del programa terminado. Una vez que tiene las direcciones de carga, puede reemplazar todas las direcciones simbólicas en el código del objeto con direcciones numéricas "reales" en el espacio de direcciones del objetivo. El programa está listo para ser ejecutado ahora.

Esto incluye tanto los archivos de objetos que el compilador creó a partir de los archivos de código fuente como los archivos de objetos que se han compilado previamente y que se han recopilado en archivos de biblioteca. Estos archivos tienen nombres que terminan en `.a` o `.so` , y normalmente no necesita conocerlos, ya que el enlazador sabe dónde se encuentran la mayoría de ellos y los vinculará automáticamente según sea necesario.

Invocación implícita del enlazador

Al igual que el preprocesador, el enlazador es un programa separado, a menudo llamado `ld` (pero Linux usa `collect2` , por ejemplo). Al igual que el preprocesador, el vinculador se invoca automáticamente cuando utiliza el compilador. Por lo tanto, la forma normal de utilizar el enlazador es la siguiente:

```
% gcc foo.o bar.o baz.o -o myprog
```

Esta línea le dice al compilador que vincule tres archivos de objetos (`foo.o` , `bar.o` y `baz.o`) en un archivo ejecutable binario llamado `myprog` . Ahora tiene un archivo llamado `myprog` que puede ejecutar y que, con suerte, hará algo interesante y / o útil.

Invocación explícita del enlazador

Es posible invocar el enlazador directamente, pero esto rara vez es recomendable, y suele ser muy específico de la plataforma. Es decir, las opciones que funcionan en Linux no necesariamente funcionarán en Solaris, AIX, macOS, Windows y de manera similar para cualquier otra plataforma. Si trabaja con GCC, puede usar `gcc -v` para ver qué se ejecuta en su nombre.

Opciones para el enlazador.

El enlazador también toma algunos argumentos para modificar su comportamiento. El siguiente comando le diría a gcc que vincule `foo.o` y `bar.o` , pero que también incluya la biblioteca `ncurses` .

```
% gcc foo.o bar.o -o foo -lncurses
```

Esto es en realidad (más o menos) equivalente a

```
% gcc foo.o bar.o /usr/lib/libncurses.so -o foo
```

(aunque `libncurses.so` podría ser `libncurses.a` , que es solo un archivo creado con `ar`). Tenga en cuenta que debe enumerar las bibliotecas (ya sea por nombre de ruta o mediante las opciones `-lname`) después de los archivos de objeto. Con las bibliotecas estáticas, el orden en que se especifican importa; A menudo, con bibliotecas compartidas, el orden no importa.

Tenga en cuenta que en muchos sistemas, si está utilizando funciones matemáticas (de `<math.h>`), debe especificar `-lm` para cargar la biblioteca de matemáticas, pero Mac OS X y macOS Sierra no lo requieren. Hay otras bibliotecas que son bibliotecas separadas en Linux y otros sistemas Unix, pero no en macOS - POSIX threads, y POSIX en tiempo real, y las bibliotecas de red son ejemplos. En consecuencia, el proceso de vinculación varía entre plataformas.

Otras opciones de compilación

Esto es todo lo que necesita saber para comenzar a compilar sus propios programas en C. En general, también recomendamos que use la opción de línea de comando `-Wall` :

```
% gcc -Wall -c foo.cc
```

La opción `-Wall` hace que el compilador le advierta sobre construcciones de código legales pero dudosas, y lo ayudará a detectar muchos errores muy pronto.

Si desea que el compilador le envíe más advertencias (incluidas las variables declaradas pero no utilizadas, olvidándose de devolver un valor, etc.), puede usar este conjunto de opciones, ya que `-Wall` , a pesar del nombre, no gira *todas las posibles advertencias* sobre:

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
>     -Wmissing-declarations -Wredundant-decls -Wshadow …
```

Tenga en cuenta que `clang` tiene una opción: `-Weverything` que realmente `-Weverything` todas las advertencias en `clang` .

Tipos de archivo

La compilación de programas en C requiere que trabajes con cinco tipos de archivos:

1. **Archivos de origen** : estos archivos contienen definiciones de funciones y tienen nombres

que terminan en `.c` por convención. Nota: `.cc` y `.cpp` son archivos C ++; *no* archivos C.
por ejemplo, `foo.c`

2. **Archivos de encabezado** : estos archivos contienen prototipos de funciones y varias declaraciones del preprocesador (ver más abajo). Se utilizan para permitir que los archivos de código fuente accedan a funciones definidas externamente. Los archivos de encabezado terminan en `.h` por convención.
por ejemplo, `foo.h`

3. **Archivos de objeto** : estos archivos se producen como la salida del compilador. Consisten en definiciones de funciones en forma binaria, pero no son ejecutables por sí mismas. Los archivos de objetos terminan en `.o` por convención, aunque en algunos sistemas operativos (por ejemplo, Windows, MS-DOS), a menudo terminan en `.obj` .
por ejemplo, `foo.o` `foo.obj`

4. **Ejecutables binarios** : estos se producen como la salida de un programa llamado "vinculador". El vinculador vincula una serie de archivos de objeto para producir un archivo binario que se puede ejecutar directamente. Los ejecutables binarios no tienen un sufijo especial en los sistemas operativos Unix, aunque generalmente terminan en `.exe` en Windows.
por ejemplo, `foo` `foo.exe`

5. **Bibliotecas** : una biblioteca es un binario compilado, pero no es en sí un ejecutable (es decir, no hay una función `main()` en una biblioteca). Una biblioteca contiene funciones que pueden ser utilizadas por más de un programa. Una biblioteca debe enviarse con archivos de encabezado que contengan prototipos para todas las funciones de la biblioteca; estos archivos de encabezado deben ser referenciados (por ejemplo, `#include <library.h>`) en cualquier archivo fuente que use la biblioteca. El enlazador entonces debe ser referido a la biblioteca para que el programa pueda compilarse exitosamente. Hay dos tipos de bibliotecas: estáticas y dinámicas.

   - **Biblioteca estática** : una biblioteca estática (archivos `.a` para sistemas POSIX y archivos `.lib` para Windows, que no debe confundirse con los archivos de biblioteca de importación DLL , que también utilizan la extensión `.lib`) está incorporada estáticamente en el programa. Las bibliotecas estáticas tienen la ventaja de que el programa sabe exactamente qué versión de una biblioteca se utiliza. Por otro lado, los tamaños de los ejecutables son más grandes, ya que se incluyen todas las funciones de biblioteca utilizadas.
   por ejemplo, `libfoo.a` `foo.lib`
   - **Biblioteca dinámica** : una biblioteca dinámica (archivos `.so` para la mayoría de los sistemas POSIX, `.dylib` para OSX y archivos `.dll` para Windows) está vinculada dinámicamente en tiempo de ejecución por el programa. En ocasiones, también se hace referencia a estas bibliotecas como bibliotecas compartidas, ya que muchos programas pueden compartir una imagen de biblioteca. Las bibliotecas dinámicas tienen la ventaja de ocupar menos espacio en disco si más de una aplicación está utilizando la biblioteca. Además, permiten actualizaciones de la biblioteca (corrección de errores) sin tener que reconstruir archivos ejecutables.
   por ejemplo, `foo.so` `foo.dylib` `foo.dll`

El preprocesador

Antes de que el compilador de C comience a compilar un archivo de código fuente, el archivo se procesa en una fase de preprocesamiento. Esta fase se puede realizar mediante un programa separado o se puede integrar completamente en un ejecutable. En cualquier caso, el compilador lo invoca automáticamente antes de que comience la compilación propiamente dicha. La fase de preprocesamiento convierte su código fuente en otro código fuente o unidad de traducción mediante la aplicación de reemplazos textuales. Puede considerarlo como un código fuente "modificado" o "expandido". Esa fuente expandida puede existir como un archivo real en el sistema de archivos, o solo puede almacenarse en la memoria por un corto tiempo antes de seguir procesándose.

Los comandos del preprocesador comienzan con el signo de número ("#"). Hay varios comandos de preprocesador; Dos de los más importantes son:

1. **Define** :

   `#define` se usa principalmente para definir constantes. Por ejemplo,

   ```
   #define BIGNUM 1000000
   int a = BIGNUM;
   ```

   se convierte en

   ```
   int a = 1000000;
   ```

   `#define` se usa de esta manera para evitar tener que escribir explícitamente algún valor constante en muchos lugares diferentes en un archivo de código fuente. Esto es importante en caso de que necesite cambiar el valor constante más adelante; es mucho menos propenso a errores cambiarlo una vez, en `#define` , que tener que cambiarlo en varios lugares dispersos por todo el código.

   Como `#define` solo hace una búsqueda avanzada y reemplaza, también puede declarar macros. Por ejemplo:

   ```
   #define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
   // in the function:
   a = x;
   ISTRUE(a);
   ```

   se convierte en:

   ```
   // in the function:
   a = x;
   do {
       a = a ? 1 : 0;
   } while(0);
   ```

   En la primera aproximación, este efecto es aproximadamente el mismo que con las

funciones en línea, pero el preprocesador no proporciona la comprobación de tipos para las macros `#define` . Se sabe que esto es propenso a errores y su uso requiere una gran precaución.

También tenga en cuenta que el preprocesador también reemplazará los comentarios con un espacio en blanco como se explica a continuación.

2. **Incluye** :

   `#include` se usa para acceder a definiciones de funciones definidas fuera de un archivo de código fuente. Por ejemplo:

   ```
   #include <stdio.h>
   ```

   hace que el preprocesador pegue el contenido de `<stdio.h>` en el archivo de código fuente en la ubicación de la instrucción `#include` antes de que se compile. `#include` casi siempre se usa para incluir archivos de encabezado, que son archivos que contienen principalmente declaraciones de funciones y `#define` sentencias. En este caso, utilizamos `#include` para poder utilizar funciones como `printf` y `scanf` , cuyas declaraciones se encuentran en el archivo `stdio.h` . Los compiladores de C no le permiten usar una función a menos que se haya declarado o definido previamente en ese archivo; Por lo tanto, las declaraciones `#include` son la forma de reutilizar el código escrito previamente en sus programas C.

3. **Operaciones lógicas** :

   ```
   #if defined A || defined B
   variable = another_variable + 1;
   #else
   variable = another_variable * 2;
   #endif
   ```

   se cambiará a:

   ```
   variable = another_variable + 1;
   ```

   si A o B se definieron en algún lugar del proyecto antes. Si este no es el caso, por supuesto, el preprocesador hará esto:

   ```
   variable = another_variable * 2;
   ```

   Esto se usa a menudo para el código, que se ejecuta en diferentes sistemas o compila en diferentes compiladores. Ya que hay definiciones globales, que son específicas del compilador / sistema, puede probarlas y dejar que el compilador solo use el código que compilará con seguridad.

4. **Comentarios**

   El preprocesador reemplaza todos los comentarios en el archivo fuente por espacios individuales. Los comentarios se indican con `//` hasta el final de la línea, o una combinación

de apertura `/*` y cierre `*/` comentario entre paréntesis.

El compilador

Después de que el preprocesador C haya incluido todos los archivos de encabezado y expandido todas las macros, el compilador puede compilar el programa. Lo hace convirtiendo el código fuente de C en un archivo de código objeto, que es un archivo que termina en `.o` que contiene la versión binaria del código fuente. Sin embargo, el código objeto no es ejecutable directamente. Para hacer un ejecutable, también debe agregar código para todas las funciones de biblioteca que fueron `#include` d en el archivo (esto no es lo mismo que incluir las declaraciones, que es lo que hace `#include`). Este es el trabajo del enlazador .

En general, la secuencia exacta de cómo invocar un compilador de C depende mucho del sistema que esté utilizando. Aquí estamos usando el compilador GCC, aunque se debe tener en cuenta que existen muchos más compiladores:

```
% gcc -Wall -c foo.c
```

`%` es el símbolo del sistema operativo. Esto le indica al compilador que ejecute el preprocesador en el archivo `foo.c` y luego lo compile en el archivo de código de objeto `foo.o` La opción `-c` significa compilar el archivo de código fuente en un archivo objeto pero no invocar el enlazador. Esta opción `-c` está disponible en sistemas POSIX, como Linux o macOS; Otros sistemas pueden usar una sintaxis diferente.

Si su programa completo está en un archivo de código fuente, puede hacer esto:

```
% gcc -Wall foo.c -o foo
```

Esto le dice al compilador que ejecute el preprocesador en `foo.c` , lo compila y luego lo vincula para crear un ejecutable llamado `foo` . La opción `-o` indica que la siguiente palabra en la línea es el nombre del archivo ejecutable binario (programa). Si no especifica la `-o` , (si solo escribe `gcc foo.c`), el ejecutable se llamará `a.out` por razones históricas.

En general, el compilador toma cuatro pasos al convertir un archivo `.c` en un ejecutable:

1. **preprocesamiento** : expande textualmente las directivas `#include` y `#define` macros en su archivo `.c`
2. **compilación** : convierte el programa en ensamblaje (puede detener el compilador en este paso agregando la opción `-S`)
3. **ensamblaje** - convierte el ensamblaje en código de máquina
4. **vinculación** : vincula el código objeto a bibliotecas externas para crear un ejecutable

Tenga en cuenta también que el nombre del compilador que estamos utilizando es GCC, que significa "compilador GNU C" y "colección de compiladores GNU", según el contexto. Existen otros compiladores de C. Para sistemas operativos similares a Unix, muchos de ellos tienen el nombre `cc` , para "compilador C", que a menudo es un enlace simbólico a algún otro compilador. En sistemas Linux, `cc` es a menudo un alias para GCC. En macOS o OS-X, apunta a clang.

Los estándares POSIX actualmente exigen a `c99` como el nombre de un compilador de C: es compatible con el estándar C99 de forma predeterminada. Las versiones anteriores de POSIX exigían `c89` como compilador. POSIX también exige que este compilador entienda las opciones `-c` y `-o` que usamos anteriormente.

Nota: La opción `-Wall` presente en ambos ejemplos de `gcc` le dice al compilador que imprima advertencias sobre construcciones cuestionables, lo cual es muy recomendable. También es una buena idea agregar otras opciones de advertencia , por ejemplo, `-Wextra` .

Las fases de traducción

A partir de la norma C 2011, enumerada en *§5.1.1.2 Fases de traducción* , la traducción del código fuente a la imagen del programa (p. Ej., El archivo ejecutable) se indica en 8 pasos ordenados.

1. La entrada del archivo de origen se asigna al conjunto de caracteres de origen (si es necesario). Trigraphs se reemplazan en este paso.
2. Las líneas de continuación (líneas que terminan con `\`) se empalman con la siguiente línea.
3. El código fuente se analiza en espacios en blanco y tokens de preprocesamiento.
4. Se aplica el preprocesador, que ejecuta directivas, expande macros y aplica pragmas. Cada archivo fuente obtenido por `#include` pasa por las fases de traducción 1 a 4 (recursivamente si es necesario). Todas las directivas relacionadas con el preprocesador se eliminan.
5. Los valores del conjunto de caracteres de origen en las constantes de caracteres y los literales de cadena se asignan al conjunto de caracteres de ejecución.
6. Los literales de cuerdas adyacentes entre sí están concatenados.
7. El código fuente se analiza en tokens, que comprenden la unidad de traducción.
8. Las referencias externas se resuelven y se forma la imagen del programa.

Una implementación de un compilador de C puede combinar varios pasos juntos, pero la imagen resultante todavía debe comportarse como si los pasos anteriores hubieran ocurrido por separado en el orden indicado anteriormente.

Lea Compilacion en línea: https://riptutorial.com/es/c/topic/1337/compilacion

Capítulo 18: Comportamiento definido por la implementación

Observaciones

Visión general

El estándar de C describe la sintaxis del lenguaje, las funciones proporcionadas por la biblioteca estándar y el comportamiento de los procesadores de C conformes (en términos generales, compiladores) y los programas de C conformes. Con respecto al comportamiento, el estándar en su mayor parte especifica comportamientos particulares para programas y procesadores. Por otro lado, algunas operaciones tienen *un comportamiento indefinido* explícito o implícito, tales operaciones siempre deben evitarse, ya que no se puede confiar en nada sobre ellas. En el medio, hay una variedad de comportamientos *definidos* de *implementación* . Estos comportamientos pueden variar entre los procesadores de C, los tiempos de ejecución y las bibliotecas estándar (colectivamente, *implementaciones*), pero son consistentes y confiables para cualquier implementación dada, y las implementaciones correspondientes documentan su comportamiento en cada una de estas áreas.

A veces es razonable que un programa se base en un comportamiento definido por la implementación. Por ejemplo, si el programa es de alguna manera específico para un entorno operativo en particular, es poco probable que el hecho de depender de comportamientos generales definidos por la implementación de los procesadores comunes para ese entorno sea un problema. Alternativamente, uno puede usar directivas de compilación condicional para seleccionar comportamientos definidos por la implementación apropiados para la implementación en uso. En cualquier caso, es esencial saber qué operaciones tienen un comportamiento definido de implementación, ya sea para evitarlas o para tomar una decisión informada sobre si y cómo usarlas.

El balance de estos comentarios constituye una lista de todos los comportamientos y características definidos por la implementación especificados en el estándar C2011, con referencias al estándar. Muchos de ellos utilizan la terminología de la norma . Algunos otros se basan más generalmente en el contexto del estándar, como las ocho etapas de la traducción del código fuente en un programa, o la diferencia entre las implementaciones alojadas y las independientes. Algunos que pueden ser particularmente sorprendentes o notables se presentan en letra negrita. No todos los comportamientos descritos están respaldados por estándares C anteriores, pero en general, tienen comportamientos definidos por la implementación en todas las versiones del estándar que los admiten.

Programas y Procesadores

General

- **El número de bits en un byte** (3.6 / 3). Al menos `8` , el valor real se puede consultar con la macro `CHAR_BIT` .

- Qué mensajes de salida se consideran "mensajes de diagnóstico" (3.10 / 1)

Fuente de traducción

- La manera en que los caracteres multibyte del archivo de origen físico se asignan al conjunto de caracteres de origen (5.1.1.2/1).

- Si las secuencias no vacías de espacios en blanco que no son de nueva línea se reemplazan por espacios individuales durante la fase de traducción 3 (5.1.1.2/1)

- Los caracteres del conjunto de ejecución a los que se convierten los literales de caracteres y los caracteres en constantes de cadena (durante la fase de traducción 5) cuando, de lo contrario, no hay ningún carácter correspondiente (5.1.1.2/1).

Entorno operativo

- La manera en que se identifican los mensajes de diagnóstico a emitir (5.1.1.3/1).

- El nombre y el tipo de la función llamada al inicio en una implementación independiente (5.1.2.1/1).

- Qué recursos de biblioteca están disponibles en una implementación independiente, más allá de un conjunto mínimo especificado (5.1.2.1/1).

- El efecto de la terminación del programa en un entorno independiente (5.1.2.1/2).

- En un entorno alojado, cualquier firma permitida para la función `main()` que no sea `int main(int argc, char *arg[])` e `int main(void)` (5.1.2.2.1 / 1).

- La manera en que una implementación alojada define las cadenas señaladas por el segundo argumento a `main()` (5.1.2.2.1 / 2).

- Lo que constituye un "dispositivo interactivo" para los fines de las secciones 5.1.2.3 (Ejecución del programa) y 7.21.3 (Archivos) (5.1.2.3/7).

- Cualquier restricción en los objetos a los que hacen referencia las rutinas de manejo de interrupciones en una implementación de optimización (5.1.2.3/10).

- En una implementación independiente, si se admiten múltiples subprocesos de ejecución (5.1.2.4/1).

- Los valores de los miembros del conjunto de caracteres de ejecución (5.2.1 / 1).

- Los valores de `char` correspondientes a las secuencias de escape alfabéticas definidas (5.2.2 / 3).

- **Límites y características de enteros y puntos flotantes** (5.2.4.2/1).

- La precisión de las operaciones aritméticas de punto flotante y de las conversiones de la biblioteca estándar de representaciones internas de punto flotante a representaciones de cadena (5.2.4.2.2 / 6).
- El valor de la macro `FLT_ROUNDS` , que codifica el modo de redondeo de punto flotante predeterminado (5.2.4.2.2 / 8).
- Los comportamientos de redondeo caracterizados por valores soportados de `FLT_ROUNDS` mayores que 3 o menores que -1 (5.2.4.2.2 / 8).
- El valor de la macro `FLT_EVAL_METHOD` , que caracteriza el comportamiento de evaluación de punto flotante (5.2.4.2.2 / 9).
- El comportamiento se caracteriza por cualquier valor admitido de `FLT_EVAL_METHOD` menor que -1 (5.2.4.2.2 / 9).
- Los valores de las macros `FLT_HAS_SUBNORM` , `DBL_HAS_SUBNORM` y `LDBL_HAS_SUBNORM` , que caracterizan si los formatos de punto flotante estándar admiten números subnormales (5.2.4.2.2 / 10)

Los tipos

- El resultado de intentar (indirectamente) acceder a un objeto con una duración de almacenamiento de subprocesos desde un subproceso diferente al que está asociado con el objeto (6.2.4 / 4)
- El valor de un `char` al que se ha asignado un carácter fuera del conjunto de ejecución básico (6.2.5 / 3).
- Los tipos enteros con signo extendido admitidos, si existen, (6.2.5 / 4), y cualquier palabra clave de extensión utilizada para identificarlos.
- **Si `char` tiene la misma representación y comportamiento que `signed char` o `unsigned char`** (6.2.5 / 15). Se puede consultar con `CHAR_MIN` , que es `0` o `SCHAR_MIN` si `char` no está firmado o firmado, respectivamente.
- **El número, orden y codificación de los bytes en las representaciones de objetos** , excepto donde se especifique explícitamente en la norma (6.2.6.1/2).
- **Cuál de las tres formas reconocidas de representación de enteros se aplica en cualquier situación dada, y si ciertos patrones de bits de objetos enteros son representaciones de trampas** (6.2.6.2/2).
- El requisito de alineación de cada tipo (6.2.8 / 1).
- Si y en qué contextos se admiten las alineaciones extendidas (6.2.8 / 3).
- El conjunto de alineaciones extendidas soportadas (6.2.8 / 4).
- Los rangos de conversión de enteros de cualquier tipo de enteros con signo extendido

relacionados entre sí (6.3.1.1/1).

- **El efecto de asignar un valor fuera de rango a un entero con signo** (6.3.1.3/3).
- Cuando se asigna un valor dentro del rango pero no representable a un objeto de punto flotante, cómo se elige el valor representable almacenado en el objeto entre los dos valores representables más cercanos (6.3.1.4/2 ; 6.3.1.5/1 ; 6.4.4.2 / 3).
- **El resultado de convertir un entero en un tipo de puntero** , excepto las expresiones constantes de enteros con valor `0` (6.3.2.3/5).

Forma de fuente

- Las ubicaciones dentro de las directivas `#pragma` donde se reconocen los tokens de nombre de encabezado (6.4 / 4).
- Los caracteres, incluidos los caracteres multibyte, excepto el subrayado, las letras latinas sin acento, los nombres de caracteres universales y los dígitos decimales que pueden aparecer en los identificadores (6.4.2.1/1).
- **El número de caracteres significativos en un identificador** (6.4.2.1/5).
- Con algunas excepciones, la manera en que los caracteres de origen en una constante de caracteres enteros se asignan a los caracteres del conjunto de ejecución (6.4.4.4/2 ; 6.4.4.4/10).
- La configuración regional actual utilizada para calcular el valor de una constante de carácter ancho y la mayoría de los demás aspectos de la conversión para muchas de estas constantes (6.4.4.4/11).
- Si se pueden concatenar tokens literales de cadena ancha con prefijos diferentes y, de ser así, el tratamiento de la secuencia de caracteres multibyte resultante (6.4.5 / 5)
- La configuración regional utilizada durante la fase de traducción 7 para convertir los literales de cadena ancha en secuencias de caracteres multibyte, y su valor cuando el resultado no se puede representar en el conjunto de caracteres de ejecución (6.4.5 / 6).
- La forma en que los nombres de encabezado se asignan a los nombres de archivo (6.4.7 / 2).

Evaluación

- Si las expresiones de punto flotante se contratan cuando no se usa `FP_CONTRACT` (6.5 / 8).
- **Los valores de los resultados de los operadores `sizeof` y `_Alignof`** (6.5.3.4/5).
- El tamaño del tipo de resultado de la resta del puntero (6.5.6 / 9).
- **El resultado de desplazar a la derecha un entero con signo con un valor negativo** (6.5.7 / 5).

Comportamiento en tiempo de ejecución

- La medida en que la palabra clave de `register` es efectiva (6.7.1 / 6).
- Si el tipo de campo de bits declarado como `int` es el mismo tipo que `unsigned int` o como `signed int` (6.7.2 / 5).
- ¿Qué tipos de campos de bits pueden tomar, además de `_Bool` calificados, `signed int` y `unsigned int` ; si los campos de bits pueden tener tipos atómicos (6.7.2.1/5).
- Aspectos de cómo las implementaciones diseñan el almacenamiento para campos de bits (6.7.2.1/11).
- La alineación de los miembros que no son de campo de bits de estructuras y uniones (6.7.2.1/14).
- El tipo subyacente para cada tipo enumerado (6.7.2.2/4).
- Lo que constituye un "acceso" a un objeto de tipo calificado `volatile` (6.7.3 / 7).
- La efectividad de las declaraciones de funciones en `inline` (6.7.4 / 6).

Preprocesador

- Si las constantes de caracteres se convierten a valores enteros de la misma manera en las condicionales de preprocesador que en las expresiones ordinarias, y si una constante de un solo carácter puede tener un valor negativo (6.10.1 / 4).
- Las ubicaciones buscaron los archivos designados en una directiva `#include` (6.10.2 / 2-3).
- La forma en que se forma un nombre de encabezado a partir de los tokens de una directiva multiincendio `#include` (6.10.2 / 4).
- El límite para `#include` anidación (6.10.2 / 6).
- Si un `\` carácter se inserta antes de `\` introducir un nombre de carácter universal en el resultado del operador `#` del preprocesador (6.10.3.2/2).
- El comportamiento de la directiva de preprocesamiento `#pragma` para pragmas distintos de `STDC` (6.10.6 / 1).
- El valor de las macros `__DATE__` y `__TIME__` si no hay fecha u hora de traducción, respectivamente, está disponible (6.10.8.1/1).
- La codificación de caracteres interna utilizada para `wchar_t` si la macro `__STDC_ISO_10646__` no está definida (6.10.8.2/1).
- La codificación de caracteres interna utilizada para `char32_t` si la macro `__STDC_UTF_32__` no está definida (6.10.8.2/1).

Biblioteca estándar

General

- El formato de los mensajes emitidos cuando fallan las aserciones (7.2.1.1/2).

Funciones de entorno de punto flotante

- Cualquier excepción adicional de punto flotante más allá de las definidas por el estándar (7.6 / 6).
- Cualquier modo de redondeo de punto flotante adicional más allá de los definidos por el estándar (7.6 / 8).
- Cualquier entorno de punto flotante adicional más allá de los definidos por el estándar (7.6 / 10).
- El valor predeterminado del conmutador de acceso de entorno de punto flotante (7.6.1 / 2).
- La representación de los indicadores de estado de punto flotante registrados por `fegetexceptflag()` (7.6.2.2/1).
- Si la función `feraiseexcept()` aumenta adicionalmente la excepción de punto flotante "inexacta" cada vez que genera la excepción de punto flotante de "desbordamiento" o "subdesbordamiento" (7.6.2.3/2).

Funciones relacionadas con la localización

- Las cadenas de locale que no sean `"C"` compatibles con `setlocale()` (7.11.1.1/3).

Funciones matematicas

- Los tipos representados por `float_t` y `double_t` cuando la macro `FLT_EVAL_METHOD` tiene un valor diferente de `0` , `1` y `2` (7.12 / 2).
- Cualquier clasificación de punto flotante admitida más allá de las definidas por el estándar (7.12 / 6).
- El valor devuelto por las funciones `math.h` en el caso de un error de dominio (7.12.1 / 2).
- El valor devuelto por las funciones `math.h` en el caso de un error de polo (7.12.1 / 3).
- El valor devuelto por las funciones `math.h` cuando el resultado se desborda, y los aspectos de si `errno` se establece en `ERANGE` y si se `ERANGE` una excepción de punto flotante en esas circunstancias (7.12.1 / 6).
- El valor predeterminado del interruptor de contracción FP (7.12.2 / 2).
- Si las funciones `fmod()` devuelven 0 o `fmod()` un error de dominio cuando su segundo

argumento es 0 (7.12.10.1/3).

- Si las funciones `remainder()` devuelven 0 o generan un error de dominio cuando su segundo argumento es 0 (7.12.10.2/3).
- El número de bits significativos en los módulos de cociente calculados por las funciones `remquo()` (7.12.10.3/2).
- Si las funciones `remquo()` devuelven 0 o `remquo()` un error de dominio cuando su segundo argumento es 0 (7.12.10.3/3).

Señales

- El conjunto completo de señales admitidas, su semántica y su manejo predeterminado (7.14 / 4).
- Cuando se genera una señal y hay un controlador personalizado asociado con esa señal, las señales, si las hay, se bloquean durante la ejecución del controlador (7.14.1.1/3).
- Las señales que no sean `SIGFPE` , `SIGILL` y `SIGSEGV` causan que el comportamiento al regresar de un controlador de señal personalizado no esté definido (7.14.1.1/3).
- Las señales que inicialmente están configuradas para ser ignoradas (independientemente de su manejo predeterminado; 7.14.1.1/6).

Diverso

- La constante de puntero nulo específica a la que se expande la macro `NULL` (7.19 / 3).

Funciones de manejo de archivos.

- Si la última línea de un flujo de texto requiere una nueva línea de terminación (7.21.2 / 2).
- El número de caracteres nulos se anexa automáticamente a una secuencia binaria (7.21.2 / 3).
- La posición inicial de un archivo abierto en modo de adición (7.21.3 / 1).
- Si una escritura en una secuencia de texto hace que la secuencia se trunca (7.21.3 / 2).
- Soporte para buffering de flujo (7.21.3 / 3).
- Si existen realmente archivos de longitud cero (7.21.3 / 4).
- Las reglas para componer nombres de archivos válidos (7.21.3 / 8).
- Si el mismo archivo se puede abrir simultáneamente varias veces (7.21.3 / 8).
- La naturaleza y la elección de la codificación para caracteres multibyte (7.21.3 / 10).

- El comportamiento de la función `remove()` cuando el archivo de destino está abierto (7.21.4.1/2).
- El comportamiento de la función `rename()` cuando el archivo de destino ya existe (7.21.4.2/2).
- Si los archivos creados a través de la función `tmpfile()` se eliminan en caso de que el programa finalice de manera anormal (7.21.4.3/2).
- Qué modo cambia en qué circunstancias se permiten a través de `freopen()` (7.21.5.4/3).

Funciones de E / S

- ¿Cuál de las representaciones permitidas de los valores infinitos y no de número de PF son producidos por las funciones de familia printf () - (7.21.6.1/8).
- La forma en que los punteros son formateados por las funciones de familia `printf()` (7.21.6.1/8).
- El comportamiento de `scanf()` -family funciona cuando el carácter `-` aparece en una posición interna de la lista de escaneo de un `[` campo (7.21.6.2/12).
- La mayoría de los aspectos de las funciones de `scanf()` -family 'manejan los campos `p` (7.21.6.2/12).
- El valor `errno` establecido por `fgetpos()` en caso de error (7.21.9.1/2).
- El valor `errno` establecido por `fsetpos()` en caso de fallo (7.21.9.3/2).
- El valor `errno` establecido por `ftell()` en caso de fallo (7.21.9.4/3).
- El significado de las funciones de familia `strtod()` de algunos aspectos soportados de un formato NaN (7.22.1.3p4).
- Si las funciones de familia `strtod()` configuran `errno` en `ERANGE` cuando el resultado se desborda (7.22.1.3/10).

Funciones de asignación de memoria

- El comportamiento de las funciones de asignación de memoria cuando el número de bytes solicitados es 0 (7.22.3 / 1).

Funciones del entorno del sistema

- Qué limpiezas, si las hay, se realizan y qué estado se devuelve al sistema operativo del host cuando se llama a la función `abort()` (7.22.4.1/2).
- Qué estado se devuelve al entorno de host cuando se llama a `exit()` (7.22.4.4/5).
- El manejo de las secuencias abiertas y el estado que se devuelve al entorno del host

cuando se llama `_Exit()` (7.22.4.5/2).

- El conjunto de nombres de entorno accesibles a través de `getenv()` y el método para modificar el entorno (7.22.4.6/2).
- El valor de retorno de la función `system()` (7.22.4.8/3).

Funciones de fecha y hora.

- La zona horaria local y el horario de verano (7.27.1 / 1).
- El rango y la precisión de los tiempos se pueden representar a través de los tipos `clock_t` y `time_t` (7.27.1 / 4).
- El comienzo de la era que sirve como referencia para los tiempos devueltos por la función `clock()` (7.27.2.1/3).
- El comienzo de la época que sirve como referencia para los tiempos devueltos por la función `timespec_get()` (cuando la base de tiempo es `TIME_UTC` ; 7.27.2.5/3).
- El reemplazo de `strftime()` para el especificador de conversión `%Z` en la configuración regional "C" (7.27.3.5/7).

Funciones de E / S de caracteres anchos

- ¿Cuál de las representaciones permitidas de los valores de PF infinitos y sin número se producen mediante las funciones de familia `wprintf()` (7.29.2.1/8)?
- La forma en que los punteros son formateados por las funciones de familia `wprintf()` (7.29.2.1/8).
- El comportamiento de `wscanf()` -family funciona cuando el carácter `-` aparece en una posición interna de la lista de exploración de un `[` campo (7.29.2.2/12).
- La mayoría de los aspectos de las `wscanf()` de la `wscanf()` entregan los campos `p` (7.29.2.2/12).
- El significado de las funciones de familia `wstrtod()` de algunos aspectos soportados del formato NaN (7.29.4.1.1 / 4).
- Si las funciones de familia `wstrtod()` configuran `errno` en `ERANGE` cuando el resultado se desborda (7.29.4.1.1 / 10).

Examples

Desplazamiento a la derecha de un entero negativo

```
int signed_integer = -1;
```

```
// The right shift operation exhibits implementation-defined behavior:
int result = signed_integer >> 1;
```

Asignar un valor fuera de rango a un entero

```
// Supposing SCHAR_MAX, the maximum value that can be represented by a signed char, is
// 127, the behavior of this assignment is implementation-defined:
signed char integer;
integer = 128;
```

Asignación de cero bytes

```
// The allocation functions have implementation-defined behavior when the requested size
// of the allocation is zero.
void *p = malloc(0);
```

Representación de enteros con signo

Cada tipo entero con signo se puede representar en uno de los tres formatos; está definido por la implementación cuál se usa. La implementación en uso para cualquier tipo de entero con signo dado al menos tan ancho como `int` puede determinarse en tiempo de ejecución a partir de los dos bits de orden más bajo de la representación del valor `-1` en ese tipo, así:

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };
#define SIGN_REP(T) ((T)-1 & (T)3)

switch (SIGN_REP(long)) {
   case sign_magnitude: { /* do something */ break; }
   case ones_compl:     { /* do otherwise */ break; }
   case twos_compl:     { /* do yet else  */ break; }
   case 0:  { _Static_assert(SIGN_REP(long), "bogus sign representation"); }
}
```

El mismo patrón se aplica a la representación de tipos más estrechos, pero esta técnica no puede probarlos porque los operandos de `&` están sujetos a "las conversiones aritméticas habituales" antes de que se calcule el resultado.

Lea Comportamiento definido por la implementación en línea: https://riptutorial.com/es/c/topic/4832/comportamiento-definido-por-la-implementacion

Capítulo 19: Comportamiento indefinido

Introducción

En C, algunas expresiones producen *un comportamiento indefinido* . El estándar elige explícitamente no definir cómo debe comportarse un compilador si encuentra tal expresión. Como resultado, un compilador es libre de hacer lo que crea conveniente y puede producir resultados útiles, resultados inesperados o incluso fallar.

El código que invoca a UB puede funcionar según lo previsto en un sistema específico con un compilador específico, pero es probable que no funcione en otro sistema, o con un compilador diferente, una versión del compilador o la configuración del compilador.

Observaciones

¿Qué es el comportamiento indefinido (UB)?

Comportamiento indefinido es un término usado en el estándar C. El estándar C11 (ISO / IEC 9899: 2011) define el término comportamiento indefinido como

> comportamiento, en el uso de un constructo de programa no portátil o erróneo o de datos erróneos, para los cuales esta Norma Internacional no impone requisitos

¿Qué pasa si hay UB en mi código?

Estos son los resultados que pueden ocurrir debido a un comportamiento indefinido según el estándar:

> NOTA El posible comportamiento indefinido abarca desde ignorar la situación completamente con resultados impredecibles, hasta comportarse durante la traducción o la ejecución del programa de una manera documentada característica del entorno (con o sin la emisión de un mensaje de diagnóstico), hasta terminar una traducción o ejecución (con la emisión de un mensaje de diagnóstico).

La siguiente cita se usa a menudo para describir (de manera menos formal) los resultados que suceden a partir de un comportamiento indefinido:

> "Cuando el compilador se encuentra con [una construcción indefinida dada] es legal que haga que los demonios salgan volando de tu nariz" (la implicación es que el compilador puede elegir cualquier manera arbitrariamente extraña de interpretar el código sin violar el estándar ANSI C)

¿Por qué existe UB?

Si es tan malo, ¿por qué no lo definieron o lo hicieron definido por la implementación?

El comportamiento indefinido permite más oportunidades de optimización; El compilador puede

suponer justificadamente que cualquier código no contiene un comportamiento indefinido, lo que puede permitirle evitar verificaciones en tiempo de ejecución y realizar optimizaciones cuya validez sería costosa o imposible de demostrar de otro modo.

¿Por qué es difícil localizar a UB?

Hay al menos dos razones por las que un comportamiento indefinido crea errores que son difíciles de detectar:

- No se requiere que el compilador, y generalmente no puede advertirle, sobre un comportamiento indefinido. De hecho, exigir que lo haga iría directamente en contra de la razón de la existencia de un comportamiento indefinido.
- Es posible que los resultados impredecibles no comiencen a desplegarse en el punto exacto de la operación donde se produce la construcción cuyo comportamiento no está definido; El comportamiento no definido mancha toda la ejecución y sus efectos pueden ocurrir en cualquier momento: durante, después o incluso *antes de* la construcción indefinida.

Considere la posibilidad de no hacer referencia al puntero nulo: el compilador no está obligado a diagnosticar la falta de referencia al puntero nulo, e incluso no podría hacerlo, ya que, en el tiempo de ejecución, cualquier puntero pasado a una función o en una variable global podría ser nulo. *Y cuando se produce la anulación de referencia a un puntero nulo, la norma no exige que el programa deba bloquearse.* Más bien, el programa podría fallar más temprano o más tarde o no fallar; Incluso podría comportarse como si el puntero nulo apuntara a un objeto válido, y se comportara completamente normalmente, solo para bloquearse en otras circunstancias.

En el caso de la desreferenciación de puntero nulo, el lenguaje C difiere de los lenguajes administrados como Java o C #, donde se *define* el comportamiento de la desreferencia de puntero nulo: se lanza una excepción, en el momento exacto (`NullPointerException` en Java, `NullReferenceException` en C #) Por lo tanto, aquellos que vienen de Java o C # pueden *creer incorrectamente que, en tal caso, un programa en C debe fallar, con o sin la emisión de un mensaje de diagnóstico* .

Información Adicional

Hay varias situaciones de este tipo que deben distinguirse claramente:

- Comportamiento explícitamente indefinido, ahí es donde el estándar C le dice explícitamente que está fuera de los límites.
- Comportamiento implícito indefinido, donde simplemente no hay texto en el estándar que prevea un comportamiento para la situación en la que trajo su programa.

También tenga en cuenta que en muchos lugares el comportamiento de ciertas construcciones está deliberadamente indefinido por el estándar C para dejar espacio para que los implementadores de compiladores y bibliotecas elaboren sus propias definiciones. Un buen ejemplo son las señales y los manejadores de señales, donde las extensiones a C, como el estándar del sistema operativo POSIX, definen reglas mucho más elaboradas. En tales casos, solo tiene que consultar la documentación de su plataforma; El estándar C no te puede decir nada.

También tenga en cuenta que si ocurre un comportamiento indefinido en el programa, esto no significa que solo el punto donde ocurrió un comportamiento indefinido sea problemático, un programa completo deja de tener sentido.

Debido a estas preocupaciones, es importante (sobre todo porque los compiladores no siempre nos advierten acerca de UB) para que la persona programada en C esté al menos familiarizada con el tipo de cosas que desencadenan un comportamiento indefinido.

Cabe señalar que hay algunas herramientas (por ejemplo, herramientas de análisis estático, como PC-Lint) que ayudan a detectar un comportamiento indefinido, pero nuevamente, no pueden detectar todas las apariciones de un comportamiento indefinido.

Examples

Desreferenciación de un puntero nulo

Este es un ejemplo de desreferenciación de un puntero NULL, lo que provoca un comportamiento indefinido.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

El estándar C garantiza un puntero `NULL` para comparar desigualdades con cualquier puntero a un objeto válido, y la anulación de la referencia invoca un comportamiento indefinido.

Modificar cualquier objeto más de una vez entre dos puntos de secuencia.

```
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Código como este a menudo conduce a especulaciones sobre el "valor resultante" de `i` . Sin embargo, en lugar de especificar un resultado, los estándares de C especifican que la evaluación de tal expresión produce *un comportamiento indefinido* . Antes de C2011, el estándar formalizó estas reglas en términos de los llamados *puntos de secuencia* :

> Entre el punto de secuencia anterior y el siguiente, un objeto escalar tendrá su valor almacenado modificado a lo sumo una vez por la evaluación de una expresión. Además, el valor anterior se leerá solo para determinar el valor que se almacenará.

(Norma C99, sección 6.5, párrafo 2)

Ese esquema demostró ser un poco demasiado tosco, lo que dio como resultado que algunas expresiones mostraran un comportamiento indefinido con respecto a C99 que no deberían ser verosímiles. C2011 conserva los puntos de secuencia, pero introduce un enfoque más matizado en esta área basado en la *secuenciación* y una relación que llama "secuenciada antes":

> Si un efecto secundario en un objeto escalar no tiene secuencia en relación con un

> efecto secundario diferente en el mismo objeto escalar o un cálculo de valor utilizando el valor del mismo objeto escalar, el comportamiento no está definido. Si hay varios ordenamientos permitidos de las subexpresiones de una expresión, el comportamiento no está definido si se produce un efecto secundario no secuenciado en cualquiera de los ordenamientos.

(Norma C2011, sección 6.5, párrafo 2)

Los detalles completos de la relación "secuenciada antes" son demasiado largos para describirlos aquí, pero complementan los puntos de la secuencia en lugar de suplantarlos, por lo que tienen el efecto de definir el comportamiento para algunas evaluaciones cuyo comportamiento no estaba definido previamente. En particular, si hay un punto de secuencia entre dos evaluaciones, entonces el que está antes del punto de secuencia está "secuenciado antes" de la siguiente.

El siguiente ejemplo tiene un comportamiento bien definido:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

El siguiente ejemplo tiene un comportamiento indefinido:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-
operators */
```

Al igual que con cualquier forma de comportamiento indefinido, observar el comportamiento real de evaluar expresiones que violan las reglas de secuencia no es informativo, excepto en un sentido retrospectivo. El estándar de lenguaje no proporciona ninguna base para esperar que tales observaciones sean predictivas, incluso del comportamiento futuro del mismo programa.

Falta la declaración de retorno en la función de retorno de valor

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* Trying to use the (not) returned value causes UB */
  int value = foo();
  return 0;
}
```

Cuando se declara que una función devuelve un valor, debe hacerlo en cada ruta de código posible a través de ella. El comportamiento indefinido ocurre tan pronto como la persona que llama (que espera un valor de retorno) intenta usar el valor de retorno [1] .

Tenga en cuenta que el comportamiento indefinido ocurre *solo si* la persona que llama intenta usar / acceder al valor de la función. Por ejemplo,

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* The value (not) returned from foo() is unused. So, this program
   * doesn't cause *undefined behaviour*. */
  foo();
  return 0;
}
```

C99

La `main()` la función es una excepción a esta regla en la que es posible para que pueda ser terminado sin una instrucción de retorno, ya que un valor de retorno supuesta de `0` automáticamente se utilizará en este caso [2].

---

[1] (*ISO / IEC 9899: 201x* , 6.9.1 / 12)

> Si se alcanza el} que termina una función, y el llamante utiliza el valor de la llamada a la función, el comportamiento es indefinido.

[2] (*ISO / IEC 9899: 201x* , 5.1.2.2.3 / 1)

> alcanzando el} que termina la función principal devuelve un valor de 0.

Desbordamiento de entero firmado

Según el párrafo 6.5 / 5 de C99 y C11, la evaluación de una expresión produce un comportamiento indefinido si el resultado no es un valor representable del tipo de expresión. Para los tipos aritméticos, eso se llama un *desbordamiento* . La aritmética de enteros sin signo no se desborda porque se aplica el párrafo 6.2.5 / 9, lo que ocasiona que cualquier resultado sin firmar que de otra forma esté fuera de rango se reduzca a un valor dentro del rango. No hay ninguna disposición análoga para los tipos de enteros con *signo,* sin embargo; Estos pueden y se desbordan, produciendo un comportamiento indefinido. Por ejemplo,

```
#include <limits.h>      /* to get INT_MAX */

int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

La mayoría de los casos de este tipo de comportamiento indefinido son más difíciles de reconocer o predecir. En principio, el desbordamiento puede surgir de cualquier operación de suma, resta o multiplicación en enteros con signo (sujeto a las conversiones aritméticas habituales) donde no existen límites efectivos o una relación entre los operandos para evitarlo. Por ejemplo, esta función:

```
int square(int x) {
```

```
    return x * x;   /* overflows for some values of x */
}
```

es razonable, y hace lo correcto para valores de argumento suficientemente pequeños, pero su comportamiento no está definido para valores de argumento más grandes. No se puede juzgar solo por la función si los programas que lo llaman exhiben un comportamiento indefinido como resultado. Depende de qué argumentos le pasen.

Por otro lado, considere este ejemplo trivial de aritmética de enteros con signo de desbordamiento seguro:

```
int zero(int x) {
    return x - x;   /* Cannot overflow */
}
```

La relación entre los operandos del operador de resta asegura que la resta nunca se desborda. O considere este ejemplo algo más práctico:

```
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++;   /* might overflow */
    while (fgetc(f2) != EOF) count2++;   /* might overflow */

    return count1 - count2; /* provided no UB to this point, will not overflow */
}
```

Mientras los contadores no se desborden individualmente, los operandos de la resta final serán ambos no negativos. Todas las diferencias entre cualquiera de estos dos valores se pueden representar como `int` .

Uso de una variable sin inicializar.

```
int a;
printf("%d", a);
```

La variable `a` es un `int` con duración de almacenamiento automático. El código de ejemplo anterior está tratando de imprimir el valor de una variable no inicializada (`a` no se ha inicializado). Las variables automáticas que no están inicializadas tienen valores indeterminados; El acceso a estos puede llevar a un comportamiento indefinido.

Nota: Las variables con almacenamiento local estático o de subprocesos, incluidas las variables globales sin la palabra clave `static` , se inicializan a cero, o su valor inicializado. Por lo tanto lo siguiente es legal.

```
static int b;
printf("%d", b);
```

Un error muy común es no inicializar las variables que sirven como contadores a 0. Usted les

agrega valores, pero como el valor inicial es basura, invocará un **comportamiento indefinido** , como en la pregunta Compilación en el terminal emite una advertencia de puntero y símbolos extraños

Ejemplo:

```
#include <stdio.h>

int main(void) {
    int i, counter;
    for(i = 0; i < 10; ++i)
        counter += i;
    printf("%d\n", counter);
    return 0;
}
```

Salida:

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o main
main.c:6:9: warning: variable 'counter' is uninitialized when used here [-Wuninitialized]
        counter += i;
        ^~~~~~~
main.c:4:19: note: initialize the variable 'counter' to silence this warning
    int i, counter;
                  ^
                   = 0
1 warning generated.
C02QT2UBFVH6-lm:~ gsamaras$ ./main
32812
```

Las reglas anteriores son aplicables para los punteros también. Por ejemplo, los siguientes resultados en un comportamiento indefinido

```
int main(void)
{
    int *p;
    p++; // Trying to increment an uninitialized pointer.
}
```

Tenga en cuenta que el código anterior por sí solo puede no causar un error o un error de segmentación, pero tratar de eliminar la referencia a este puntero más adelante causaría el comportamiento indefinido.

Desreferenciación de un puntero a una variable más allá de su vida útil

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* (1) The lifetime of baz and bar end here as they have automatic storage
   * duration (local variables), thus the returned pointer is not valid! */

int main (void)
```

```
{
    int* p;

    p = foo(5);  /* (2) this expression's behavior is undefined */
    *p = *p - 6; /* (3) Undefined behaviour here */

    return 0;
}
```

Algunos compiladores lo señalan de manera útil. Por ejemplo, `gcc` advierte con:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

y `clang` advierte con:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

para el código anterior. Pero los compiladores pueden no ser capaces de ayudar en código complejo.

(1) La referencia de retorno a la variable `static` declarada es un comportamiento definido, ya que la variable no se destruye después de dejar el alcance actual.

(2) De acuerdo con ISO / IEC 9899: 2011 6.2.4 §2, "El valor de un puntero se vuelve indeterminado cuando el objeto al que apunta llega al final de su vida útil".

(3) Eliminar la referencia del puntero devuelto por la función `foo` es un comportamiento indefinido ya que la memoria a la que hace referencia tiene un valor indeterminado.

División por cero

```
int x = 0;
int y = 5 / x;  /* integer division */
```

o

```
double x = 0.0;
double y = 5.0 / x;  /* floating point division */
```

o

```
int x = 0;
int y = 5 % x;  /* modulo operation */
```

Para la segunda línea en cada ejemplo, donde el valor del segundo operando (x) es cero, el comportamiento no está definido.

Tenga en cuenta que la mayoría de las implementaciones de matemática de punto flotante seguirán un estándar (p. Ej., IEEE 754), en cuyo caso las operaciones como dividir por cero

tendrán resultados consistentes (p. Ej., `INFINITY`) aunque el estándar C dice que la operación no está definida.

Accediendo a la memoria más allá del trozo asignado

Un puntero a un fragmento de memoria que contiene `n` elementos solo puede ser referenciado si está en el rango de `memory` y `memory + (n - 1)` . La desreferenciación de un puntero fuera de ese rango da como resultado un comportamiento indefinido. Como ejemplo, considere el siguiente código:

```
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

La tercera línea accede al cuarto elemento de una matriz que solo tiene 3 elementos de longitud, lo que lleva a un comportamiento indefinido. Del mismo modo, el comportamiento de la segunda línea en el siguiente fragmento de código tampoco está bien definido:

```
int array[3];
array[3] = 0;
```

Tenga en cuenta que apuntar más allá del último elemento de una matriz no es un comportamiento indefinido (`beyond_array = array + 3` está bien definido aquí), pero la `*beyond_array` referencia es (`*beyond_array` es un comportamiento indefinido). Esta regla también se aplica a la memoria asignada dinámicamente (como los buffers creados a través de `malloc`).

Copiando memoria superpuesta

Una amplia variedad de funciones de biblioteca estándar tienen entre sus efectos la copia de secuencias de bytes de una región de memoria a otra. La mayoría de estas funciones tienen un comportamiento indefinido cuando las regiones de origen y destino se superponen.

Por ejemplo, este ...

```
#include <string.h> /* for memcpy() */

char str[19] = "This is an example";
memcpy(str + 7, str, 10);
```

... intenta copiar 10 bytes donde las áreas de memoria de origen y destino se superponen con tres bytes. Visualizar:

```
                overlapping area
                |
                _ _
               |   |
               v   v
T h i s   i s   a n   e x a m p l e \0
^              ^
|              |
```

```
|             destination
|
source
```

Debido a la superposición, el comportamiento resultante no está definido.

Entre las funciones de la biblioteca estándar con una limitación de este tipo se encuentran `memcpy()` , `strcpy()` , `strcat()` , `sprintf()` y `sscanf()` . El estándar dice de estas y varias otras funciones:

> Si la copia tiene lugar entre objetos que se superponen, el comportamiento no está definido.

La función `memmove()` es la excepción principal a esta regla. Su definición especifica que la función se comporta como si los datos de origen se copiaron primero en un búfer temporal y luego se escribieron en la dirección de destino. No hay excepción para la superposición de regiones de origen y destino, ni ninguna necesidad de una, por lo que `memmove()` tiene un comportamiento bien definido en tales casos.

La distinción refleja una eficiencia *vs.* compensación generalidad. La copia como la que realizan estas funciones generalmente ocurre entre regiones separadas de la memoria y, a menudo, es posible saber en el momento del desarrollo si una instancia particular de la copia de la memoria estará en esa categoría. Suponiendo que la no superposición permite implementaciones comparativamente más eficientes que no producen resultados correctos de manera confiable cuando la suposición no se cumple. A la mayoría de las funciones de la biblioteca de C se les permite implementaciones más eficientes, y `memmove()` llena los vacíos, atendiendo a los casos en que el origen y el destino pueden o se superponen. Para producir el efecto correcto en todos los casos, sin embargo, debe realizar pruebas adicionales y / o emplear una implementación comparativamente menos eficiente.

Lectura de un objeto sin inicializar que no está respaldado por la memoria

C11

La lectura de un objeto causará un comportamiento indefinido, si el objeto es [1] :

- sin inicializar
- definido con duración de almacenamiento automático
- su dirección nunca se toma

La variable a en el siguiente ejemplo satisface todas esas condiciones:

```
void Function( void )
{
    int a;
    int b = a;
}
```

---

[1] (Citado de: ISO: IEC 9899: 201X 6.3.2.1 Lvalores, matrices y designadores de funciones 2)

Si el lvalue designa un objeto de duración de almacenamiento automático que podría haberse declarado con la clase de almacenamiento de registro (nunca se tomó su dirección), y ese objeto no está inicializado (no se declaró con un inicializador y no se realizó ninguna asignación antes de su uso)), el comportamiento es indefinido.

Carrera de datos

C11

C11 introdujo el soporte para múltiples hilos de ejecución, lo que ofrece la posibilidad de carreras de datos. Un programa contiene una carrera de datos si se accede a un objeto [1] por dos subprocesos diferentes, donde al menos uno de los accesos no es atómico, al menos uno modifica el objeto y la semántica del programa no garantiza que los dos accesos no se superpongan temporalmente. [2] Tenga en cuenta que la concurrencia real de los accesos involucrados no es una condición para una carrera de datos; las carreras de datos cubren una clase más amplia de problemas que surgen de inconsistencias (permitidas) en vistas de memoria de diferentes hilos.

Considera este ejemplo:

```
#include <threads.h>

int a = 0;

int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}
```

El hilo principal llama a `thrd_create` para iniciar una nueva función de ejecución de hilo. `Function` . El segundo hilo modifica `a` , y el hilo principal lee `a` . Ninguno de esos accesos es atómico, y los dos hilos no hacen nada individual ni conjuntamente para garantizar que no se superpongan, por lo que hay una carrera de datos.

Entre las formas en que este programa podría evitar la carrera de datos están

- el hilo principal podría realizar su lectura de `a` antes de iniciar el otro hilo;
- el hilo principal podría realizar su lectura de `a` después de asegurar a través de `thrd_join` que la otra ha terminado;
- los hilos podrían sincronizar sus accesos a través de un mutex, cada uno de ellos

> bloqueando ese mutex antes de acceder a `a` y desbloquearlo después.

Como lo demuestra la opción de exclusión mutua, evitar una carrera de datos no requiere garantizar un orden específico de operaciones, como la modificación de `a` subproceso secundario `a` antes de que el subproceso principal lo lea; es suficiente (para evitar una carrera de datos) para asegurar que para una ejecución dada, un acceso suceda antes que el otro.

---

[1] Modificar o leer un objeto.

[2] (Citado de ISO: IEC 9889: 201x, sección 5.1.2.4 "Ejecuciones multiproceso y carreras de datos")

La ejecución de un programa contiene una carrera de datos si contiene dos acciones en conflicto en subprocesos diferentes, al menos una de las cuales no es atómica, y ninguna sucede antes que la otra. Cualquier carrera de datos de este tipo resulta en un comportamiento indefinido.

Leer el valor del puntero que fue liberado.

Incluso solo **leer** el valor de un puntero que se liberó (es decir, sin intentar desreferenciarlo) es un comportamiento indefinido (UB), por ejemplo

```
char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{

}
```

Citando **ISO / IEC 9899: 2011** , sección 6.2.4 §2:

> […] El valor de un puntero se vuelve indeterminado cuando el objeto al que apunta (o simplemente pasa) llega al final de su vida útil.

El uso de memoria indeterminada para cualquier cosa, incluida la comparación aparentemente inofensiva o la aritmética, puede tener un comportamiento indefinido si el valor puede ser una representación de trampa para el tipo.

Modificar cadena literal

En este ejemplo de código, el puntero de carácter `p` se inicializa en la dirección de una cadena literal. El intento de modificar la cadena literal tiene un comportamiento indefinido.

```
char *p = "hello world";
p[0] = 'H'; // Undefined behavior
```

Sin embargo, modificar una matriz mutable de `char` directamente, o mediante un puntero, naturalmente no es un comportamiento indefinido, incluso si su inicializador es una cadena literal. Lo siguiente está bien:

```
char a[] = "hello, world";
char *p = a;

a[0] = 'H';
p[7] = 'W';
```

Esto se debe a que la cadena literal se copia efectivamente a la matriz cada vez que se inicializa (una vez para las variables con duración estática, cada vez que se crea la matriz para las variables con duración automática o de subproceso: las variables con la duración asignada no se inicializan), y está bien para modificar el contenido de la matriz.

Liberar la memoria dos veces

Liberar memoria dos veces es un comportamiento indefinido, por ejemplo,

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Cita del estándar (7.20.3.2. La función libre de C99):

> De lo contrario, si el argumento no coincide con un puntero anterior devuelto por la función calloc, malloc o realloc, o si el espacio ha sido desasignado por una llamada a free o realloc, el comportamiento no está definido.

Usando un especificador de formato incorrecto en printf

El uso de un especificador de formato incorrecto en el primer argumento para `printf` invoca un comportamiento indefinido. Por ejemplo, el siguiente código invoca un comportamiento indefinido:

```
long z = 'B';
printf("%c\n", z);
```

Aquí hay otro ejemplo.

```
printf("%f\n",0);
```

Sobre la línea de código hay un comportamiento indefinido. `%f` espera doble. Sin embargo, 0 es de tipo `int` .

Tenga en cuenta que su compilador generalmente puede ayudarlo a evitar casos como estos, si `-Wformat` indicadores adecuados durante la compilación (`-Wformat` en `clang` y `gcc`). Del último ejemplo:

```
warning: format specifies type 'double' but the argument has type
      'int' [-Wformat]
    printf("%f\n",0);
            ~~    ^
            %d
```

La conversión entre tipos de puntero produce un resultado alineado incorrectamente

Lo siguiente *podría* tener un comportamiento indefinido debido a una alineación incorrecta del puntero:

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);
uint32_t *intptr = (uint32_t*)(memory_block + 1);  /* possible undefined behavior */
uint32_t mvalue = *intptr;
```

El comportamiento indefinido ocurre cuando el puntero se convierte. De acuerdo con C11, si una *conversión entre dos tipos de punteros produce un resultado que se alinea incorrectamente (6.3.2.3), el comportamiento no está definido* . Aquí, un `uint32_t` podría requerir una alineación de 2 o 4, por ejemplo.

`calloc` otro lado, se requiere `calloc` para devolver un puntero que esté alineado adecuadamente para cualquier tipo de objeto; por lo tanto, `memory_block` está alineado correctamente para contener un `uint32_t` en su parte inicial. Luego, en un sistema donde `uint32_t` ha requerido una alineación de 2 o 4, `memory_block + 1` será una dirección *impar* y, por lo tanto, no se alineará correctamente.

Observe que el estándar C solicita que la operación de conversión ya no esté definida. Esto se impone porque en las plataformas donde las direcciones están segmentadas, la dirección de byte `memory_block + 1` puede que ni siquiera tenga una representación adecuada como un puntero de entero.

La `char *` a punteros a otros tipos sin preocuparse por los requisitos de alineación a veces se usa incorrectamente para decodificar estructuras empaquetadas como encabezados de archivos o paquetes de red.

Puede evitar el comportamiento indefinido derivado de una conversión de puntero desalineada utilizando `memcpy` :

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Aquí no se realiza la conversión del puntero a `uint32_t*` y los bytes se copian uno por uno.

Esta operación de copia para nuestro ejemplo solo lleva a un valor válido de `mvalue` porque:

- Utilizamos `calloc` , por lo que los bytes están correctamente inicializados. En nuestro caso, todos los bytes tienen un valor de `0` , pero cualquier otra inicialización adecuada sería suficiente.
- `uint32_t` es un tipo de ancho exacto y no tiene bits de relleno
- Cualquier patrón de bits arbitrario es una representación válida para cualquier tipo sin signo.

Suma o resta del puntero no correctamente delimitada.

El siguiente código tiene un comportamiento indefinido:

```
char buffer[6] = "hello";
char *ptr1 = buffer - 1;  /* undefined behavior */
char *ptr2 = buffer + 5;  /* OK, pointing to the '\0' inside the array */
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char *ptr4 = buffer + 7;  /* undefined behavior */
```

De acuerdo con C11, si la suma o resta de un puntero a, o más allá de un objeto de matriz y un tipo de entero, produce un resultado que no apunta, o simplemente más allá, del mismo objeto de matriz, el comportamiento es indefinido (6.5.6).

Además, es naturalmente un comportamiento indefinido *desreferenciar* un puntero que apunta a algo más allá de la matriz:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char value = *ptr3;       /* undefined behavior */
```

Modificar una variable const mediante un puntero

```
int main (void)
{
    const int foo_readonly = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_readonly; /* (1) This casts away the const qualifier */
    *foo_ptr = 20; /* This is undefined behavior */

    return 0;
}
```

Citando *ISO / IEC 9899: 201x* , sección 6.7.3 §2:

> Si se intenta modificar un objeto definido con un tipo constante calificado mediante el uso de un lvalue con un tipo no calificado, el comportamiento no está definido. [...]

---

(1) En GCC, esto puede lanzar la siguiente advertencia: `warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]`

Pasar un puntero nulo a printf% s conversión

La conversión `%s` de `printf` indica que el argumento correspondiente es *un puntero al elemento inicial de una matriz de tipo de carácter* . Un puntero nulo no apunta al elemento inicial de ninguna matriz de tipo de carácter y, por lo tanto, el comportamiento de los siguientes no está definido:

```
char *foo = NULL;
printf("%s", foo); /* undefined behavior */
```

Sin embargo, el comportamiento indefinido no siempre significa que el programa se bloquee; algunos sistemas toman medidas para evitar el bloqueo que normalmente ocurre cuando se elimina la referencia a un puntero nulo. Por ejemplo, se sabe que Glibc imprime

```
(null)
```

para el código de arriba. Sin embargo, agregue (solo) una nueva línea a la cadena de formato y obtendrá un bloqueo:

```
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

En este caso, sucede porque GCC tiene una optimización que se convierte en `printf("%s\n", argument);` en una llamada a `puts` con `puts(argument)` , y `puts` en Glibc no maneja los punteros nulos. Todo este comportamiento es estándar conforme.

Tenga en cuenta que el *puntero nulo* es diferente de una *cadena vacía* . Por lo tanto, lo siguiente es válido y no tiene un comportamiento indefinido. Solo imprimirá una *nueva línea* :

```
char *foo = "";
printf("%s\n", foo);
```

Enlace inconsistente de identificadores

```
extern int var;
static int var; /* Undefined behaviour */
```

C11, §6.2.2, 7 dice:

> Si, dentro de una unidad de traducción, aparece el mismo identi fi cador con enlaces internos y externos, el comportamiento no está definido.

Tenga en cuenta que si una declaración previa de un identificador es visible, tendrá el enlace de la declaración anterior. *C11, §6.2.2, 4 lo* permite:

> Para un identi fi cador declarado con el especi fi cador de clase de almacenamiento externo en un ámbito en el que se puede ver una declaración previa de ese identi fi cador, 31) si la declaración anterior especifica un enlace interno o externo, el enlace del identi fi cador en la declaración posterior es el mismo que El enlace especificado en la declaración anterior. Si no hay una declaración previa visible, o si la declaración anterior no especifica ningún enlace, entonces el identi fi cador tiene un enlace externo.

```
/* 1. This is NOT undefined */
static int var;
extern int var;


/* 2. This is NOT undefined */
static int var;
static int var;

/* 3. This is NOT undefined */
```

```
extern int var;
extern int var;
```

Usando fflush en un flujo de entrada

Los estándares POSIX y C establecen explícitamente que el uso de `fflush` en una secuencia de entrada es un comportamiento indefinido. El `fflush` está definido solo para flujos de salida.

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <-- undefined behavior
    gets(input);

    return 0;
}
```

No hay una manera estándar de descartar caracteres no leídos de un flujo de entrada. Por otro lado, algunas implementaciones usan `fflush` para borrar el búfer `stdin` . Microsoft define el comportamiento de `fflush` en una secuencia de entrada: si la secuencia está abierta para la entrada, `fflush` borra el contenido del búfer. Según POSIX.1-2008, el comportamiento de `fflush` es indefinido a menos que el archivo de entrada se pueda buscar.

Consulte Uso de `fflush(stdin)` para obtener muchos más detalles.

Desplazamiento de bits utilizando recuentos negativos o más allá del ancho del tipo

Si el valor del *conteo de cambios* es **negativo,** entonces las operaciones de *desplazamiento a la izquierda* y *derecha* no están definidas [1] :

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

Si el *desplazamiento a la izquierda* se realiza en un **valor negativo** , no está definido:

```
int x = -5 << 3; /* undefined */
```

Si el *desplazamiento a la izquierda* se realiza en un **valor positivo** y el resultado del valor matemático **no** se **puede** representar en el tipo, es indefinido [1] :

```
/* Assuming an int is 32-bits wide, the value '5 * 2^72' doesn't fit
 * in an int. So, this is undefined. */

int x = 5 << 72;
```

Tenga en cuenta que *el desplazamiento a la derecha* en un **valor negativo** (.eg `-5 >> 3`) *no* está indefinido sino que está definido por la *implementación* .

---

[1] Cotización *ISO / IEC 9899: 201x* , sección 6.5.7:

> Si el valor del operando derecho es negativo o es mayor o igual que el ancho del operando izquierdo promovido, el comportamiento no está definido.

Modificación de la cadena devuelta por las funciones getenv, strerror y setlocale

La modificación de las cadenas devueltas por las funciones estándar `getenv()` , `strerror()` y `setlocale()` no está definida. Por lo tanto, las implementaciones pueden usar almacenamiento estático para estas cadenas.

La función getenv (), C11, §7.22.4.7, 4 , dice:

> La función getenv devuelve un puntero a una cadena asociada con el miembro de la lista coincidente. La cadena a la que se apunta no debe ser modificada por el programa, pero puede ser sobrescrita por una llamada posterior a la función getenv.

La función strerror (), C11, §7.23.6.3, 4 dice:

> La función strerror devuelve un puntero a la cadena, cuyo contenido es localpeci fi cado. La matriz a la que se apunta no debe ser modificada por el programa, pero puede ser sobrescrita por una llamada posterior a la función strerror.

La función setlocale (), C11, §7.11.1.1, 8 dice:

> El puntero a la cadena devuelto por la función setlocale es tal que una llamada posterior con ese valor de cadena y su categoría asociada restaurará esa parte de la configuración regional del programa. La cadena a la que se apunta no debe ser modificada por el programa, pero puede ser sobrescrita por una llamada posterior a la función setlocale.

De manera similar, la función `localeconv()` devuelve un puntero a la `struct lconv` que no se modificará.

La función localeconv (), C11, §7.11.2.1, 8 dice:

> La función localeconv devuelve un puntero al objeto rellenado. La estructura apuntada por el valor de retorno no debe ser modificada por el programa, pero puede ser sobrescrita por una llamada posterior a la función localeconv.

Volviendo de una función declarada con el especificador de función `_Noreturn` o` noreturn`

C11

El especificador de función `_Noreturn` se introdujo en C11. El encabezado <stdnoreturn.h> proporciona una macro `noreturn` que se expande a `_Noreturn` . Por lo tanto, usar `_Noreturn` o `noreturn` desde <stdnoreturn.h> es <stdnoreturn.h> y equivalente.

Una función que se declara con `_Noreturn` (o `noreturn`) no tiene permitido regresar a su interlocutor. Si tal función *no* vuelve a su interlocutor, el comportamiento no está definido.

En el siguiente ejemplo, `func()` se declara con el especificador `noreturn` pero regresa a su llamador.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void func(void);

void func(void)
{
    printf("In func()...\n");
} /* Undefined behavior as func() returns */

int main(void)
{
    func();
    return 0;
}
```

`gcc` y `clang` producen advertencias para el programa anterior:

```
$ gcc test.c
test.c: In function ‘func’:
test.c:9:1: warning: ‘noreturn’ function does return
 }
 ^
$ clang test.c
test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]
}
^
```

Un ejemplo usando `noreturn` que tiene un comportamiento bien definido:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void my_exit(void);

/* calls exit() and doesn't return to its caller. */
void my_exit(void)
{
    printf("Exiting...\n");
    exit(0);
}

int main(void)
```

```
{
    my_exit();
    return 0;
}
```

Lea Comportamiento indefinido en línea: https://riptutorial.com/es/c/topic/364/comportamiento-indefinido

Capítulo 20: Comunicación entre procesos (IPC)

Introducción

Los mecanismos de comunicación entre procesos (IPC) permiten que diferentes procesos independientes se comuniquen entre sí. El estándar C no proporciona ningún mecanismo de IPC. Por lo tanto, todos estos mecanismos están definidos por el sistema operativo del host. POSIX define un extenso conjunto de mecanismos de IPC; Windows define otro conjunto; Y otros sistemas definen sus propias variantes.

Examples

Semáforos

Los semáforos se utilizan para sincronizar operaciones entre dos o más procesos. POSIX define dos conjuntos diferentes de funciones de semáforo:

1. 'System V IPC' - `semctl()` , `semop()` , `semget()` .
2. 'POSIX Semaphores' - `sem_close()` , `sem_destroy()` , `sem_getvalue()` , `sem_init()` , `sem_open()` , `sem_post()` , `sem_trywait()` , `sem_unlink()` .

Esta sección describe los semáforos de IPC del Sistema V, llamados así porque se originaron con el Sistema V de Unix.

Primero, deberás incluir los encabezados requeridos. Las versiones anteriores de POSIX requerían `#include <sys/types.h>` ; POSIX moderno y la mayoría de los sistemas no lo requieren.

```
#include <sys/sem.h>
```

Luego, deberá definir una clave tanto para el padre como para el niño.

```
#define KEY 0x1111
```

Esta clave debe ser la misma en ambos programas o no se referirán a la misma estructura de IPC. Hay formas de generar una clave acordada sin tener que codificar su valor.

A continuación, dependiendo de su compilador, puede o no necesitar hacer este paso: declarar una unión con el fin de realizar operaciones de semáforo.

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};
```

A continuación, defina sus `semwait` *try* (`semwait`) y *raise* (`semsignal`). Los nombres P y V proceden del holandés.

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

Ahora, comience por obtener el ID para su semáforo de IPC.

```
int id;
// 2nd argument is number of semaphores
// 3rd argument is the mode (IPC_CREAT creates the semaphore set if needed)
if ((id = semget(KEY, 1, 0666 | IPC_CREAT) < 0) {
    /* error handling code */
}
```

En el padre, inicialice el semáforo para tener un contador de 1.

```
union semun u;
u.val = 1;
if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL is a macro to specify that you're setting the
value of the semaphore to that specified by the union u
    /* error handling code */
}
```

Ahora, puedes disminuir o incrementar el semáforo según lo necesites. Al comienzo de su sección crítica, disminuye el contador utilizando la función `semop()` :

```
if (semop(id, &p, 1) < 0) {
    /* error handling code */
}
```

Para incrementar el semáforo, usa `&v` lugar de `&p` :

```
if (semop(id, &v, 1) < 0) {
    /* error handling code */
}
```

Tenga en cuenta que cada función devuelve `0` en caso de éxito y `-1` en caso de fallo. No verificar estos estados de retorno puede causar problemas devastadores.

Ejemplo 1.1: Carreras con hilos

El programa a continuación tendrá un proceso de `fork` un niño y tanto el padre como el niño intentarán imprimir caracteres en el terminal sin ninguna sincronización.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

```
int main()
{
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
}
```

Salida (1ª carrera):

```
aAABaBCbCbDDcEEcddeFFGGHHeffgghh
```

(2da carrera):

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

La compilación y ejecución de este programa debe darle una salida diferente cada vez.

Ejemplo 1.2: Evita competir con semáforos

Modificando el *Ejemplo 1.1* para usar semáforos, tenemos:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};

struct sembuf p = { 0, -1, SEM_UNDO};
struct sembuf v = { 0, +1, SEM_UNDO};

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget"); exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {
        perror("semctl"); exit(12);
    }
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(13);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(14);
            }

            sleep(rand() % 2);
        }
    }
    else
    {
```

```
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(15);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(16);
            }

            sleep(rand() % 2);
        }
    }
}
```

Salida:

```
aabbAABBCCccddeeDDffEEFFGGHHgghh
```

Compilar y ejecutar este programa te dará la misma salida cada vez.

Lea Comunicación entre procesos (IPC) en línea:
https://riptutorial.com/es/c/topic/10564/comunicacion-entre-procesos--ipc-

Capítulo 21: Conversiones implícitas y explícitas

Sintaxis

- Conversión explícita (también conocida como "Casting"): expresión (tipo)

Observaciones

La " *conversión explícita* " también se conoce comúnmente como "casting".

Examples

Conversiones enteras en llamadas a funciones

Dado que la función tiene un prototipo adecuado, los enteros se amplían para las llamadas a funciones de acuerdo con las reglas de conversión de enteros, C11 6.3.1.3.

> **6.3.1.3 Enteros firmados y sin firmar**
> Cuando un valor con tipo entero se convierte en otro tipo entero distinto de _Bool, si el valor puede representarse por el nuevo tipo, no se modifica.
>
> De lo contrario, si el nuevo tipo no está firmado, el valor se convierte sumando o restando repetidamente uno más que el valor máximo que se puede representar en el nuevo tipo hasta que el valor esté en el rango del nuevo tipo.
>
> De lo contrario, el nuevo tipo está firmado y el valor no puede representarse en él; o bien el resultado está definido por la implementación o se genera una señal definida por la implementación.

Por lo general, no debe truncar un tipo de signo ancho para un tipo de signo más estrecho, ya que, obviamente, los valores no se ajustan y no hay un significado claro que debería tener. El estándar C mencionado anteriormente define estos casos como "definidos por la implementación", es decir, no son portátiles.

El siguiente ejemplo supone que `int` es 32 bit de ancho.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u16(uint16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
```

```
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val);  /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}

void param_s64(int64_t val) {
    printf("%s val is " PRI64d "\n", __func__, val); /* Fixed with format string */
}

int main(void) {

    /* Declare integers of various widths */
    uint8_t  u8  = 127;
    uint8_t  s64  = INT64_MAX;

    /* Integer argument is widened when function parameter is wider */
    param_u8(u8);   /* param_u8 val is 127 */
    param_u16(u8);  /* param_u16 val is 127 */
    param_u32(u8);  /* param_u32 val is 127 */
    param_u64(u8);  /* param_u64 val is 127 */
    param_s8(u8);   /* param_s8 val is 127 */
    param_s16(u8);  /* param_s16 val is 127 */
    param_s32(u8);  /* param_s32 val is 127 */
    param_s64(u8);  /* param_s64 val is 127 */

    /* Integer argument is truncated when function parameter is narrower */
    param_u8(s64);  /* param_u8 val is 255 */
    param_u16(s64); /* param_u16 val is 65535 */
    param_u32(s64); /* param_u32 val is 4294967295 */
    param_u64(s64); /* param_u64 val is 9223372036854775807 */
    param_s8(s64);  /* param_s8 val is implementation defined */
    param_s16(s64); /* param_s16 val is implementation defined */
    param_s32(s64); /* param_s32 val is implementation defined */
    param_s64(s64); /* param_s64 val is 9223372036854775807 */

    return 0;
}
```

Conversiones de puntero en llamadas de función

Las conversiones de puntero a `void*` son implícitas, pero cualquier otra conversión de puntero debe ser explícita. Mientras que el compilador permite una conversión explícita de cualquier tipo

de puntero a datos a cualquier otro tipo de puntero a datos, acceder a un objeto a través de un puntero mal escrito es erróneo y conduce a un comportamiento indefinido. El único caso que se permiten es si los tipos son compatibles o si el puntero con el que está mirando el objeto es un tipo de carácter.

```
#include <stdio.h>

void func_voidp(void* voidp) {
    printf("%s Address of ptr is %p\n", __func__, voidp);
}

/* Structures have same shape, but not same type */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {
    printf("%s Address of ptr is %p\n", __func__, (void*) bp);
}

int main(void) {


    /* Implicit ptr conversion allowed for void* */
    func_voidp(&data_a);

    /*
     * Explicit ptr conversion for other types
     *
     * Note that here although the have identical definitions,
     * the types are not compatible, and that the this call is
     * erroneous and leads to undefined behavior on execution.
     */
    func_struct_b((struct struct_b*)&data_a);

    /* My output shows: */
    /* func_charp Address of ptr is 0x601030 */
    /* func_voidp Address of ptr is 0x601030 */
    /* func_struct_b Address of ptr is 0x601030 */

    return 0;
}
```

Lea Conversiones implícitas y explícitas en línea:
https://riptutorial.com/es/c/topic/2529/conversiones-implicitas-y-explicitas

Capítulo 22: Crear e incluir archivos de encabezado

Introducción

En la C moderna, los archivos de encabezado son herramientas cruciales que deben diseñarse y usarse correctamente. Permiten al compilador realizar una verificación cruzada de partes compiladas independientemente de un programa.

Los encabezados declaran los tipos, funciones, macros, etc. que necesitan los consumidores de un conjunto de instalaciones. Todo el código que utiliza cualquiera de esas instalaciones incluye el encabezado. Todo el código que define esas facilidades incluye el encabezado. Esto permite que el compilador compruebe que los usos y las definiciones coinciden.

Examples

Introducción

Hay una serie de pautas a seguir al crear y usar archivos de encabezado en un proyecto de C:

- Idemopotencia

  Si un archivo de encabezado se incluye varias veces en una unidad de traducción (TU), no debe interrumpir las compilaciones.

- Autocontención

  Si necesita las instalaciones declaradas en un archivo de encabezado, no debería tener que incluir ningún otro encabezado explícitamente.

- Minimalidad

  No debería ser capaz de eliminar ninguna información de un encabezado sin causar errores en las compilaciones.

- Incluye lo que usas (IWYU)

  De mayor preocupación para C ++ que para C, pero también importante en C. Si el código en una TU (`code.c`) usa directamente las características declaradas por un encabezado (`"headerA.h"`), entonces `code.c` debería `#include "headerA.h"` directamente, incluso si la TU incluye otro encabezado (`"headerB.h"`) que sucede, en este momento, para incluir `"headerA.h"` .

Ocasionalmente, puede haber razones suficientes para romper una o más de estas pautas, pero ambos deben ser conscientes de que están infringiendo la regla y ser conscientes de las consecuencias de hacerlo antes de romperla.

Idempotencia

Si un archivo de encabezado en particular se incluye más de una vez en una unidad de traducción (TU), no debería haber ningún problema de compilación. Esto se denomina 'idempotencia'; Sus encabezados deben ser idempotentes. Piense lo difícil que sería la vida si tuviera que asegurarse de que `#include <stdio.h>` solo se incluyera una vez.

Hay dos formas de lograr la idempotencia: los protectores de encabezado y la directiva `#pragma once` .

Guardias de cabecera

Los protectores de encabezado son simples y confiables y se ajustan al estándar C. Las primeras líneas sin comentarios en un archivo de encabezado deben tener el siguiente formato:

```
#ifndef UNIQUE_ID_FOR_HEADER
#define UNIQUE_ID_FOR_HEADER
```

La última línea sin comentarios debe ser `#endif` , opcionalmente con un comentario después de ella:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

Todos los códigos operativos, incluidas otras directivas `#include` , deben estar entre estas líneas.

Cada nombre debe ser único. A menudo, se utiliza un esquema de nombre como `HEADER_H_INCLUDED` . Algunos códigos más antiguos utilizan un símbolo definido como la protección del encabezado (por ejemplo, `#ifndef BUFSIZ` en `<stdio.h>`), pero no es tan confiable como un nombre único.

Una opción sería usar un hash MD5 (u otro) generado para el nombre del guardián del encabezado. Debe evitar emular los esquemas utilizados por los encabezados del sistema, que con frecuencia utilizan nombres reservados para la implementación; los nombres comienzan con un guión bajo seguido de otro guión bajo o una letra mayúscula.

La `#pragma once` Directiva

Alternativamente, algunos compiladores admiten la directiva `#pragma once` que tiene el mismo efecto que las tres líneas mostradas para los guardias de encabezado.

```
#pragma once
```

Los compiladores que admiten `#pragma once` incluyen MS Visual Studio, GCC y Clang. Sin embargo, si la portabilidad es una preocupación, es mejor usar protectores de encabezado, o usar ambos. Los compiladores modernos (aquellos que soportan C89 o posterior) deben ignorar, sin comentarios, los pragmas que no reconocen ('Cualquier pragma que no sea reconocido por la

implementación es ignorado') pero las versiones antiguas de GCC no eran tan indulgentes.

Autocontención

Los encabezados modernos deben ser autocontenidos, lo que significa que un programa que necesita usar las facilidades definidas por `header.h` puede incluir ese encabezado (`#include "header.h"`) y no preocuparse de si otros encabezados deben incluirse primero.

Recomendación: Los archivos de encabezado deben ser autocontenidos.

Reglas historicas

Históricamente, este ha sido un tema ligeramente contencioso.

Una vez en otro milenio, los estándares de codificación y estilo de AT&T Indian Hill C establecían:

> Los archivos de encabezado no deben estar anidados. El prólogo de un archivo de encabezado debe, por lo tanto, describir qué otros encabezados deben ser `#include` d para que el encabezado sea funcional. En casos extremos, cuando se debe incluir una gran cantidad de archivos de encabezado en varios archivos fuente diferentes, es aceptable colocar todos los `#include` s comunes en un archivo de inclusión.

Esta es la antítesis de la autocontención.

Reglas modernas

Sin embargo, desde entonces, la opinión ha tendido en la dirección opuesta. Si un archivo de origen necesita usar las instalaciones declaradas por un encabezado `header.h` , el programador debería poder escribir:

```
#include "header.h"
```

y (sujeto solo a que se hayan establecido las rutas de búsqueda correctas en la línea de comando), `header.h` cualquier encabezado necesario necesario sin que sea necesario agregar más encabezados al archivo fuente.

Esto proporciona una mejor modularidad para el código fuente. También protege la fuente del enigma "adivina por qué se agregó este encabezado" que surge después de que el código haya sido modificado y pirateado durante una o dos décadas.

Los estándares de codificación de la NASA para el Centro de Vuelo Espacial Goddard (GSFC) para C es uno de los estándares más modernos, pero ahora es un poco difícil de rastrear. Establece que los encabezados deben ser autocontenidos. También proporciona una forma sencilla de garantizar que los encabezados sean independientes: el archivo de implementación del encabezado debe incluir el encabezado como primer encabezado. Si no es autocontenido,

ese código no se compilará.

La razón dada por GSFC incluye:

> §2.1.1 Encabezado incluye fundamento
>
> Este estándar requiere que el encabezado de una unidad contenga `#include` instrucciones para todos los demás encabezados requeridos por el encabezado de la unidad. La colocación de `#include` para el encabezado de la unidad primero en el cuerpo de la unidad le permite al compilador verificar que el encabezado contiene todas las declaraciones de `#include` requeridas.
>
> Un diseño alternativo, no permitido por esta norma, no permite `#include` declaraciones en los encabezados; Todos los `#includes` se realizan en los archivos del cuerpo. Los archivos de encabezado de unidad luego deben contener declaraciones #ifdef que verifiquen que los encabezados requeridos estén incluidos en el orden correcto.
>
> Una de las ventajas del diseño alternativo es que la lista `#include` en el archivo del cuerpo es exactamente la lista de dependencias necesaria en un archivo make, y el compilador verifica esta lista. Con el diseño estándar, se debe usar una herramienta para generar la lista de dependencias. Sin embargo, todos los entornos de desarrollo recomendados por las sucursales proporcionan dicha herramienta.
>
> Una desventaja importante del diseño alternativo es que si la lista de encabezados requerida de una unidad cambia, cada archivo que usa esa unidad debe editarse para actualizar la lista de la instrucción `#include` . Además, la lista de encabezados requerida para una unidad de biblioteca del compilador puede ser diferente en diferentes objetivos.
>
> Otra desventaja del diseño alternativo es que los archivos de encabezado de la biblioteca del compilador, y otros archivos de terceros, deben modificarse para agregar las declaraciones necesarias de `#ifdef` .

Así, la autocontención significa que:

- Si un encabezado `header.h` necesita un nuevo encabezado anidado `extra.h` , no tiene que verificar todos los archivos de origen que usan `header.h` para ver si necesita agregar `extra.h` .
- Si un encabezado `header.h` ya no necesita incluir un encabezado específico `notneeded.h` , no tiene que verificar todos los archivos de origen que usan `header.h` para ver si puede eliminar `notneeded.h` forma `notneeded.h` (pero vea Incluir lo que usa) .
- No es necesario establecer la secuencia correcta para incluir los encabezados de requisitos previos (lo que requiere una clasificación topológica para hacer el trabajo correctamente).

Comprobando la autocontención.

Consulte Vinculación contra una biblioteca estática para ver un script `chkhdr` que se puede usar para probar la identidad y la autocontención de un archivo de encabezado.

Minimalidad

Los encabezados son un mecanismo de verificación de consistencia crucial, pero deben ser lo más pequeños posible. En particular, eso significa que un encabezado no debe incluir otros encabezados solo porque el archivo de implementación necesitará los otros encabezados. Un encabezado debe contener solo los encabezados necesarios para un consumidor de los servicios descritos.

Por ejemplo, un encabezado de proyecto no debe incluir `<stdio.h>` menos que una de las interfaces de función use el tipo `FILE *` (o uno de los otros tipos definidos únicamente en `<stdio.h>`). Si una interfaz usa `size_t` , el encabezado más pequeño que basta es `<stddef.h>` . Obviamente, si se incluye otro encabezado que define `size_t` , no es necesario incluir `<stddef.h>` también.

Si los encabezados son mínimos, también se reduce al mínimo el tiempo de compilación.

Es posible diseñar encabezados cuyo único propósito es incluir muchos otros encabezados. Esto rara vez resulta ser una buena idea a largo plazo porque pocos archivos de origen realmente necesitarán todas las facilidades descritas por todos los encabezados. Por ejemplo, se podría `<standard-ch>` un `<standard-ch>` que incluya todos los encabezados C estándar, con cuidado, ya que algunos encabezados no siempre están presentes. Sin embargo, muy pocos programas usan las instalaciones de `<locale.h>` o `<tgmath.h>` .

- Consulte también ¿Cómo vincular varios archivos de implementación en C?

Incluye lo que usas (IWYU)

El proyecto Incluir lo que usa de Google, o IWYU, garantiza que los archivos de origen incluyan todos los encabezados utilizados en el código.

Supongamos que un archivo de origen `source.c` incluye un encabezado `arbitrary.h` que a su vez incluye `freeloader.h` , pero el archivo de origen también utiliza explícita e independientemente las instalaciones de `freeloader.h` . Todo está bien para empezar. Entonces, un día se cambia `arbitrary.h` para que sus clientes ya no necesiten las facilidades de `freeloader.h` . De repente, `source.c` deja de compilar, porque no cumple con los criterios de IWYU. Debido a que el código en `source.c` usaba explícitamente las facilidades de `freeloader.h` , debería haber incluido lo que usa; debería haber un explícito `#include "freeloader.h"` en la fuente también. (La idempotencia hubiera asegurado que no hubiera ningún problema.)

La filosofía IWYU maximiza la probabilidad de que el código continúe compilando incluso con cambios razonables realizados en las interfaces. Claramente, si su código llama a una función que se elimina posteriormente de la interfaz publicada, ninguna cantidad de preparación puede evitar que los cambios sean necesarios. Esta es la razón por la que se evitan los cambios en las API cuando es posible, y por qué hay ciclos de desaprobación en varias versiones, etc.

Este es un problema particular en C ++ porque los encabezados estándar pueden incluirse entre sí. El archivo de origen `file.cpp` podría incluir un encabezado `header1.h` que en una plataforma incluye otro encabezado `header2.h` . `file.cpp` puede resultar que use las facilidades de `header2.h` también. Esto no sería un problema inicialmente: el código se compilaría porque `header1.h` incluye

header2.h . En otra plataforma, o una actualización de la plataforma actual, header1.h podría ser revisado para que ya no incluya header2.h , y entonces file.cpp dejaría de compilar como resultado.

IWYU detectaría el problema y recomendaría que header2.h se incluya directamente en file.cpp . Esto aseguraría que continúe compilando. También se aplican consideraciones análogas al código C

Notación y Miscelánea

El estándar C dice que hay muy poca diferencia entre las `#include <header.h>` y `#include "header.h"` .

> [`#include <header.h>`] busca una secuencia de lugares definidos por la implementación para un encabezado identificado únicamente por la secuencia especificada entre los delimitadores `<` y `>` , y provoca el reemplazo de esa directiva por todo el contenido del encabezado. La forma en que se especifican los lugares o el encabezado identificado se define por la implementación.
>
> [`#include "header.h"`] provoca el reemplazo de esa directiva por todo el contenido del archivo fuente identificado por la secuencia especificada entre los delimitadores `"…"` . El archivo fuente nombrado se busca de una manera definida por la implementación. Si esta búsqueda no es compatible, o si la búsqueda falla, la directiva se vuelve a `#include <header.h>` como si leyera [`#include <header.h>`] ...

Por lo tanto, la forma de doble cita puede verse en más lugares que la forma de ángulo entre corchetes. El estándar especifica, por ejemplo, que los encabezados estándar deberían incluirse entre paréntesis angulares, aunque la compilación funciona si utiliza comillas dobles en su lugar. De manera similar, estándares como POSIX usan el formato de ángulo entre corchetes, y usted también debería hacerlo. Reserve los encabezados de comillas dobles para los encabezados definidos por el proyecto. Para los encabezados definidos externamente (incluidos los encabezados de otros proyectos en los que se basa su proyecto), la notación de ángulo-corchete es la más apropiada.

Tenga en cuenta que debe haber un espacio entre `#include` y el encabezado, aunque los compiladores no acepten ningún espacio allí. Los espacios son baratos.

Varios proyectos usan una notación tal como:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

Debería considerar si usar ese control de espacio de nombres en su proyecto (probablemente sea una buena idea). Debe mantenerse alejado de los nombres utilizados por los proyectos existentes (en particular, tanto sys como linux serían malas elecciones).

Si usa esto, su código debe ser cuidadoso y consistente en el uso de la notación.

No utilice la notación `#include "../include/header.h"` .

Los archivos de encabezado rara vez deben definir variables. Aunque mantendrá las variables globales al mínimo, si necesita una variable global, la declarará en un encabezado y la definirá en un archivo fuente adecuado, y ese archivo de origen incluirá el encabezado para verificar la declaración y la definición. y todos los archivos de origen que usan la variable usarán el encabezado para declararla.

Corolario: no declarará variables globales en un archivo de origen; un archivo de origen solo contendrá definiciones.

Los archivos de encabezado rara vez deben declarar funciones `static` , con la notable excepción de las funciones en `static inline` que se definirán en los encabezados si la función es necesaria en más de un archivo fuente.

- Los archivos fuente definen variables globales y funciones globales.
- Los archivos de origen no declaran la existencia de variables o funciones globales; Incluyen el encabezado que declara la variable o función.
- Los archivos de encabezado declaran variables y funciones globales (y tipos y otro material de apoyo).
- Los archivos de encabezado no definen variables o funciones, excepto las funciones en `inline` (`static`).

Referencias cruzadas

- ¿Dónde documentar funciones en C?
- Lista de archivos de encabezado estándar en C y C ++
- ¿Es `inline` sin `static` o `extern` alguna vez útil en C99?
- ¿Cómo uso `extern` para compartir variables entre archivos de origen?
- ¿Cuáles son los beneficios de una ruta relativa como `"../include/header.h"` para un encabezado?
- Optimización de inclusión de encabezado
- ¿Debo incluir cada encabezado?

Lea Crear e incluir archivos de encabezado en línea: https://riptutorial.com/es/c/topic/6257/crear-e-incluir-archivos-de-encabezado

Capítulo 23: Declaración vs Definición

Observaciones

Fuente: ¿Cuál es la diferencia entre una definición y una declaración?

Fuente (para símbolos débiles y fuertes): https://www.amazon.com/Computer-Systems-Programmers-Perspective-2nd/dp/0136108040/

Examples

Entendiendo la Declaración y la Definición

Una declaración introduce un identificador y describe su tipo, ya sea un tipo, objeto o función. Una declaración es lo que el compilador necesita para aceptar referencias a ese identificador. Estas son declaraciones:

```
extern int bar;
extern int g(int, int);
double f(int, double); /* extern can be omitted for function declarations */
double h1();           /* declaration without prototype */
double h2();           /* ditto                         */
```

Una definición en realidad crea una instancia / implementa este identificador. Es lo que necesita el enlazador para vincular las referencias a esas entidades. Estas son definiciones correspondientes a las declaraciones anteriores:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {}  /* prototype is implied in definition, same as double h2(void) */
```

Una definición puede ser usada en lugar de una declaración.

Sin embargo, debe definirse exactamente una vez. Si olvida definir algo que ha sido declarado y referenciado en alguna parte, entonces el enlazador no sabe a qué vincular las referencias y se queja de los símbolos que faltan. Si define algo más de una vez, entonces el enlazador no sabe con cuál de las definiciones vincular las referencias y se queja sobre los símbolos duplicados.

Excepción:

```
extern int i = 0;  /* defines i */
extern int j;  /* declares j */
```

Esta excepción se puede explicar utilizando los conceptos de "Símbolos fuertes frente a símbolos débiles" (desde la perspectiva de un enlazador). Por favor mira aquí (Diapositiva 22) para más

explicación.

```
/* All are definitions. */
struct S { int a; int b; };             /* defines S */
struct X {                              /* defines X */
    int x;                              /* defines non-static data member x */
};
struct X anX;                                  /* defines anX */
```

Lea Declaración vs Definición en línea: https://riptutorial.com/es/c/topic/3104/declaracion-vs-definicion

Capítulo 24: Declaraciones

Observaciones

La declaración de identificador que se refiere a un objeto o función a menudo se denomina abreviatura simplemente como una declaración de objeto o función.

Examples

Llamando a una función desde otro archivo C

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */
                     /* Including a header file twice causes all kinds */
                     /* of interesting problems.*/

/**
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

C Principal

```
#include "foo.h"
```

```
int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Compilar y enlazar

En primer lugar, se *compila* tanto `foo.c` y `main.c` a *los ficheros objeto.* Aquí usamos el compilador `gcc` , su compilador puede tener un nombre diferente y necesitar otras opciones.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Ahora los unimos para producir nuestro ejecutable final:

```
$ gcc -o testprogram foo.o main.o
```

Usando una variable global

El uso de variables globales es generalmente desaconsejado. Hace que su programa sea más difícil de entender y más difícil de depurar. Pero a veces usar una variable global es aceptable.

global.h

```
#ifndef GLOBAL_DOT_H    /* This is an "include guard" */
#define GLOBAL_DOT_H

/**
 * This tells the compiler that g_myglobal exists somewhere.
 * Without "extern", this would create a new variable named
 * g_myglobal in _every file_ that included it. Don't miss this!
 */
extern int g_myglobal; /* _Declare_ g_myglobal, that is promise it will be _defined_ by
                        * some module. */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.
                     */

int g_myglobal;     /* _Define_ my_global. As living in global scope it gets initialised to 0
                     * on program start-up. */
```

C Principal

```
#include "global.h"
```

```
int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

Ver también ¿Cómo uso `extern` para compartir variables entre archivos de origen?

Uso de constantes globales

Los encabezados se pueden usar para declarar recursos de solo lectura de uso global, como tablas de cadenas, por ejemplo.

Declare a aquellos en un encabezado separado que se incluya en cualquier archivo (" *Unidad de traducción* ") que quiera usarlos. Es útil usar el mismo encabezado para declarar una enumeración relacionada para identificar todos los recursos de cadena:

recursos.h:

```
#ifndef RESOURCES_H
#define RESOURCES_H

typedef enum { /* Define a type describing the possible valid resource IDs. */
  RESOURCE_UNDEFINED = -1, /* To be used to initialise any EnumResourceID typed variable to be

                              marked as "not in use", "not in list", "undefined", wtf.
                              Will say un-initialised on application level, not on language
level. Initialised uninitialised, so to say ;-)
                              Its like NULL for pointers ;-)*/
  RESOURCE_UNKNOWN = 0,     /* To be used if the application uses some resource ID,
                              for which we do not have a table entry defined, a fall back in
                              case we _need_ to display something, but do not find anything
                              appropriate. */

  /* The following identify the resources we have defined: */
  RESOURCE_OK,
  RESOURCE_CANCEL,
  RESOURCE_ABORT,
  /* Insert more here. */

  RESOURCE_MAX /* The maximum number of resources defined. */
} EnumResourceID;


extern const char * const resources[RESOURCE_MAX]; /* Declare, promise to anybody who includes

                                      this, that at linkage-time this symbol will be around.
                                      The 1st const guarantees the strings will not change,
                                      the 2nd const guarantees the string-table entries
                                      will never suddenly point somewhere else as set during
                                      initialisation. */
#endif
```

Para definir realmente los recursos, se creó un archivo .c relacionado, que es otra unidad de traducción que contiene las instancias reales de lo que se había declarado en el archivo de encabezado relacionado (.h):

recursos.c:

```
#include "resources.h" /* To make sure clashes between declaration and definition are
                          recognised by the compiler include the declaring header into
                          the implementing, defining translation unit (.c file).

/* Define the resources. Keep the promise made in resources.h. */
const char * const resources[RESOURCE_MAX] = {
  "<unknown>",
  "OK",
  "Cancel",
  "Abort"
};
```

Un programa que usa esto podría verse así:

C Principal:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>

#include "resources.h"


int main(void)
{
  EnumResourceID resource_id = RESOURCE_UNDEFINED;

  while ((++resource_id) < RESOURCE_MAX)
  {
    printf("resource ID: %d, resource: '%s'\n", resource_id, resources[resource_id]);
  }

  return EXIT_SUCCESS;
}
```

Compile los tres archivos anteriores utilizando GCC y vincúlelos para que se conviertan en el archivo `main` del programa, por ejemplo, utilizando esto:

```
gcc -Wall -Wextra -pedantic -Wconversion -g  main.c resources.c -o main
```

(use estos `-Wall -Wextra -pedantic -Wconversion` para hacer que el compilador sea realmente delicado, para que no se pierda nada antes de publicar el código en SO, dirá que el mundo o incluso vale la pena implementarlo en producción)

Ejecuta el programa creado:

```
$ ./main
```

Y obten:

```
resource ID: 0, resource: '<unknown>'
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
```

```
resource ID: 3, resource: 'Abort'
```

Introducción

Ejemplos de declaraciones son:

```
int a; /* declaring single identifier of type int */
```

La declaración anterior declara un identificador único llamado `a` que se refiere a algún objeto con tipo `int` .

```
int a1, b1; /* declaring 2 identifiers of type int */
```

La segunda declaración declara 2 identificadores llamados `a1` y `b1` que se refieren a algunos otros objetos aunque con el mismo tipo `int` .

Básicamente, la forma en que funciona es así: primero se escribe un tipo y luego se escriben una o varias expresiones separadas mediante comas (`,`) (**que no se evaluarán en este punto) y que, de lo contrario, deberían denominarse declaradores en este contexto**). Al escribir tales expresiones, se le permite aplicar solo los operadores de indirección (`*`), llamada de función (`( )`) o subíndice (o indexación de matriz - `[ ]`) en algún identificador (tampoco puede usar ningún operador). No es necesario que el identificador utilizado esté visible en el alcance actual. Algunos ejemplos:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

#	Descripción
1	El nombre del tipo entero.
2	Expresión no evaluada aplicando indirección a algún identificador `z` .
3	Tenemos una coma que indica que una expresión más seguirá en la misma declaración.
4	Expresión no evaluada aplicando indirección a algún otro identificador `x` .
5	Expresión sin evaluar aplicando indirección al valor de la expresión `(*c)` .
6	Fin de la declaración.

Tenga en cuenta que ninguno de los identificadores anteriores era visible antes de esta declaración y, por lo tanto, las expresiones utilizadas no serían válidas antes.

Después de cada expresión, el identificador utilizado en ella se introduce en el alcance actual. (Si el identificador le ha asignado un enlace, también se puede volver a declarar con el mismo tipo de enlace para que ambos identificadores se refieran al mismo objeto o función)

Además, el signo del operador igual (`=`) se puede usar para la inicialización. Si una expresión no

evaluada (declarador) va seguida de = dentro de la declaración, decimos que el identificador que se está introduciendo también se está inicializando. Después del signo = podemos poner una vez más alguna expresión, pero esta vez será evaluada y su valor se usará como inicial para el objeto declarado.

Ejemplos:

```
int l = 90; /* the same as: */

int l; l = 90; /* if it the declaration of l was in block scope */

int c = 2, b[c]; /* ok, equivalent to: */

int c = 2; int b[c];
```

Más adelante en su código, se le permite escribir la misma expresión exacta de la parte de declaración del identificador recién introducido, lo que le da un objeto del tipo especificado al principio de la declaración, asumiendo que ha asignado valores válidos a todos los accesos Objetos en el camino. Ejemplos:

```
void f()
{
    int b2; /* you should be able to write later in your code b2
            which will directly refer to the integer object
            that b2 identifies */

    b2 = 2; /* assign a value to b2 */

    printf("%d", b2); /*ok - should print 2*/

    int *b3; /* you should be able to write later in your code *b3 */

    b3 = &b2; /* assign valid pointer value to b3 */

    printf("%d", *b3); /* ok - should print 2 */

    int **b4; /* you should be able to write later in your code **b4 */

    b4 = &b3;

    printf("%d", **b4); /* ok - should print 2 */

    void (*p)(); /* you should be able to write later in your code (*p)() */

    p = &f; /* assign a valid pointer value */

    (*p)(); /* ok - calls function f by retrieving the
            pointer value inside p -    p
            and dereferencing it -      *p
            resulting in a function
            which is then called -      (*p)() -

            it is not *p() because else first the () operator is
            applied to p and then the resulting void object is
            dereferenced which is not what we want here */
}
```

La declaración de `b3` especifica que potencialmente puede utilizar el valor `b3` como un medio para acceder a algún objeto entero.

Por supuesto, para aplicar indirection (`*`) a `b3` , también debe tener un valor adecuado almacenado en él (vea los punteros para más información). También debe primero almacenar algo de valor en un objeto antes de intentar recuperarlo (puede ver más sobre este problema aquí). Hemos hecho todo esto en los ejemplos anteriores.

```
int a3(); /* you should be able to call a3 */
```

Este le dice al compilador que intentará llamar `a3` . En este caso, `a3` refiere a la función en lugar de un objeto. Una diferencia entre el objeto y la función es que las funciones siempre tendrán algún tipo de enlace. Ejemplos:

```
void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}
```

En el ejemplo anterior, las 2 declaraciones se refieren a la misma función `f2` , mientras que si declararan objetos, en este contexto (con 2 ámbitos de bloque diferentes), tendrían que ser 2 objetos distintos distintos.

```
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Ahora puede parecer que se está complicando, pero si conoce la prioridad de los operadores, tendrá 0 problemas para leer la declaración anterior. Los paréntesis son necesarios porque el operador `*` tiene menos precedencia que el `( )` uno.

En el caso de usar el operador de subíndice, la expresión resultante no sería realmente válida después de la declaración porque el índice utilizado en él (el valor dentro de `[` y `]`) siempre será 1 por encima del valor máximo permitido para este objeto / función.

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

Pero debería ser accesible por todos los demás índices inferiores a 5. Ejemplos:

```
a4[0], a4[1]; a4[4];
```

`a4[5]` resultará en UB. Más información sobre matrices se puede encontrar aquí .

```
int (*a5)[5](); /* here a4 could be applied indirection
                indexed up to (but not including) 5
```

```
                and called */
```

Desafortunadamente para nosotros, aunque sintácticamente posible, la declaración de `a5` está prohibida por la norma actual.

Typedef

Los typedefs son declaraciones que tienen la palabra clave `typedef` delante y antes del tipo. P.ej:

```
typedef int (*(*t0)())[5];
```

(*Técnicamente, también puede poner el typedef después del tipo, como este `int typedef (*(*t0)())[5];` pero esto no es recomendable*)

Las declaraciones anteriores declaran un identificador para un nombre typedef. Puedes usarlo así después:

```
t0 pf;
```

Que tendrá el mismo efecto que la escritura:

```
int (*(*pf)())[5];
```

Como puede ver, el nombre typedef "guarda" la declaración como un tipo que se usará más adelante para otras declaraciones. De esta manera puedes guardar algunas pulsaciones de teclado. Además, como la declaración que usa `typedef` sigue siendo una declaración, no está limitado solo por el ejemplo anterior:

```
t0 (*pf1);
```

Es lo mismo que:

```
int (*(**pf1)())[5];
```

Uso de la regla derecha-izquierda o espiral para descifrar la declaración de C

La regla "derecha-izquierda" es una regla completamente regular para descifrar las declaraciones C También puede ser útil para crearlos.

Lee los símbolos a medida que los encuentres en la declaración ...

```
*   as "pointer to"          - always on the left side
[]  as "array of"            - always on the right side
()  as "function returning"  - always on the right side
```

Cómo aplicar la regla.

PASO 1

Encuentra el identificador. Este es su punto de partida. Entonces di a ti mismo, "identificador es". Has comenzado tu declaración.

PASO 2

Mira los símbolos a la derecha del identificador. Si, por ejemplo, encuentra `()` allí, entonces sabe que esta es la declaración de una función. Entonces tendrías que *"el identificador es la función que regresa"* . O si encontraras un `[]` allí, dirías *"el identificador es una matriz de"* . Continúa a la derecha hasta que te quedes sin símbolos O golpeas un paréntesis derecho `)` . (Si golpeas un paréntesis izquierdo `(` , ese es el comienzo de un símbolo `()` , incluso si hay cosas entre los paréntesis. Más sobre esto más abajo).

PASO 3

Mira los símbolos a la izquierda del identificador. Si no es uno de nuestros símbolos anteriores (por ejemplo, algo como "int"), solo dígalo. De lo contrario, tradúzcalo al inglés usando la tabla de arriba. Sigue yendo a la izquierda hasta que te quedes sin símbolos O golpeas un paréntesis izquierdo `(` .

Ahora repita los pasos 2 y 3 hasta que haya formado su declaración.

Aquí hay unos ejemplos:

```
int *p[];
```

Primero, encuentra el identificador:

```
int *p[];
     ^
```

"p es"

Ahora, muévete a la derecha hasta que salga de los símbolos o del paréntesis derecho.

```
int *p[];
      ^^
```

"p es una matriz de"

Ya no puedes moverte a la derecha (sin símbolos), así que muévete a la izquierda y encuentra:

```
int *p[];
    ^
```

"p es matriz de puntero a"

Sigue yendo a la izquierda y encuentra:

```
int *p[];
^^^
```

"p es la matriz de puntero a int".

(o *"p es una matriz donde cada elemento es de tipo puntero a int"*)

Otro ejemplo:

```
int *(*func())();
```

Encuentra el identificador.

```
int *(*func())();
       ^^^^
```

"func es"

Mover a la derecha.

```
int *(*func())();
           ^^
```

"func es la función de retorno"

Ya no se puede mover a la derecha debido al paréntesis derecho, así que muévete a la izquierda.

```
int *(*func())();
      ^
```

"func es la función que devuelve el puntero a"

Ya no puedo moverme a la izquierda debido al paréntesis izquierdo, así que sigue derecho.

```
int *(*func())();
              ^^
```

"func es la función que devuelve el puntero a la función que regresa"

Ya no puedes moverte a la derecha porque nos hemos quedado sin símbolos, así que ve a la izquierda.

```
int *(*func())();
    ^
```

"func es la función que devuelve el puntero a la función que devuelve el puntero a"

Y finalmente, sigue hacia la izquierda, porque no queda nada a la derecha.

```
int *(*func())();
^^^
```

"func es la función que devuelve el puntero a la función que devuelve el puntero a int".

Como puedes ver, esta regla puede ser bastante útil. También puede usarlo para comprobar la cordura mientras crea las declaraciones y para darle una pista sobre dónde colocar el siguiente símbolo y si se requieren paréntesis.

Algunas declaraciones parecen mucho más complicadas que debido a los tamaños de matriz y las listas de argumentos en forma de prototipo. Si ve `[3]` , se lee como *"matriz (tamaño 3) de ..."* . Si ve `(char *,int)` , se lee como * "función esperando (char *, int) y devolviendo ..."* .

Aquí hay una divertida:

```
int (*(*fun_one)(char *,double))[9][20];
```

No voy a seguir cada uno de los pasos para descifrar este.

* "fun_one es un puntero a la función esperando (char *, doble) y devuelve el puntero a la matriz (tamaño 9) de la matriz (tamaño 20) de int."*

Como puede ver, no es tan complicado si se deshace de los tamaños de matriz y las listas de argumentos:

```
int (*(*fun_one)())[][];
```

Puede descifrarlo de esa manera, y luego colocarlo en la matriz de tamaños y listas de argumentos más adelante.

Algunas palabras finales:

Es muy posible hacer declaraciones ilegales usando esta regla, por lo que es necesario conocer algo de lo que es legal en C. Por ejemplo, si lo anterior hubiera sido:

```
int *((*fun_one)())[][];
```

habría leído *"fun_one es un puntero a la función que devuelve una matriz de matriz de puntero a int"* . Dado que una función no puede devolver una matriz, sino solo un puntero a una matriz, esa declaración es ilegal.

Las combinaciones ilegales incluyen:

```
[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array
```

En todos los casos anteriores, necesitaría un conjunto de paréntesis para enlazar un símbolo `*` a

la izquierda entre estos `()` y `[]` símbolos del lado derecho para que la declaración sea legal.

Aquí hay algunos ejemplos más:

Legal

```
int i;                an int
int *p;               an int pointer (ptr to an int)
int a[];              an array of ints
int f();              a function returning an int
int **pp;             a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];          a pointer to an array of ints
int (*pf)();          a pointer to a function returning an int
int *ap[];            an array of int pointers (array of ptrs to ints)
int aa[][];           an array of arrays of ints
int *fp();            a function returning an int pointer
int ***ppp;           a pointer to a pointer to an int pointer
int (**ppa)[];        a pointer to a pointer to an array of ints
int (**ppf)();        a pointer to a pointer to a function returning an int
int *(*pap)[];        a pointer to an array of int pointers
int (*paa)[][];       a pointer to an array of arrays of ints
int *(*pfp)();        a pointer to a function returning an int pointer
int **app[];          an array of pointers to int pointers
int (*apa[])[];       an array of pointers to arrays of ints
int (*apf[])();       an array of pointers to functions returning an int
int *aap[][];         an array of arrays of int pointers
int aaa[][][];        an array of arrays of arrays of int
int **fpp();          a function returning a pointer to an int pointer
int (*fpa())[];       a function returning a pointer to an array of ints
int (*fpf())();       a function returning a pointer to a function returning an int
```

Ilegal

```
int af[]();           an array of functions returning an int
int fa()[];           a function returning an array of ints
int ff()();           a function returning a function returning an int
int (*pfa)()[];       a pointer to a function returning an array of ints
int aaf[][]();        an array of arrays of functions returning an int
int (*paf)[]();       a pointer to a an array of functions returning an int
int (*pff)()();       a pointer to a function returning a function returning an int
int *afp[]();         an array of functions returning int pointers
int afa[]()[];        an array of functions returning an array of ints
int aff[]()();        an array of functions returning functions returning an int
int *fap()[];         a function returning an array of int pointers
int faa()[][];        a function returning an array of arrays of ints
int faf()[]();        a function returning an array of functions returning an int
int *ffp()();         a function returning a function returning an int pointer
```

Fuente: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Lea Declaraciones en línea: https://riptutorial.com/es/c/topic/3729/declaraciones

Capítulo 25: Declaraciones de selección

Examples

if () Declaraciones

Una de las formas más simples de controlar el flujo del programa es mediante el uso de sentencias de selección `if` . Esta sentencia puede decidir si un bloque de código se ejecutará o no se ejecutará.

La sintaxis de `if` declaración de selección en C podría ser la siguiente:

```
if(cond)
{
  statement(s);  /*to be executed, on condition being true*/
}
```

Por ejemplo,

```
if (a > 1) {
    puts("a is larger than 1");
}
```

Donde `a > 1` es una *condición* que debe evaluarse como `true` para ejecutar las sentencias dentro del bloque `if` . En este ejemplo, "a es mayor que 1" solo se imprime si `a > 1` es verdadero.

`if` declaraciones de selección pueden omitir las llaves de ajuste `{` y `}` si solo hay una declaración dentro del bloque. El ejemplo anterior se puede reescribir a

```
if (a > 1)
    puts("a is larger than 1");
```

Sin embargo, para ejecutar varias instrucciones dentro del bloque, las llaves deben usarse.

La *condición* para `if` puede incluir múltiples expresiones. `if` solo realizará la acción si el resultado final de la expresión es verdadero.

Por ejemplo

```
if ((a > 1) && (b > 1)) {
    puts("a is larger than 1");
    a++;
}
```

solamente se ejecutará la `printf` y `a++` si **tanto** `a` y `b` son mayores que `1` .

if () ... else sentencias y sintaxis

Mientras que `if` realiza una acción solo cuando su condición se evalúa como `true` , `if` / `else` permite especificar las diferentes acciones cuando la condición sea `true` y cuando la condición sea `false` .

Ejemplo:

```
if (a > 1)
    puts("a is larger than 1");
else
    puts("a is not larger than 1");
```

Al igual que el `if` declaración, cuando el bloque dentro de `if` o `else` está formado por una sola declaración, a continuación, los tirantes se puede omitir (pero el hacerlo no se recomienda ya que puede introducir fácilmente los problemas involuntariamente). Sin embargo, si hay más de una declaración dentro del bloque `if` o `else` , entonces las llaves deben usarse en ese bloque en particular.

```
if (a > 1)
{
    puts("a is larger than 1");
    a--;
}
else
{
    puts("a is not larger than 1");
    a++;
}
```

Switch () Declaraciones

`switch` instrucciones de `switch` son útiles cuando quiere que su programa haga muchas cosas diferentes de acuerdo con el valor de una variable de prueba en particular.

Un ejemplo de uso de la instrucción `switch` es como esto:

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a is neither 1 nor 2");
    break;
}
```

Este ejemplo es equivalente a

```
int a = 1;
```

```
if (a == 1) {
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a is neither 1 nor 2");
}
```

Si el valor de `a` es 1 cuando se usa la instrucción de `switch` , `a is 1` se imprimirá. Si el valor de `a` es 2, entonces, `a is 2` será impreso. De lo contrario, `a is neither 1 nor 2` se imprimirá `a is neither 1 nor 2` .

`case n:` se utiliza para describir dónde saltará el flujo de ejecución cuando el valor pasado a la instrucción de `switch` es *n* . *n* debe ser una constante de tiempo de compilación y la misma *n* puede existir como máximo una vez en una instrucción `switch` .

`default:` se utiliza para describir que cuando el valor no coincide con ninguna de las opciones para el `case n:` Es una buena práctica incluir un caso `default` en cada instrucción de cambio para detectar un comportamiento inesperado.

Un `break;` Se requiere una declaración para saltar fuera del bloque de `switch` .

Nota: Si accidentalmente olvida agregar una `break` después del final de un `case` , el compilador asumirá que tiene la intención de "fracasar" y todas las declaraciones de casos subsiguientes, si las hubiera, se ejecutarán (a menos que se encuentre una declaración de *ruptura* en cualquiera de los casos subsiguientes), independientemente de si las declaraciones de caso subsiguientes coinciden o no. Esta propiedad en particular se utiliza para implementar el dispositivo de Duff . Este comportamiento a menudo se considera un defecto en la especificación del lenguaje C.

A continuación se muestra un ejemplo que muestra los efectos de la ausencia de `break;` :

```
int a = 1;

switch (a) {
case 1:
case 2:
    puts("a is 1 or 2");
case 3:
    puts("a is 1, 2 or 3");
    break;
default:
    puts("a is neither 1, 2 nor 3");
    break;
}
```

Cuando el valor de `a` es 1 o 2, `a is 1 or 2` y `a is 1, 2 or 3` se imprimirán. Cuando `a` es 3, solo se imprimirá a `a is 1, 2 or 3` . De lo contrario, `a is neither 1, 2 nor 3` se imprimirán.

Tenga en cuenta que el caso `default` no es necesario, especialmente cuando el conjunto de valores que obtiene en el `switch` finaliza y se conoce en el momento de la compilación.

El mejor ejemplo es usar un `switch` en una `enum` .

```
enum msg_type { ACK, PING, ERROR };
void f(enum msg_type t)
{
  switch (t) {
  case ACK:
    // do nothing
    break;
  case PING:
    // do something
    break;
  case ERROR:
    // do something else
    break;
  }
}
```

Hay múltiples ventajas de hacer esto:

- la mayoría de los compiladores informarán una advertencia si no maneja un valor (esto no se informaría si hubiera un caso `default`)
- por el mismo motivo, si agrega un nuevo valor a la `enum` , se le notificará de todos los lugares donde se olvidó de manejar el nuevo valor (con un caso `default` , deberá explorar manualmente su código en busca de dichos casos)
- El lector no necesita averiguar "qué está oculto por el `default:` ", si existen otros valores de `enum` o si es una protección para "por si acaso". Y si hay otros valores de `enum` , ¿el codificador usó intencionalmente el caso `default` para ellos o hay un error que se introdujo cuando agregó el valor?
- el manejo de cada valor de `enum` hace que el código sea autoexplicativo ya que no puede esconderse detrás de un comodín, debe manejar explícitamente cada uno de ellos.

Sin embargo, no puedes evitar que alguien escriba código malvado como:

```
enum msg_type t = (enum msg_type)666; // I'm evil
```

Por lo tanto, puede agregar un control adicional antes de que su interruptor lo detecte, si realmente lo necesita.

```
void f(enum msg_type t)
{
   if (!is_msg_type_valid(t)) {
      // Handle this unlikely error
   }

   switch(t) {
    // Same code than before
   }
}
```

if () ... else Ladder Chaining dos o más if () ... else sentencias

Mientras que la instrucción `if ()... else` permite definir solo un comportamiento (predeterminado) que ocurre cuando no se cumple la condición dentro de `if ()` , encadenando dos o más `if () ... else`

declaraciones `if () ... else` permiten definir un par más comportamientos antes de ir a la última `else` rama que actúa como un "defecto", en su caso.

Ejemplo:

```
int a = ... /* initialise to some value. */

if (a >= 1)
{
    printf("a is greater than or equals 1.\n");
}
else if (a == 0) //we already know that a is smaller than 1
{
    printf("a equals 0.\n");
}
else /* a is smaller than 1 and not equals 0, hence: */
{
    printf("a is negative.\n");
}
```

Anidado si () ... else VS if () .. else Ladder

Las instrucciones anidadas `if()...else` tardan más tiempo de ejecución (son más lentas) en comparación con una escalera `if()...else` porque las instrucciones anidadas `if()...else` comprueban todas las declaraciones condicionales internas una vez que son externas la sentencia condicional `if()` se cumple, mientras que la escalera `if()..else` detendrá la prueba de condición una vez que cualquiera de las sentencias condicionales `if()` o `else if()` sean verdaderas.

Una escalera `if()...else` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if ((a < b) && (a < c))
  {
    printf("\na = %d is the smallest.", a);
  }
  else if ((b < a) && (b < c))
  {
    printf("\nb = %d is the smallest.", b);
  }
  else if ((c < a) && (c < b))
  {
    printf("\nc = %d is the smallest.", c);
  }
  else
  {
    printf("\nImprove your coding logic");
  }
  return 0;
}
```

En el caso general, se considera que es mejor que el equivalente anidado `if()...else` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if (a < b)
  {
    if (a < c)
      {
        printf("\na = %d is the smallest.", a);
      }
    else
      {
        printf("\nc = %d is the smallest.", c);
      }
  }
  else
  {
    if(b < c)
    {
      printf("\nb = %d is the smallest.", b);
    }
    else
    {
      printf("\nc = %d is the smallest.", c);
    }
  }
  return 0;
}
```

Lea Declaraciones de selección en línea: https://riptutorial.com/es/c/topic/3073/declaraciones-de-seleccion

Capítulo 26: Efectos secundarios

Examples

Operadores Pre / Post Incremento / Decremento

En C, hay dos operadores únicos: '++' y '-' que son una fuente muy común de confusión. El operador `++` se llama el *operador de incremento* y el operador `--` se llama el *operador de disminución* . Ambos pueden usarse en forma de *prefijo* o de *postfijo* . La sintaxis para el formulario de prefijo para el operador `++` es `++operand` y la sintaxis para el formulario postfix es `operand++` . Cuando se usa en la forma de prefijo, el operando se incrementa primero en `1` y el valor resultante del operando se usa en la evaluación de la expresión. Considere el siguiente ejemplo:

```
int n, x = 5;
n = ++x; /* x is incremented  by 1(x=6), and result is assigned to n(6) */
         /* this is a short form for two statements: */
         /* x = x + 1; */
         /* n = x ; */
```

Cuando se usa en la forma de posfijo, el valor actual del operando se usa en la expresión y luego el valor del operando se incrementa en `1` . Considere el siguiente ejemplo:

```
int n, x = 5;
n = x++; /* value of x(5) is assigned first to n(5), and then x is incremented by 1; x(6) */
         /* this is a short form for two statements: */
         /* n = x; */
         /* x = x + 1; */
```

El funcionamiento del operador decremento `--` puede entenderse de manera similar.

El siguiente código demuestra lo que hace cada uno.

```
    int main()
    {
        int a, b, x = 42;
        a = ++x; /* a and x are 43 */
        b = x++; /* b is 43, x is 44 */
        a = x--; /* a is is 44, x is 43 */
        b = --x; /* b and x are 42 */

        return 0;
    }
```

De lo anterior queda claro que los operadores de correos devuelven el valor actual de una variable y *luego lo* modifican, pero los operadores previos modifican la variable y *luego* devuelven el valor modificado.

En todas las versiones de C, el orden de evaluación de los operadores pre y post no está

definido, por lo que el siguiente código puede devolver resultados inesperados:

```
int main()
{
    int a, x = 42;
    a = x++ + x; /* wrong */
    a = x + x; /* right */
    ++x;

    int ar[10];
    x = 0;
    ar[x] = x++; /* wrong */
    ar[x++] = x; /* wrong */
    ar[x] = x; /* right */
    ++x;
    return 0;
}
```

Tenga en cuenta que también es una buena práctica utilizar operadores de publicación previa a publicación previa cuando se utiliza solo en una declaración. Mira el código anterior para esto.

Tenga en cuenta también que cuando se llama a una función, todos los efectos secundarios en los argumentos deben tener lugar antes de que se ejecute la función.

```
int foo(int x)
{
    return x;
}

int main()
{
    int a = 42;
    int b = foo(a++);  /* This returns 43, even if it seems like it should return 42 */
    return 0;
}
```

Lea Efectos secundarios en línea: https://riptutorial.com/es/c/topic/7094/efectos-secundarios

Capítulo 27: En línea

Examples

Funciones de alineación utilizadas en más de un archivo fuente

Para funciones pequeñas que se llaman a menudo, la sobrecarga asociada con la llamada de función puede ser una fracción significativa del tiempo total de ejecución de esa función. Una forma de mejorar el rendimiento, entonces, es eliminar los gastos generales.

En este ejemplo, usamos cuatro funciones (más `main()`) en tres archivos de origen. Dos de esos (`plusfive()` y `timestwo()`) son llamados por los otros dos ubicados en "source1.c" y "source2.c". El `main()` está incluido, así que tenemos un ejemplo de trabajo.

C Principal:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);
    printf("First result is %d\n", intermediate);
    intermediate = complicated2(start);
    printf("Second result is %d\n", intermediate);
    return 0;
}
```

fuente1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated2(int input) {
    int tmp = plusfive(input);
    tmp = timestwo(tmp);
```

```
    return tmp;
}
```

headerfile.h:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
  return input * 2;
}
inline int plusfive(int input) {
  return input + 5;
}

#endif
```

A las funciones `timestwo` y `plusfive` se las llama `complicated1` y `complicated2` , que se encuentran en diferentes "unidades de traducción" o archivos fuente. Para usarlos de esta manera, tenemos que definirlos en el encabezado.

Compilar de esta manera, asumiendo gcc:

```
cc -O2 -std=c99 -c -o main.o main.c
cc -O2 -std=c99 -c -o source1.o source1.c
cc -O2 -std=c99 -c -o source2.o source2.c
cc main.o source1.o source2.o -o main
```

Usamos la opción de optimización -O2 porque algunos compiladores no están en línea sin la optimización activada.

El efecto de la palabra clave en `inline` es que el símbolo de función en cuestión no se emite en el archivo objeto. De lo contrario, se produciría un error en la última línea, donde estamos vinculando los archivos de objeto para formar el ejecutable final. Si no estuviéramos en `inline` , el mismo símbolo se definiría en ambos archivos `.o` , y se produciría un error de "símbolo definido múltiple".

En situaciones donde el símbolo es realmente necesario, esto tiene la desventaja de que el símbolo no se produce en absoluto. Hay dos posibilidades para lidiar con eso. El primero es agregar una declaración `extern` adicional de las funciones en línea en exactamente uno de los archivos `.c` . Entonces agregue lo siguiente a `source1.c` :

```
extern int timestwo(int input);
extern int plusfive(int input);
```

La otra posibilidad es definir la función con `static inline` lugar de en `inline` . Este método tiene el inconveniente de que eventualmente se puede producir una copia de la función en cuestión en **cada** archivo de objeto que se produce con este encabezado.

Lea En línea en línea: https://riptutorial.com/es/c/topic/7427/en-linea

Capítulo 28: Entrada / salida formateada

Examples

Impresión del valor de un puntero a un objeto

Para imprimir el valor de un puntero a un objeto (en lugar de un puntero de función) use el especificador de conversión `p` . Se define para imprimir solo los puntos de `void` , por lo que para imprimir el valor de un punto de no `void` se debe convertir explícitamente ("fundido *") a `void*` .

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>  /* for printf() */

int main(void)
{
  int i;
  int * p = &i;

  printf("The address of i is %p.\n", (void*) p);

  return EXIT_SUCCESS;
}
```

C99

Usando `<inttypes.h>` y `uintptr_t`

Otra forma de imprimir punteros en C99 o posterior utiliza el tipo `uintptr_t` y las macros de `<inttypes.h>` :

```
#include <inttypes.h> /* for uintptr_t and PRIXPTR */
#include <stdio.h>    /* for printf() */

int main(void)
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%" PRIXPTR ".\n", (uintptr_t)p);

  return 0;
}
```

En teoría, es posible que no haya un tipo de entero que pueda contener cualquier puntero convertido en un entero (por lo que el tipo `uintptr_t` puede no existir). En la práctica, sí existe. Los punteros a las funciones no necesitan ser convertibles al tipo `uintptr_t` , aunque nuevamente son convertibles.

Si el tipo `uintptr_t` existe, también lo hace el tipo `intptr_t` . Sin embargo, no está claro por qué querría tratar las direcciones como enteros con signo.

K&R C89

Historia Pre-Estándar:

Antes de C89 durante los tiempos K y R-C, no había ningún tipo `void*` (ni encabezado `<stdlib.h>` , ni prototipos, y por lo tanto no había una notación `int main(void)`), por lo que el puntero se proyectó a `long unsigned int` y se imprimió utilizando `lx` modificador de longitud / especificador de conversión.

El siguiente ejemplo es sólo para fines informativos. Hoy en día este código no es válido, lo que muy bien podría provocar el infame comportamiento indefinido .

```
#include <stdio.h> /* optional in pre-standard C - for printf() */

int main()
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%lx.\n", (long unsigned) p);

  return 0;
}
```

Impresión de la diferencia de los valores de dos punteros a un objeto

Restar los valores de dos punteros a un objeto da como resultado un entero con signo [* 1] . Por lo tanto, se imprimiría utilizando *al menos* el especificador de conversión `d` .

Para asegurarse de que hay un tipo lo suficientemente ancho como para contener tal "diferencia de puntero", ya que C99 `<stddef.h>` define el tipo `ptrdiff_t` . Para imprimir un `ptrdiff_t` usa el modificador `t` longitud.

C99

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
#include <stddef.h> /* for ptrdiff_t */


int main(void)
{
  int a[2];
  int * p1 = &a[0], * p2 = &a[1];
  ptrdiff_t pd = p2 - p1;

  printf("p1 = %p\n", (void*) p1);
  printf("p2 = %p\n", (void*) p2);
  printf("p2 - p1 = %td\n", pd);

  return EXIT_SUCCESS;
}
```

El resultado podría verse así:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

Tenga en cuenta que el valor resultante de la diferencia se escala según el tamaño del tipo al que los punteros le restaron el punto, un `int` aquí. El tamaño de un `int` para este ejemplo es 4.

---

* 1 Si los dos punteros que se deben restar no apuntan al mismo objeto, el comportamiento no está definido.

Especificadores de conversión para la impresión

Especificador de conversión	Tipo de argumento	Descripción
`i` , `d`	En t	imprime decimal
`u`	sin firmar int	imprime decimal
`o`	sin firmar int	impresiones octales
`x`	sin firmar int	Impresiones hexadecimales, minúsculas.
`X`	sin firmar int	impresiones hexadecimales, mayúsculas
`f`	doble	las impresiones flotan con una precisión predeterminada de 6, si no se proporciona ninguna precisión (minúscula utilizada para números especiales `nan` e `inf` o `infinity`)
`F`	doble	las impresiones flotan con una precisión predeterminada de 6, si no se proporciona ninguna precisión (mayúscula utilizada para los números especiales `NAN` e `INF` o `INFINITY`)
`e`	doble	las impresiones flotan con una precisión predeterminada de 6, si no se da una precisión, usando notación científica usando mantisa / exponente; Exponente en minúsculas y números especiales.
`E`	doble	las impresiones flotan con una precisión predeterminada de 6, si no se da una precisión, usando notación científica usando mantisa / exponente; Exponente en mayúsculas y números especiales.
`g`	doble	utiliza `f` o `e` *[ver más abajo]*

Especificador de conversión	Tipo de argumento	Descripción
`G`	doble	utiliza `F` o `E` *[ver más abajo]*
`a`	doble	Impresiones hexadecimales, minúsculas.
`A`	doble	impresiones hexadecimales, mayúsculas
`c`	carbonizarse	imprime un solo carácter
`s`	carbonizarse*	imprime una cadena de caracteres hasta un terminador `NUL` , o truncada a la longitud dada por la precisión, si se especifica
`p`	vacío*	imprime el `void` puntero `void` ; un punto no `void` debe convertirse explícitamente ("cast") a `void*` ; puntero a objeto solamente, **no** un puntero a función
`%`	n / A	imprime el carácter `%`
`n`	En t *	***escriba*** el número de bytes impresos hasta el momento en el `int` apuntado.

Tenga en cuenta que los modificadores de longitud se pueden aplicar a `%n` (por ejemplo, `%hhn` indica que el *siguiente especificador de conversión* `n` *aplica a un puntero a un argumento* `signed char` , según ISO / IEC 9899: 2011 §7.21.6.1 (7)).

Tenga en cuenta que las conversiones de punto flotante se aplican a los tipos `float` y `double` debido a las reglas de promoción predeterminadas - §6.5.2.2 Llamadas de función, ¶7 *La notación de puntos suspensivos en un declarador de prototipo de función hace que la conversión de tipo de argumento se detenga después del último parámetro declarado. Las promociones de argumentos predeterminados se realizan en los argumentos finales.*) Por lo tanto, funciones como `printf()` solo pasan valores `double` , incluso si la variable referenciada es de tipo `float` .

Con los formatos `g` y `G` , la elección entre la notación `e` y `f` (o `E` y `F`) está documentada en el estándar C y en la especificación POSIX para `printf()` :

> El doble argumento que representa un número de punto flotante se convertirá en el estilo `f` o `e` (o en el estilo `F` o `E` en el caso de un especificador de conversión `G`), dependiendo del valor convertido y la precisión. Sea `P` igual a la precisión si no es cero, 6 si se omite la precisión o 1 si la precisión es cero. Entonces, si una conversión con estilo `E` tendría un exponente de `X` :
>
> - Si P> X> = -4, la conversión será con estilo `f` (o `F`) y precisión `P - (X+1)` .
> - De lo contrario, la conversión será con estilo `e` (o `E`) y precisión `P - 1` .
>
> Finalmente, a menos que se use la bandera '#', los ceros finales se eliminarán de la parte fraccionaria del resultado y el carácter de punto decimal se eliminará si no queda

parte fraccionaria.

La función printf ()

Se accede a través de la inclusión de `<stdio.h>` , la función `printf()` es la herramienta principal utilizada para imprimir texto en la consola en C.

```
printf("Hello world!");
// Hello world!
```

Los arreglos de caracteres normales y sin formato se pueden imprimir solos colocándolos directamente entre los paréntesis.

```
printf("%d is the answer to life, the universe, and everything.", 42);
// 42 is the answer to life, the universe, and everything.

int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

Alternativamente, los enteros, los números de punto flotante, los caracteres y más pueden imprimirse usando el carácter de escape `%` , seguido de un carácter o secuencia de caracteres que denotan el formato, conocido como el *especificador de formato* .

Todos los argumentos adicionales a la función `printf()` están separados por comas, y estos argumentos deben estar en el mismo orden que los especificadores de formato. Los argumentos adicionales se ignoran, mientras que los argumentos escritos incorrectamente o la falta de argumentos causarán errores o comportamiento indefinido. Cada argumento puede ser un valor literal o una variable.

Después de la ejecución exitosa, la cantidad de caracteres impresos se devuelve con el tipo `int` . De lo contrario, un fallo devuelve un valor negativo.

Modificadores de longitud

Los estándares C99 y C11 especifican los siguientes modificadores de longitud para `printf()` ; sus significados son:

Modificador	Modifica	Se aplica a
S.S	d, i, o, u, x, o x	`char` , `signed char` o `unsigned char`
h	d, i, o, u, x, o x	`short int` o `unsigned short int`
l	d, i, o, u, x, o x	`long int` o `unsigned long int`
l	a, A, e, E, f, F, g, o G	`double` (para compatibilidad con `scanf()` ; indefinido en C90)

Modificador	Modifica	Se aplica a
ll	d, i, o, u, x, o x	`long long int` o `unsigned long long int`
j	d, i, o, u, x, o x	`intmax_t` o `uintmax_t`
z	d, i, o, u, x, o x	`size_t` o el tipo firmado correspondiente (`ssize_t` en POSIX)
t	d, i, o, u, x, o x	`ptrdiff_t` o el tipo entero sin signo correspondiente
L	a, A, e, E, f, F, g, o G	`long double`

Si aparece un modificador de longitud con cualquier especificador de conversión distinto al especificado anteriormente, el comportamiento no está definido.

Microsoft especifica algunos modificadores de longitud diferentes y explícitamente no admite `hh` , `j` , `z` o `t` .

Modificador	Modifica	Se aplica a
I32	d, i, o, x, o x	`__int32`
I32	o, u, x, o x	`unsigned __int32`
I64	d, i, o, x, o x	`__int64`
I64	o, u, x, o x	`unsigned __int64`
yo	d, i, o, x, o x	`ptrdiff_t` (es decir, `__int32` en plataformas de 32 bits, `__int64` en plataformas de 64 bits)
yo	o, u, x, o x	`size_t` (es decir, `unsigned __int32` en plataformas de 32 bits, `unsigned __int64` en plataformas de 64 bits)
l o l	a, A, e, E, f, g, o G	`long double` (en Visual C ++, aunque el `long double` es un tipo distinto, tiene la misma representación interna que el `double`).
l o w	c o c	Carácter ancho con funciones `printf` y `wprintf` . (Un especificador de tipo `lc` , `lC` , `wc` o `wC` es sinónimo de `C` en las funciones de `printf` y de `c` en `wprintf` funciones de `wprintf`).
l o w	s, s o z	Cadena de caracteres `wprintf` funciones `printf` y `wprintf` . (Un especificador de tipo `ls` , `lS` , `ws` o `wS` es sinónimo de `S` en las funciones de `printf` y de `s` en `wprintf` funciones de `wprintf`).

Tenga en cuenta que el `c` , `s` , y `Z` especificadores de conversión y la `I` , `I32` , `I64` , y `w` modificadores de longitud son extensiones de Microsoft. Tratar a `l` como un modificador para el `long double` lugar de `double` es diferente del estándar, aunque será difícil encontrar la diferencia a menos que el `long double` tenga una representación diferente del `double` .

Formato de impresión de banderas

El estándar C (C11 y C99 también) define los siguientes indicadores para `printf()` :

Bandera	Conversiones	Sentido
`-`	todos	El resultado de la conversión se justificará a la izquierda dentro del campo. La conversión está justificada a la derecha si no se especifica este indicador.
`+`	numérico firmado	El resultado de una conversión firmada siempre comenzará con un signo ('+' o '-'). La conversión comenzará con un signo solo cuando se convierta un valor negativo si no se especifica este indicador.
`<space>`	numérico firmado	Si el primer carácter de una conversión firmada no es un signo o si una conversión firmada no produce ningún carácter, se colocará un prefijo `<space>` sobre el resultado. Esto significa que si los indicadores `<space>` y ' `+` ' aparecen ambos, el indicador `<space>` se ignorará.
`#`	todos	Especifica que el valor se convertirá a una forma alternativa. Para la conversión `o` , aumentará la precisión, si y solo si es necesario, para forzar que el primer dígito del resultado sea cero (si el valor y la precisión son ambos 0, se imprime un solo 0). Para los especificadores de conversión `x` o `X` , un resultado distinto de cero tendrá `0x` (o `0X`) prefijados. Para los especificadores de conversión `a` , `A` , `e` , `E` , `f` , `F` , `g` y `G` , el resultado siempre contendrá un carácter de raíz, incluso si ningún dígito sigue al carácter de raíz. Sin este indicador, un carácter de raíz aparece en el resultado de estas conversiones solo si un dígito lo sigue. Para los especificadores de conversión `g` y `G` , los ceros finales no se eliminarán del resultado como lo son normalmente. Para otros especificadores de conversión, el comportamiento no está definido.
`0`	numérico	Para los especificadores de conversión d, i, o, u, x, X, a, A, e, E, f, F, g y G, se utilizan ceros a la izquierda (después de cualquier indicación de signo o base) para rellenar el campo ancho en lugar de realizar relleno de espacio, excepto al convertir un infinito o NaN. Si los indicadores '0' y '-' aparecen ambos, el indicador '0' se ignora. Para los especificadores de conversión

Bandera	Conversiones	Sentido
		d, i, o, u, xy X, si se especifica una precisión, se ignorará el indicador '0'. Si aparecen los indicadores '0' y `<apostrophe>` , los caracteres de agrupación se insertan antes del relleno cero. Para otras conversiones, el comportamiento es indefinido.

Microsoft también admite estos indicadores con los mismos significados.

La especificación POSIX para `printf()` agrega:

Bandera	Conversiones	Sentido
'	i, d, u, f, F, g, G	La parte entera del resultado de una conversión decimal se formateará con miles de caracteres de agrupación. Para otras conversiones el comportamiento es indefinido. Se utiliza el carácter de agrupación no monetaria.

Lea Entrada / salida formateada en línea: https://riptutorial.com/es/c/topic/3750/entrada---salida-formateada

Capítulo 29: Enumeraciones

Observaciones

Las enumeraciones consisten en la palabra clave `enum` y un *identificador* opcional seguido de una *lista de* enumeradores entre llaves.

Un *identificador* es de tipo `int` .

La *lista de enumeradores* tiene al menos un elemento de *enumerador* .

A un *enumerador* se le puede "asignar" opcionalmente una expresión constante de tipo `int` .

Un *enumerador* es constante y es compatible con un `char` , un entero con signo o un entero sin signo. Lo que se usa siempre está definido por la implementación . En cualquier caso, el tipo utilizado debe poder representar todos los valores definidos para la enumeración en cuestión.

Si no se "asigna" una expresión constante a un *enumerador* y es la primera entrada en una *lista de enumeradores* , toma el valor de `0` , de lo contrario, toma el valor de la entrada anterior en la *lista de enumeradores* más 1.

El uso de múltiples "asignaciones" puede llevar a que diferentes *enumeradores* de la misma enumeración tengan los mismos valores.

Examples

Enumeración simple

Una enumeración es un tipo de datos definido por el usuario que consta de constantes integrales y cada constante integral recibe un nombre. La `enum` palabras clave se utiliza para definir el tipo de datos enumerados.

Si usa `enum` lugar de `int` o `string/ char*` , aumenta la verificación en tiempo de compilación y evita que los errores pasen en constantes no válidas, y documenta qué valores son legales usar.

Ejemplo 1

```
enum color{ RED, GREEN, BLUE };

void printColor(enum color chosenColor)
{
    const char *color_name = "Invalid color";
    switch (chosenColor)
    {
       case RED:
         color_name = "RED";
         break;
```

```
        case GREEN:
         color_name = "GREEN";
         break;

        case BLUE:
         color_name = "BLUE";
         break;
    }
    printf("%s\n", color_name);
}
```

Con una función principal definida como sigue (por ejemplo):

```
int main(){
    enum color chosenColor;
    printf("Enter a number between 0 and 2");
    scanf("%d", (int*)&chosenColor);
    printColor(chosenColor);
    return 0;
}
```

C99

Ejemplo 2

(Este ejemplo utiliza inicializadores designados que están estandarizados desde C99).

```
enum week{ MON, TUE, WED, THU, FRI, SAT, SUN };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   printf("%s\n", dow[day]);
}
```

El mismo ejemplo usando la verificación de rango:

```
enum week{ DOW_INVALID = -1,
  MON, TUE, WED, THU, FRI, SAT, SUN,
  DOW_MAX };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   assert(day > DOW_INVALID && day < DOW_MAX);
   printf("%s\n", dow[day]);
}
```

Typedef enum

Hay varias posibilidades y convenciones para nombrar una enumeración. La primera es usar un *nombre de etiqueta* justo después de la palabra clave `enum` .

```
enum color
{
    RED,
    GREEN,
    BLUE
};
```

Esta enumeración se debe usar siempre con la palabra clave *y* la etiqueta como esta:

```
enum color chosenColor = RED;
```

Si usamos `typedef` directamente al declarar la `enum` , podemos omitir el nombre de la etiqueta y luego usar el tipo sin la palabra clave `enum` :

```
typedef enum
{
    RED,
    GREEN,
    BLUE
} color;

color chosenColor = RED;
```

Pero en este último caso no podemos usarlo como `enum color` , porque no usamos el nombre de la etiqueta en la definición. Una convención común es usar ambos, de modo que se pueda usar el mismo nombre con o sin la palabra clave `enum` . Esto tiene la ventaja particular de ser compatible con C ++

```
enum color                /* as in the first example */
{
    RED,
    GREEN,
    BLUE
};
typedef enum color color; /* also a typedef of same identifier */

color chosenColor  = RED;
enum color defaultColor = BLUE;
```

Función:

```
void printColor()
{
    if (chosenColor == RED)
    {
        printf("RED\n");
    }
    else if (chosenColor == GREEN)
    {
        printf("GREEN\n");
    }
```

```
    else if (chosenColor == BLUE)
    {
        printf("BLUE\n");
    }
}
```

Para más información sobre `typedef` ver Typedef

Enumeración con valor duplicado

Un valor de enumeración de ninguna manera debe ser único:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */


enum Dupes
{
   Base, /* Takes 0 */
   One, /* Takes Base + 1 */
   Two, /* Takes One + 1 */
   Negative = -1,
   AnotherZero /* Takes Negative + 1 == 0, sigh */
};

int main(void)
{
  printf("Base = %d\n", Base);
  printf("One = %d\n", One);
  printf("Two = %d\n", Two);
  printf("Negative = %d\n", Negative);
  printf("AnotherZero = %d\n", AnotherZero);

  return EXIT_SUCCESS;
}
```

La muestra se imprime:

```
Base = 0
One = 1
Two = 2
Negative = -1
AnotherZero = 0
```

constante de enumeración sin nombre tipográfico

Los tipos de enumeración también se pueden declarar sin darles un nombre:

```
  enum { buffersize = 256, };
  static unsigned char buffer [buffersize] = { 0 };
```

Esto nos permite definir constantes de tiempo de compilación de tipo `int` que, como en este ejemplo, se pueden usar como longitud de matriz.

Lea Enumeraciones en línea: https://riptutorial.com/es/c/topic/5460/enumeraciones

Capítulo 30: Errores comunes

Introducción

Esta sección analiza algunos de los errores comunes que un programador de C debería conocer y debería evitar cometer. Para más información sobre algunos problemas inesperados y sus causas, consulte Comportamiento indefinido

Examples

Mezclar enteros con signo y sin signo en operaciones aritméticas

Por lo general no es una buena idea mezclar `signed` y `unsigned` enteros en las operaciones aritméticas. Por ejemplo, ¿cuál será la salida del siguiente ejemplo?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a is more than b");
    else puts("a is less or equal than b");

    return 0;
}
```

Dado que 1000 es más de -1, se esperaría que la salida fuera `a is more than b` , pero ese no será el caso.

Las operaciones aritméticas entre diferentes tipos integrales se realizan dentro de un tipo común definido por las llamadas conversiones aritméticas habituales (consulte la especificación del lenguaje, 6.3.1.8).

En este caso, el "tipo común" es `unsigned int` , porque, como se indica en las conversiones aritméticas habituales ,

> 714 De lo contrario, si el operando que tiene un tipo entero sin signo tiene un rango mayor o igual al rango del tipo del otro operando, entonces el operando con tipo entero con signo se convierte al tipo del operando con tipo entero sin signo.

Esto significa que `int` operando `b` se convertirá a `unsigned int` antes de la comparación.

Cuando -1 se convierte en un `unsigned int` el resultado es el máximo valor de `unsigned int` posible, que es mayor que 1000, lo que significa que `a > b` es falso.

Escribir erróneamente = en lugar de == al comparar

El operador `=` se usa para la asignación.

El operador `==` se utiliza para la comparación.

Uno debe tener cuidado de no mezclar los dos. A veces uno escribe erróneamente

```
/* assign y to x */
if (x = y) {
     /* logic */
}
```

cuando lo que realmente se quería es:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

El primero asigna un valor de y a x y verifica si ese valor no es cero, en lugar de hacer una comparación, que es equivalente a:

```
if ((x = y) != 0) {
    /* logic */
}
```

Hay ocasiones en las que se pretende probar el resultado de una asignación y se usa comúnmente, porque evita tener que duplicar el código y tener que tratar la primera vez especialmente. Comparar

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
        switch (c) {
        ...
        }
}
```

versus

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
        switch (c) {
        ...
        }
        c = getopt_long(argc, argv, short_options, long_options, &option_index);
}
```

Los compiladores modernos reconocerán este patrón y no avisarán cuando la asignación esté dentro de paréntesis como arriba, pero pueden advertir sobre otros usos. Por ejemplo:

```
if (x = y)         /* warning */

if ((x = y))       /* no warning */
```

```
if ((x = y) != 0)  /* no warning; explicit */
```

Algunos programadores utilizan la estrategia de poner la constante a la izquierda del operador (comúnmente llamadas condiciones de Yoda). Debido a que las constantes son valores, este estilo de condición hará que el compilador arroje un error si se usa el operador incorrecto.

```
if (5 = y) /* Error */

if (5 == y) /* No error */
```

Sin embargo, esto reduce considerablemente la legibilidad del código y no se considera necesario si el programador sigue las buenas prácticas de codificación en C, y no ayuda cuando se comparan dos variables, por lo que no es una solución universal. Además, muchos compiladores modernos pueden dar advertencias cuando el código está escrito con las condiciones de Yoda.

Uso incauto de punto y coma

Ten cuidado con los puntos y coma. Siguiente ejemplo

```
if (x > a);
   a = x;
```

en realidad significa:

```
if (x > a) {}
a = x;
```

lo que significa que `x` se asignará a `a` en cualquier caso, que podría no ser lo que querías originalmente.

A veces, faltar un punto y coma también causará un problema que no se nota:

```
if (i < 0)
    return
day = date[0];
hour = date[1];
minute = date[2];
```

Se omite el punto y coma detrás de la devolución, por lo que day = date [0] se devolverá.

Una técnica para evitar este y otros problemas similares es usar llaves en condicionales y bucles de varias líneas. Por ejemplo:

```
if (x > a) {
    a = x;
}
```

Olvidarse de asignar un byte extra para \ 0

Cuando esté copiando una cadena en un búfer `malloc` ed, recuerde siempre agregar 1 a `strlen` .

```
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

Esto se debe a que `strlen` no incluye el final `\0` en la longitud. Si adopta el enfoque `WRONG` (como se muestra arriba), al llamar a `strcpy` , su programa invocará un comportamiento indefinido.

También se aplica a situaciones en las que estás leyendo una cadena de longitud máxima conocida desde la `stdin` o alguna otra fuente. Por ejemplo

```
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* WRONG */
char buffer[MAX_INPUT_LEN + 1]; /* RIGHT */

scanf("%42s", buffer);  /* Ensure that the buffer is not overflowed */
```

Olvidando liberar memoria (fugas de memoria)

Una buena práctica de programación es liberar cualquier memoria que haya sido asignada directamente por su propio código, o implícitamente llamando a una función interna o externa, como una API de biblioteca como `strdup()` . Si no se puede liberar memoria, se puede producir una pérdida de memoria, que podría acumularse en una cantidad sustancial de memoria desperdiciada que no está disponible para su programa (o el sistema), lo que posiblemente provoque bloqueos o un comportamiento indefinido. Es más probable que ocurran problemas si la fuga se incurre repetidamente en un bucle o función recursiva. El riesgo de fallo del programa aumenta cuanto más se ejecuta un programa con fugas. A veces los problemas aparecen al instante; otras veces, los problemas no se verán durante horas o incluso años de funcionamiento constante. Las fallas en el agotamiento de la memoria pueden ser catastróficas, dependiendo de las circunstancias.

El siguiente bucle infinito es un ejemplo de una fuga que eventualmente agotará la fuga de memoria disponible al llamar a `getline()` , una función que asigna implícitamente una nueva memoria, sin liberar esa memoria.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    /* The loop below leaks memory as fast as it can */

    for(;;) {
        getline(&line, &size, stdin); /* New memory implicitly allocated */

        /* <do whatever> */
```

```
        line = NULL;
    }

    return 0;
}
```

En contraste, el código a continuación también usa la función `getline()` , pero esta vez, la memoria asignada se libera correctamente, evitando una fuga.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
            line = NULL;

            /* Handle failure such as setting flag, breaking out of loop and/or exiting */
        }

        /* <do whatever> */

        free(line);
        line = NULL;

    }

    return 0;
}
```

La pérdida de memoria no siempre tiene consecuencias tangibles y no es necesariamente un problema funcional. Si bien las "mejores prácticas" exigen que se libere rigurosamente la memoria en puntos y condiciones estratégicas, para reducir la huella de la memoria y reducir el riesgo de agotamiento de la memoria, puede haber excepciones. Por ejemplo, si un programa está limitado en duración y alcance, el riesgo de falla en la asignación podría considerarse demasiado pequeño para preocuparse. En ese caso, eludir la desasignación explícita podría considerarse aceptable. Por ejemplo, la mayoría de los sistemas operativos modernos liberan automáticamente toda la memoria consumida por un programa cuando finaliza, ya sea debido a un fallo del programa, a una llamada del sistema a `exit()` , a la finalización del proceso o al final de `main()` . Liberar explícitamente la memoria en el momento de la finalización inminente del programa podría ser redundante o introducir una penalización de rendimiento.

La asignación puede fallar si no hay suficiente memoria disponible, y las fallas en el manejo deben contabilizarse en los niveles apropiados de la pila de llamadas. `getline()` , que se muestra arriba, es un caso de uso interesante porque es una función de biblioteca que no solo asigna la memoria que deja al llamante gratis, sino que puede fallar por varias razones, todas las cuales deben tenerse en cuenta. Por lo tanto, es esencial al usar una API de C, leer la *documentación (página de manual)* y prestar especial atención a las condiciones de error y al uso de la memoria,

y tener en cuenta qué capa de software tiene la responsabilidad de liberar la memoria devuelta.

Otra práctica común en el manejo de la memoria es establecer constantemente los punteros de la memoria en NULL inmediatamente después de que se libere la memoria a la que hacen referencia dichos punteros, de modo que se pueda probar la validez de los punteros en cualquier momento (por ejemplo, se verificó si no tenía NULL). puede llevar a problemas graves, como obtener datos de basura (operación de lectura) o corrupción de datos (operación de escritura) y / o un bloqueo del programa. En la mayoría de los sistemas operativos modernos, liberar la ubicación de memoria 0 (`NULL`) es un NOP (por ejemplo, es inofensivo), como lo exige el estándar C, por lo que al establecer un puntero en NULL, no hay riesgo de duplicar la memoria si el puntero Se pasa a `free()` . Tenga en cuenta que la doble liberación de la memoria puede llevar a fallas de tiempo, confusas y *difíciles de diagnosticar* .

Copiando demasiado

```
char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */
```

Si el usuario introduce una cadena de más de 7 caracteres (- 1 para el terminador nulo), la memoria detrás de la memoria intermedia `buf` será sobrescrito. Esto resulta en un comportamiento indefinido. Los piratas informáticos malintencionados a menudo explotan esto para sobrescribir la dirección de retorno y cambiarla a la dirección del código malicioso del pirata informático.

Olvidando copiar el valor de retorno de realloc en un temporal

Si falla `realloc` , devuelve `NULL` . Si asigna el valor del búfer original al valor de retorno de `realloc` , y si devuelve `NULL` , el búfer original (el puntero antiguo) se pierde, lo que provoca una *pérdida de memoria* . La solución consiste en copiar en un puntero temporal, y si eso no es NULL temporal, **luego** copia en el buffer real.

```
char *buf, *tmp;

buf = malloc(...);
...

/* WRONG */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");

/* RIGHT */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

Comparando números de punto flotante

Los tipos de punto flotante (`float` , `double` y `double long double`) no pueden representar con precisión algunos números porque tienen una precisión finita y representan los valores en un formato binario. Al igual que hemos repetido decimales en la base 10 para fracciones como 1/3, hay fracciones que no se pueden representar finamente en binario (como 1/3, pero también, lo que es más importante, 1/10). No compare directamente los valores de punto flotante; usa un delta en su lugar.

```
#include <float.h> // for DBL_EPSILON and FLT_EPSILON
#include <math.h>  // for fabs()

int main(void)
{
    double a = 0.1; // imprecise: (binary) 0.000110...

    // may be false or true
    if (a + a + a + a + a + a + a + a + a + a == 1.0) {
        printf("10 * 0.1 is indeed 1.0. This is not guaranteed in the general case.\n");
    }

    // Using a small delta value.
    if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 guarantees at least 10 decimal digits
        // of precision for the double type.
        printf("10 * 0.1 is almost 1.0.\n");
    }

    return 0;
}
```

Otro ejemplo:

```
gcc -O3   -g   -I./inc   -std=c11   -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-
prototypes  -Wold-style-definition       rd11.c -o rd11 -L./lib -lsoq
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f within tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        else
            printf("%d:%.10f <=> %.10f out of tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        epsilon /= 10.0;
    }
    return 0;
```

```
}
```

Salida:

```
0:3.1415926536 <=> 3.1415929204 within tolerance 1.0000000000 (rel diff 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 within tolerance 0.1000000000 (rel diff 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 within tolerance 0.0100000000 (rel diff 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 within tolerance 0.0010000000 (rel diff 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 within tolerance 0.0001000000 (rel diff 8.4914E-08)
5:3.1415926536 <=> 3.1415929204 within tolerance 0.0000100000 (rel diff 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 within tolerance 0.0000010000 (rel diff 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 within tolerance 0.0000001000 (rel diff 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000100 (rel diff 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000010 (rel diff 8.4914E-08)
```

Haciendo escala adicional en aritmética de punteros.

En la aritmética de punteros, el número entero que se agrega o se resta al puntero se interpreta no como un cambio de *dirección* sino como el número de *elementos* a mover.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* wrong */
    printf("%d %d\n", *ptr, *ptr2);
    return 0;
}
```

Este código realiza una escala adicional al calcular el puntero asignado a `ptr2` . Si `sizeof(int)` es 4, lo que es típico en entornos modernos de 32 bits, la expresión significa "8 elementos después de la `array[0]` ", que está fuera de rango, e invoca *un comportamiento indefinido* .

Para tener el punto `ptr2` en lo que es 2 elementos después de la `array[0]` , simplemente debe agregar 2.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

La aritmética de punteros explícita con operadores aditivos puede ser confusa, por lo que el uso de subíndices de matriz puede ser mejor.

```
#include <stdio.h>

int main(void) {
```

```
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

`E1[E2]` es idéntico a `(*((E1)+(E2)))` (N1570 6.5.2.1, párrafo 2), y `&(E1[E2])` es equivalente a `((E1)+(E2))` (N1570 6.5.3.2, nota 102).

Alternativamente, si se prefiere la aritmética de punteros, la conversión del puntero para direccionar un tipo de datos diferente puede permitir el direccionamiento de bytes. Sin embargo, tenga cuidado: el endianness puede convertirse en un problema, y la conversión a tipos que no sean 'puntero a carácter' conduce a problemas estrictos de aliasing .

```
#include <stdio.h>

int main(void) {
    int array[3] = {1,2,3};  // 4 bytes * 3 allocated
    unsigned char *ptr = (unsigned char *) array;  // unsigned chars only take 1 byte
    /*
     * Now any pointer arithmetic on ptr will match
     * bytes in memory.  ptr can be treated like it
     * was declared as: unsigned char ptr[12];
     */

    return 0;
}
```

Las macros son simples reemplazos de cadena

Las macros son simples reemplazos de cadena. (Estrictamente hablando, trabajan con tokens de preprocesamiento, no con cadenas arbitrarias).

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Usted puede esperar que este código para imprimir `9` (`3*3`), pero en realidad `5` será impreso porque la macro se expandirá a `1+2*1+2` .

Debería envolver los argumentos y toda la expresión de la macro entre paréntesis para evitar este problema.

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))
```

```
int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Otro problema es que no se garantiza que los argumentos de una macro se evalúen una vez; pueden no ser evaluados en absoluto, o pueden ser evaluados varias veces.

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d\n", MIN(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

En este código, la macro se expandirá a `((a++) <= (10) ? (a++) : (10))` . Como `a++` (`0`) es menor que `10` , `a++` se evaluará dos veces y hará que el valor de `a` y lo que se devuelva desde `MIN` difiera de lo que puede esperar.

Esto se puede evitar mediante el uso de funciones, pero tenga en cuenta que los tipos serán corregidos por la definición de la función, mientras que las macros pueden ser (también) flexibles con los tipos.

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d\n", min(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Ahora el problema de la doble evaluación está solucionado, pero esta función `min` no puede tratar con datos `double` sin truncar, por ejemplo.

Las directivas macro pueden ser de dos tipos:

```
#define OBJECT_LIKE_MACRO     followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

Lo que distingue estos dos tipos de macros es el carácter que sigue al identificador después de `#define` : si es un *lparen* , es una macro similar a una función; de lo contrario, es una macro similar a un objeto. Si la intención es escribir una macro similar a una función, no debe haber ningún espacio en blanco entre el final del nombre de la macro y `(` . Verifique esto para obtener una explicación detallada.

C99

En C99 o posterior, puede usar `static inline int min(int x, int y) { … }` .

C11

En C11, podría escribir una expresión 'genérica de tipo' para `min` .

```
#include <stdio.h>

#define min(x, y) _Generic((x), \
                        long double: min_ld, \
                        unsigned long long: min_ull, \
                        default: min_i \
                        )(x, y)

#define gen_min(suffix, type) \
    static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));
    long double ld1 = 3.141592653L;
    long double ld2 = 3.141592652L;
    printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));
    int i1 = 3141653;
    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));
    return 0;
}
```

La expresión genérica podría extenderse con más tipos, como `double` , `float` , `long long` , `unsigned long` , `long` , `unsigned` , y las `gen_min` macro `gen_min` apropiadas escritas.

Errores de referencia indefinidos al enlazar

Uno de los errores más comunes en la compilación ocurre durante la etapa de vinculación. El error se parece a esto:

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: In function `main':
undefined_reference.c:(.text+0x15): undefined reference to `foo'
collect2: error: ld returned 1 exit status
$
```

Así que echemos un vistazo al código que generó este error:

```
int foo(void);

int main(int argc, char **argv)
```

```
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Aquí vemos una *declaración* de foo (`int foo();`) pero ninguna *definición* de ella (función real). Por lo tanto, le proporcionamos al compilador el encabezado de la función, pero no se definió esa función en ninguna parte, por lo que la etapa de compilación pasa, pero el vinculador sale con un error de `Undefined reference` no definido.
Para corregir este error en nuestro pequeño programa, solo tendríamos que agregar una *definición* para foo:

```
/* Declaration of foo */
int foo(void);

/* Definition of foo */
int foo(void)
{
    return 5;
}

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Ahora este código se compilará. Surge una situación alternativa donde la fuente de `foo()` está en un archivo fuente separado `foo.c` (y hay un encabezado `foo.h` para declarar `foo()` que se incluye tanto en `foo.c` como en `undefined_reference.c`). Luego, la solución es vincular tanto el archivo objeto de `foo.c` como `undefined_reference.c` , o compilar ambos archivos fuente:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

O:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

Un caso más complejo es donde están involucradas las bibliotecas, como en el código:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
```

```
    double second;
    double power;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Translate user input to numbers, extra error checking
     * should be done here. */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* Use function pow() from libm - this will cause a linkage
     * error unless this code is compiled against libm! */
    power = pow(first, second);

    printf("%f to the power of %f = %f\n", first, second, power);

    return EXIT_SUCCESS;
}
```

El código es sintácticamente correcto, la declaración para `pow()` existe desde `#include <math.h>` , por lo que tratamos de compilar y vincular pero obtenemos un error como este:

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: In function `main':
no_library_in_link.c:(.text+0x8b): undefined reference to `pow'
collect2: error: ld returned 1 exit status
$
```

Esto sucede porque no se encontró la *definición* de `pow()` durante la etapa de enlace. Para arreglar esto, tenemos que especificar que queremos vincular con la biblioteca matemática llamada `libm` especificando el indicador `-lm` . (Tenga en cuenta que hay plataformas como macOS donde no se necesita `-lm` , pero cuando obtiene la referencia no definida, se necesita la biblioteca).

Así que ejecutamos la etapa de compilación nuevamente, esta vez especificando la biblioteca (después de los archivos de origen o de objeto):

```
$ gcc no_library_in_link.c -lm -o library_in_link_cmd
$ ./library_in_link_cmd 2 4
2.000000 to the power of 4.000000 = 16.000000
$
```

¡Y funciona!

Malentendido decaimiento de la matriz

Un problema común en el código que utiliza matrices multidimensionales, matrices de punteros, etc. es el hecho de que `Type**` y `Type[M][N]` son tipos fundamentalmente diferentes:

```
#include <stdio.h>

void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(s, 4);
    return 0;
}
```

Ejemplo de salida del compilador:

```
file1.c: In function 'main':
file1.c:13:23: error: passing argument 1 of 'print_strings' from incompatible pointer type [-
Wincompatible-pointer-types]
         print_strings(strings, 4);
                       ^
file1.c:3:10: note: expected 'char **' but argument is of type 'char (*)[20]'
     void print_strings(char **strings, size_t n)
```

El error indica que la matriz `s` en la función `main` se pasa a la función `print_strings` , que espera un tipo de puntero diferente del que recibió. También incluye una nota que expresa el tipo que se espera por `print_strings` y el tipo que se le pasó desde `main` .

El problema se debe a algo llamado *decaimiento de la matriz* . Lo que sucede cuando se pasa `s` con su tipo `char[4][20]` (matriz de 4 matrices de 20 caracteres) a la función es que se convierte en un puntero a su primer elemento como si hubiera escrito `&s[0]` , que tiene el tipo `char (*)[20]` (puntero a 1 matriz de 20 caracteres). Esto ocurre para cualquier matriz, incluida una matriz de punteros, una matriz de matrices de matrices (matrices en 3D) y una matriz de punteros a una matriz. A continuación se muestra una tabla que ilustra lo que sucede cuando una matriz se descompone. Los cambios en la descripción del tipo se resaltan para ilustrar lo que sucede:

Antes de la decadencia		Despues de la decadencia	
`char [20]`	**matriz de (20 caracteres)**	`char *`	**puntero a (1 char)**
`char [4][20]`	**matriz de (4 matrices** de 20 caracteres)	`char (*)[20]`	**puntero a (1 matriz** de 20 caracteres)
`char *[4]`	**matriz de (4 punteros** a 1 carácter)	`char **`	**puntero a (1 puntero** a 1 carácter)
`char [3][4][20]`	**matriz de (3 matrices** de 4 matrices de 20 caracteres)	`char (*)[4][20]`	**puntero a (1 matriz** de 4 matrices de 20 caracteres)

Antes de la decadencia		Despues de la decadencia	
`char (*[4])[20]`	**matriz de (4 punteros** a 1 matriz de 20 caracteres)	`char (**)[20]`	**puntero a (1 puntero** a 1 matriz de 20 caracteres)

Si una matriz se puede descomponer en un puntero, se puede decir que un puntero puede considerarse una matriz de al menos 1 elemento. Una excepción a esto es un puntero nulo, que apunta a nada y, en consecuencia, no es una matriz.

La desintegración de la matriz solo ocurre una vez. Si una matriz se ha degradado a un puntero, ahora es un puntero, no una matriz. Incluso si tiene un puntero a una matriz, recuerde que el puntero podría considerarse una matriz de al menos un elemento, por lo que ya se ha producido una caída de la matriz.

En otras palabras, un puntero a una matriz (`char (*)[20]`) nunca se convertirá en un puntero a un puntero (`char **`). Para arreglar la función `print_strings` , simplemente haga que reciba el tipo correcto:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

Surge un problema cuando desea que la función `print_strings` sea genérica para cualquier conjunto de caracteres: ¿qué `print_strings` si hay 30 caracteres en lugar de 20? O 50? La respuesta es agregar otro parámetro antes del parámetro de matriz:

```
#include <stdio.h>

/*
 * Note the rearranged parameters and the change in the parameter name
 * from the previous definitions:
 *      n (number of strings)
 *   => scount (string count)
 *
 * Of course, you could also use one of the following highly recommended forms
 * for the `strings` parameter instead:
 *
 *    char strings[scount][ccount]
 *    char strings[][ccount]
 */
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(4, 20, s);
    return 0;
```

```
}
```

Compilarlo no produce errores y da como resultado el resultado esperado:

```
Example 1
Example 2
Example 3
Example 4
```

Pasar matrices no adyacentes a funciones que esperan matrices multidimensionales "reales"

Cuando se asignan matrices multidimensionales con `malloc` , `calloc` y `realloc` , un patrón común es asignar las matrices internas con múltiples llamadas (incluso si la llamada solo aparece una vez, puede estar en un bucle):

```
/* Could also be `int **` with malloc used to allocate outer array. */
int *array[4];
int i;

/* Allocate 4 arrays of 16 ints. */
for (i = 0; i < 4; i++)
    array[i] = malloc(16 * sizeof(*array[i]));
```

La diferencia en bytes entre el último elemento de una de las matrices internas y el primer elemento de la siguiente matriz interna puede no ser 0 como lo sería con una matriz multidimensional "real" (por ejemplo, `int array[4][16];`) :

```
/* 0x40003c, 0x402000 */
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Teniendo en cuenta el tamaño de `int` , obtiene una diferencia de 8128 bytes (8132-4), que es 2032 elementos de matriz de tamaño `int` , y ese es el problema: una matriz multidimensional "real" no tiene espacios entre los elementos.

Si necesita usar una matriz asignada dinámicamente con una función que espera una matriz multidimensional "real", debe asignar un objeto de tipo `int *` y usar la aritmética para realizar cálculos:

```
void func(int M, int N, int *array);
...

/* Equivalent to declaring `int array[M][N] = {{0}};` and assigning to array4_16[i][j]. */
int *array;
int M = 4, N = 16;
array = calloc(M, N * sizeof(*array));
array[i * N + j] = 1;
func(M, N, array);
```

Si `N` es una macro o un literal entero en lugar de una variable, el código simplemente puede usar la notación de matriz 2-D más natural después de asignar un puntero a una matriz:

```
void func(int M, int N, int *array);
#define N 16
void func_N(int M, int (*array)[N]);
...

int M = 4;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;

/* Cast to `int *` works here because `array` is a single block of M*N ints with no gaps,
   just like `int array2[M * N];` and `int array3[M][N];` would be. */
func(M, N, (int *)array);
func_N(M, array);
```

C99

Si `N` no es una macro o un literal entero, la `array` apuntará a una matriz de longitud variable (VLA). Esto todavía se puede usar con `func` al lanzar `int *` y una nueva función `func_vla` reemplazaría `func_N` :

```
void func(int M, int N, int *array);
void func_vla(int M, int N, int array[M][N]);
...

int M = 4, N = 16;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
func(M, N, (int *)array);
func_vla(M, N, array);
```

C11

Nota : los VLA son opcionales a partir de C11. Si su implementación es compatible con C11 y define la macro `__STDC_NO_VLA__` a 1, está atascado con los métodos anteriores a C99.

Usando constantes de caracteres en lugar de cadenas literales, y viceversa

En C, las constantes de caracteres y los literales de cadena son cosas diferentes.

Un carácter rodeado de comillas simples como `'a'` es una *constante de carácter* . Una constante de carácter es un entero cuyo valor es el código de carácter que representa el carácter. La forma de interpretar constantes de caracteres con varios caracteres como `'abc'` está definida por la implementación.

Cero o más caracteres rodeados por comillas dobles como `"abc"` es una *cadena literal* . Un literal de cadena es una matriz no modificable cuyos elementos son tipo `char` . La cadena entre las comillas dobles y el carácter nulo que termina son los contenidos, por lo que `"abc"` tiene 4 elementos (`{'a', 'b', 'c', '\0'}`)

En este ejemplo, se usa una constante de caracteres donde se debe usar un literal de cadena. Esta constante de caracteres se convertirá en un puntero de una manera definida por la

implementación y hay pocas posibilidades de que el puntero convertido sea válido, por lo que este ejemplo invocará *un comportamiento indefinido* .

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

En este ejemplo, se usa un literal de cadena donde se debe usar una constante de caracteres. El puntero convertido de la cadena literal será convertido a un número entero de una manera definida por la implementación, y se puede convertir en `char` de una manera definida por la implementación. (Cómo convertir un entero a un tipo con signo que no puede representar el valor a convertir es definido por la implementación, y si `char` se firmó también es definido por la implementación.) La salida será algo sin sentido.

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

En casi todos los casos, el compilador se quejará de estas confusiones. Si no es así, debe usar más opciones de advertencia del compilador, o se recomienda que use un compilador mejor.

Ignorar los valores de retorno de las funciones de la biblioteca

Casi todas las funciones en la biblioteca estándar de C devuelven algo en el éxito, y algo más en el error. Por ejemplo, `malloc` devolverá un puntero al bloque de memoria asignado por la función en caso de éxito y, si la función no pudo asignar el bloque de memoria solicitado, un puntero nulo. Por lo tanto, siempre debe verificar el valor de retorno para una depuración más fácil.

Esto es malo:

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* This might invoke undefined behaviour and if lucky causes a segmentation
violation, unless your system has a lot of memory */
```

Esto es bueno:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
```

```
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {
        fprintf(stderr, "could not read string\n");
        free(x);
        exit(EXIT_FAILURE);
    }

    /* Do stuff with x. */

    /* Clean up. */
    free(x);

    return EXIT_SUCCESS;
}
```

De esta manera, sabrá de inmediato la causa del error; de lo contrario, podría pasar horas buscando un error en un lugar completamente incorrecto.

El carácter de nueva línea no se consume en una llamada a scanf () típica

Cuando este programa

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

se ejecuta con esta entrada

```
42
life
```

la salida será de `42 ""` lugar de las `42 "life"` esperadas `42 "life"` .

Esto se debe a que un carácter de nueva línea después de `42` no se consume en la llamada a `scanf()` y es consumido por `fgets()` antes de leer la `life` . Entonces, `fgets()` deja de leer antes de leer la `life` .

Para evitar este problema, una forma que es útil cuando se conoce la longitud máxima de una línea, cuando se resuelven problemas en el sistema de jueces en línea, por ejemplo, es evitar usar `scanf()` directamente y leer todas las líneas a través de `fgets()` . Puede usar `sscanf()` para

analizar las líneas leídas.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Otra forma es leer hasta que golpeas un carácter de nueva línea después de usar `scanf()` y antes de usar `fgets()` .

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != '\n' && c != EOF);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Añadiendo un punto y coma a un #define

Es fácil confundirse con el preprocesador de C y tratarlo como parte de C, pero eso es un error porque el preprocesador es solo un mecanismo de sustitución de texto. Por ejemplo, si escribes

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

el código se expande a

```
int arr[100;];
```

que es un error de sintaxis. El remedio es eliminar el punto y coma de la línea `#define` . Es casi invariablemente un error terminar un `#define` con un punto y coma.

Los comentarios multilínea no pueden ser anidados

En C, comentarios multilínea, / * y * /, no se anidan.

Si anota un bloque de código o función utilizando este estilo de comentario:

```
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

No podrás comentarlo fácilmente:

```
//Trying to comment out the block...
/*

/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

//Causes an error on the line below...
*/
```

Una solución es utilizar comentarios de estilo C99:

```
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
```

```
}
```

Ahora todo el bloque se puede comentar fácilmente:

```
/*

// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

*/
```

Otra solución es evitar deshabilitar el código usando la sintaxis de los comentarios, utilizando las directivas de preprocesador `#ifdef` o `#ifndef` lugar. Estas directivas *hacen* nido, por lo que podrá comentar su código en el estilo que prefiera.

```
#define DISABLE_MAX /* Remove or comment this line to enable max() code block */

#ifdef DISABLE_MAX
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
#endif
```

Algunas guías llegan al punto de recomendar que las secciones de código *nunca* deben ser comentadas y que si el código se desactiva temporalmente, se podría recurrir al uso de una directiva `#if 0` .

Vea #if 0 para bloquear secciones de código .

Superando los límites de la matriz

Las matrices se basan en cero, es decir, el índice siempre comienza en 0 y termina con la longitud de la matriz de índice menos 1. Por lo tanto, el siguiente código no generará el primer

elemento de la matriz y generará basura para el valor final que imprima.

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 1; x <= 5; x++) //Looping from 1 till 5.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Salida: `2 3 4 5 GarbageValue`

A continuación se muestra la forma correcta de lograr el resultado deseado:

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 0; x < 5; x++) //Looping from 0 till 4.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Salida: `1 2 3 4 5`

Es importante conocer la longitud de una matriz antes de trabajar con ella, ya que de lo contrario puede dañar el búfer o provocar un fallo de segmentación al acceder a las ubicaciones de memoria que están fuera de los límites.

Función recursiva - perdiendo la condición base

Calcular el factorial de un número es un ejemplo clásico de una función recursiva.

Falta la condición de base:

```
#include <stdio.h>

int factorial(int n)
{
       return n * factorial(n - 1);
}

int main()
{
```

```
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Salida típica: `Segmentation fault: 11`

El problema con esta función es que se produciría un bucle infinito, lo que provocaría un fallo de segmentación: necesita una condición base para detener la recursión.

Condición de base declarada:

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // Base Condition, very crucial in designing the recursive functions.
    {
       return 1;
    }
    else
    {
       return n * factorial(n - 1);
    }
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Salida de muestra

```
Factorial 3 = 6
```

Esta función terminará tan pronto como llegue a la condición `n` es igual a 1 (siempre que el valor inicial de `n` sea lo suficientemente pequeño - el límite superior es `12` cuando `int` es una cantidad de 32 bits).

Reglas a seguir:

1. Inicializa el algoritmo. Los programas recursivos a menudo necesitan un valor semilla para empezar. Esto se logra ya sea utilizando un parámetro pasado a la función o proporcionando una función de puerta de enlace que no sea recursiva pero que configure los valores semilla para el cálculo recursivo.
2. Compruebe si los valores actuales que se procesan coinciden con el caso base. Si es así, procese y devuelva el valor.
3. Redefinir la respuesta en términos de un subproblema o subproblemas más pequeños o más simples.
4. Ejecutar el algoritmo en el sub-problema.
5. Combina los resultados en la formulación de la respuesta.
6. Devuelve los resultados.

Fuente: Función Recursiva.

Comprobando la expresión lógica contra 'verdadero'

El estándar C original no tenía un tipo booleano intrínseco, por lo que `bool` , `true` y `false` no tenía un significado inherente y a menudo los definían los programadores. Normalmente, `true` se definiría como 1 y `false` se definiría como 0.

C99

C99 agrega el tipo `_Bool` y el encabezado `<stdbool.h>` que define `bool` (expandiéndose a `_Bool`), `false` y `true` . También le permite redefinir `bool` , `true` y `false` , pero observa que esta es una característica obsolescente.

Más importante aún, las expresiones lógicas tratan todo lo que se evalúa como cero como falso y cualquier evaluación que no sea cero como verdadera. Por ejemplo:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == true)  /* Comparison only succeeds if true is 0x80 and bitField
has that bit set */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

En el ejemplo anterior, la función está intentando verificar si el bit superior está establecido y devuelve `true` si lo está. Sin embargo, al verificar explícitamente contra `true` , la instrucción `if` solo tendrá éxito si `(bitfield & 0x80)` evalúa a lo que se define como `true` , que generalmente es `1` y muy raramente `0x80` . O bien compruebe explícitamente contra el caso que espera:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == 0x80) /* Explicitly test for the case we expect */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

O evalúe cualquier valor distinto de cero como verdadero.

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
```

```
{
    /* If upper bit is set, result is 0x80 which the if will evaluate as true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Los literales de punto flotante son de tipo doble por defecto.

Se debe tener cuidado al inicializar variables de tipo `float` a valores literales o al compararlas con valores literales, porque los literales de punto flotante regulares como `0.1` son de tipo `double` . Esto puede llevar a sorpresas:

```
#include <stdio.h>
int main() {
    float  n;
    n = 0.1;
    if (n > 0.1) printf("Wierd\n");
    return 0;
}
// Prints "Wierd" when n is float
```

Aquí, `n` se inicializa y se redondea a precisión simple, lo que da como resultado un valor de 0.10000000149011612. Luego, `n` se vuelve a convertir en doble precisión para compararse con `0.1` literal (lo que equivale a 0.10000000000000001), lo que resulta en una falta de coincidencia.

Además de los errores de redondeo, la mezcla de variables `float` con literales `double` dará como resultado un rendimiento deficiente en plataformas que no tienen soporte de hardware para la precisión doble.

Lea Errores comunes en línea: https://riptutorial.com/es/c/topic/2006/errores-comunes

Capítulo 31: Estructuras

Introducción

Las estructuras proporcionan una forma de agrupar un conjunto de variables relacionadas de diversos tipos en una sola unidad de memoria. La estructura en su conjunto puede ser referenciada por un solo nombre o puntero; También se puede acceder a los miembros de la estructura individualmente. Las estructuras se pueden pasar a funciones y se pueden devolver desde funciones. Se definen utilizando la palabra clave `struct` .

Examples

Estructuras de datos simples.

Los tipos de datos de estructura son una forma útil de empaquetar datos relacionados y hacer que se comporten como una sola variable.

Declarar una `struct` simple que contiene dos miembros `int` :

```
struct point
{
    int x;
    int y;
};
```

`x` e `y` se denominan *miembros* (o *campos*) de la estructura de `point` .

Definiendo y usando estructuras:

```
struct point p;    // declare p as a point struct
p.x = 5;           // assign p member variables
p.y = 3;
```

Las estructuras se pueden inicializar en la definición. Lo anterior es equivalente a:

```
struct point p = {5, 3};
```

Las estructuras también se pueden inicializar usando inicializadores designados .

El acceso a los campos también se realiza utilizando el `.` operador

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

Estructuras Typedef

Combinar `typedef` con `struct` puede hacer que el código sea más claro. Por ejemplo:

```
typedef struct
{
    int x, y;
} Point;
```

Opuesto a:

```
struct Point
{
    int x, y;
};
```

Podría ser declarado como:

```
Point point;
```

en lugar de:

```
struct Point point;
```

Aún mejor es usar lo siguiente

```
typedef struct Point Point;

struct Point
{
    int x, y;
};
```

Para tener ventaja de ambas definiciones posibles de `point` . Tal declaración es más conveniente si aprendió C ++ primero, donde puede omitir la palabra clave `struct` si el nombre no es ambiguo.

`typedef` nombres de `typedef` para estructuras pueden estar en conflicto con otros identificadores de otras partes del programa. Algunos consideran esto como una desventaja, pero para la mayoría de las personas que tienen una `struct` y otro identificador, el mismo es bastante perturbador. Notorio es, por ejemplo, la `stat` POSIX

```
int stat(const char *pathname, struct stat *buf);
```

donde se ve una función `stat` que tiene un argumento que es `struct stat` .

`typedef` 'd structs sin un nombre de etiqueta siempre impone que toda la declaración de `struct` sea visible para el código que la usa. La declaración de `struct` completa se debe colocar en un archivo de encabezado.

Considerar:

```
#include "bar.h"

struct foo
```

```
{
    bar *aBar;
};
```

Así que con una `struct typedef` que no tiene nombre de etiqueta, el archivo `bar.h` siempre tiene que incluir la definición completa de `bar` . Si usamos

```
typedef struct bar bar;
```

en `bar.h` , los detalles de la estructura de la `bar` se pueden ocultar.

Ver Typedef

Punteros a estructuras

Cuando tiene una variable que contiene una `struct` , puede acceder a sus campos utilizando el operador de punto (`.`). Sin embargo, si tiene un puntero a una `struct` , esto no funcionará. Tienes que usar el operador de flecha (`->`) para acceder a sus campos. Aquí hay un ejemplo de una implementación terriblemente simple (algunos podrían decir "terrible y simple") de una pila que usa punteros para `struct` y muestra el operador de flecha.

```
#include <stdlib.h>
#include <stdio.h>

/* structs */
struct stack
{
    struct node *top;
    int size;
};

struct node
{
    int data;
    struct node *next;
};

/* function declarations */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;

    size_t i;

    /* allocate memory for a struct stack and record its pointer */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }
```

```
    /* initialize stack */
    stack->top = NULL;
    stack->size = 0;

    /* push 10 ints */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d\n", data);
            if (-1 == push(data, stack))
            {
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* pop 5 ints */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i\n", pop(stack));
        }
    }

    /* destroy stack */
    destroy(stack);

    return result;
}

/* Push a value onto the stack. */
/* Returns 0 on success and -1 on failure. */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* allocate memory for new node */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}

/* Pop a value off of the stack. */
```

```
/* Returns the value popped off the stack */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* destroy the stack */
void destroy(struct stack *stack)
{
    /* free all pointers */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}
```

Miembros de la matriz flexible

C99

Declaración de tipo

Una estructura *con al menos un miembro* puede contener adicionalmente un único miembro de la matriz de longitud no especificada al final de la estructura. Esto se llama un miembro de matriz flexible:

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
};

/* Merged ex2_header and ex2 structures. */
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

Efectos sobre el tamaño y el relleno

Se considera que un miembro de matriz flexible no tiene tamaño cuando se calcula el tamaño de una estructura, aunque todavía puede existir un relleno entre ese miembro y el miembro anterior de la estructura:

```
/* Prints "8,8" on my machine, so there is no padding. */
printf("%zu,%zu\n", sizeof(size_t), sizeof(struct ex1));

/* Also prints "8,8" on my machine, so there is no padding in the ex2 structure itself. */
printf("%zu,%zu\n", sizeof(struct ex2_header), sizeof(struct ex2));

/* Prints "5,8" on my machine, so there are 3 bytes of padding. */
printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

Se considera que el miembro de la matriz flexible tiene un tipo de matriz incompleta, por lo que su tamaño no se puede calcular utilizando `sizeof` .

Uso

Puede declarar e inicializar un objeto con un tipo de estructura que contiene un miembro de matriz flexible, pero no debe intentar inicializar el miembro de matriz flexible ya que se trata como si no existiera. Está prohibido intentar hacer esto, y se producirán errores de compilación.

De manera similar, no debe intentar asignar un valor a ningún elemento de un miembro de matriz flexible al declarar una estructura de esta manera ya que puede que no haya suficiente relleno al final de la estructura para permitir cualquier objeto requerido por el miembro de matriz flexible. Sin embargo, el compilador no necesariamente evitará que hagas esto, por lo que esto puede llevar a un comportamiento indefinido.

```
/* invalid: cannot initialize flexible array member */
struct ex1 e1 = {1, {2, 3}};
/* invalid: hdr={foo=1, bar=2} OK, but cannot initialize flexible array member */
struct ex2 e2 = {{1, 2}, {3}};
/* valid: initialize foo=1, bar=2 members */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* undefined behavior, in my case */
e3.flex[0] = 2; /* undefined behavior again */
e2.flex[0] = e3.flex[0]; /* undefined behavior */
```

En su lugar, puede optar por usar `malloc` , `calloc` o `realloc` para asignar la estructura con almacenamiento adicional y luego liberarla, lo que le permite usar el miembro de la matriz flexible como desee:

```
/* valid: allocate an object of structure type `ex1` along with an array of 2 ints */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* valid: allocate an object of structure type ex2 along with an array of 4 ints */
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));
```

```
/* valid: allocate 5 structure type ex3 objects along with an array of 3 ints per object */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* valid */
pe3[0]->flex[0] = pe1->flex[0]; /* valid */
```

C99

La 'estructura hack'

Los miembros de la matriz flexible no existían antes de C99 y se tratan como errores. Una solución común es declarar una matriz de longitud 1, una técnica llamada 'estructura':

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

Sin embargo, esto afectará el tamaño de la estructura, a diferencia de un verdadero miembro de matriz flexible:

```
/* Prints "8,4,16" on my machine, signifying that there are 4 bytes of padding. */
printf("%d,%d,%d\n", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

Para usar el miembro `flex` como un miembro de matriz flexible, lo asignaría con `malloc` como se muestra arriba, excepto que `sizeof(*pe1)` (o el `sizeof(*pe1)` equivalente de `sizeof(struct ex1)`) se reemplazaría con `offsetof(struct ex1, flex)` o el más largo, expresión de tipo agnóstico `sizeof(*pe1)-sizeof(pe1->flex)` . Alternativamente, puede restar 1 de la longitud deseada de la matriz "flexible" ya que ya está incluido en el tamaño de la estructura, suponiendo que la longitud deseada es mayor que 0. La misma lógica puede aplicarse a los otros ejemplos de uso.

Compatibilidad

Si se desea compatibilidad con compiladores que no admiten miembros de matriz flexible, puede usar una macro definida como `FLEXMEMB_SIZE` continuación:

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
};
```

Al asignar objetos, debe usar la forma `offsetof(struct ex1, flex)` para referirse al tamaño de la estructura (excluyendo el miembro de la matriz flexible) ya que es la única expresión que

permanecerá consistente entre los compiladores que admiten miembros de la matriz flexible y los compiladores que sí lo hacen. no:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

La alternativa es usar el preprocesador para restar condicionalmente 1 de la longitud especificada. Debido al aumento en el potencial de inconsistencia y error humano general en esta forma, moví la lógica a una función separada:

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* allocate an ex1 object with "flex" array of length 3 */
struct ex1 *pe1 = ex1_alloc(3);
```

Pasando estructuras a funciones

En C, todos los argumentos se pasan a las funciones por valor, incluidas las estructuras. Para estructuras pequeñas, esto es algo bueno, ya que significa que no hay sobrecarga por acceder a los datos a través de un puntero. Sin embargo, también hace que sea muy fácil pasar accidentalmente una estructura enorme que resulta en un rendimiento deficiente, especialmente si el programador está acostumbrado a otros lenguajes donde los argumentos se pasan por referencia.

```
struct coordinates
{
    int x;
    int y;
    int z;
};

// Passing and returning a small struct by value, very fast
struct coordinates move(struct coordinates position, struct coordinates movement)
{
    position.x += movement.x;
    position.y += movement.y;
    position.z += movement.z;
    return position;
}

// A very big struct
struct lotsOfData
{
    int param1;
    char param2[80000];
};
```

```
// Passing and returning a large struct by value, very slow!
// Given the large size of the struct this could even cause stack overflow
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// Passing the large struct by pointer instead, fairly fast
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}
```

Programación basada en objetos utilizando estructuras.

Las estructuras pueden usarse para implementar código de una manera orientada a objetos. Una estructura es similar a una clase, pero faltan las funciones que normalmente también forman parte de una clase, podemos agregarlas como variables de miembro de puntero a función. Para quedarse con nuestro ejemplo de coordenadas:

```
/* coordinates.h */

typedef struct coordinate_s
{
    /* Pointers to method functions */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* Data */
    int x;
    int y;
} coordinate;

/* Constructor */
coordinate *coordinate_create(void);
/* Destructor */
void coordinate_destroy(coordinate *this);
```

Y ahora el archivo C de implementación:

```
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* Constructor */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
    {
        c->setx = &coordinate_setx;
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
```

```
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* Destructor */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* Methods */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("Coordinate: (%i, %i)\n", this->x, this->y);
    }
    else
    {
        printf("NULL pointer exception!\n");
    }
}
```

Un ejemplo de uso de nuestra clase de coordenadas sería:

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* Create and initialize pointers to coordinate objects */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();

    /* Now we can use our objects using our methods and passing the object as parameter */
    c1->setx(c1, 1);
    c1->sety(c1, 2);
```

```
    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* After using our objects we destroy them using our "destructor" function */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

Lea Estructuras en línea: https://riptutorial.com/es/c/topic/1119/estructuras

Capítulo 32: Generación de números aleatorios

Observaciones

Debido a los defectos de `rand()` , muchas otras implementaciones por defecto han surgido a lo largo de los años. Entre esos están:

- `arc4random()` (disponible en OS X y BSD)
- `random()` (disponible en Linux)
- `drand48()` (disponible en POSIX)

Examples

Generación de números aleatorios básicos

La función `rand()` se puede usar para generar un valor entero pseudoaleatorio entre `0` y `RAND_MAX` (`0` y `RAND_MAX` incluidos).

`srand(int)` se utiliza para inicializar el generador de números pseudoaleatorios. Cada vez que `rand()` se siembra con la misma semilla, debe producir la misma secuencia de valores. Solo se debe sembrar una vez antes de llamar a `rand()` . No se debe sembrar repetidamente ni volver a sembrar cada vez que desee generar un nuevo lote de números pseudoaleatorios.

La práctica estándar es usar el resultado del `time(NULL)` como semilla. Si su generador de números aleatorios requiere tener una secuencia determinista, puede inicializar el generador con el mismo valor en cada inicio de programa. Esto generalmente no es necesario para el código de la versión, pero es útil en las ejecuciones de depuración para hacer que los errores sean reproducibles.

Se recomienda sembrar siempre el generador, si no se siembra, se comporta como si se hubiera sembrado con `srand(1)` .

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("Random value between [0, %d]: %d\n", RAND_MAX, i);
    return 0;
}
```

Salida posible:

```
Random value between [0, 2147483647]: 823321433
```

Notas:

El estándar C no garantiza la calidad de la secuencia aleatoria producida. En el pasado, algunas implementaciones de `rand()` tenían serios problemas en la distribución y aleatoriedad de los números generados. **El uso de `rand()` no se recomienda para necesidades serias de generación de números aleatorios, como la criptografía.**

Generador Congruente Permutado

Aquí hay un generador de números aleatorios independiente que no se basa en `rand()` o funciones de biblioteca similares.

¿Por qué querrías tal cosa? Tal vez no confíe en el generador de números aleatorios integrado de su plataforma, o tal vez quiera una fuente reproducible de aleatoriedad independiente de cualquier implementación de biblioteca en particular.

Este código es PCG32 de pcg-random.org , un RNG moderno, rápido y de propósito general con excelentes propiedades estadísticas. No es criptográficamente seguro, así que no lo use para criptografía.

```
#include <stdint.h>

/* *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org
 * Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) */

typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* Advance internal state */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* Calculate output function (XSH RR), uses old state for max ILP */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}
```

Y aquí está cómo llamarlo:

```
#include <stdio.h>
int main(void) {
    pcg32_random_t rng; /* RNG state */
    int i;
```

```
    /* Seed the RNG */
    pcg32_srandom_r(&rng, 42u, 54u);

    /* Print some random 32-bit integers */
    for (i = 0; i < 6; i++)
        printf("0x%08x\n", pcg32_random_r(&rng));

    return 0;
}
```

Restringir la generación a un rango dado

Por lo general, cuando se generan números aleatorios, es útil generar números enteros dentro de un rango, o un valor de p entre 0.0 y 1.0. Si bien la operación de módulo se puede usar para reducir la semilla a un entero bajo, este utiliza los bits bajos, que a menudo pasan por un ciclo corto, lo que resulta en una ligera desviación de la distribución si N es grande en proporción a RAND_MAX.

La macro

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

produce un valor de p de 0.0 a 1.0 - épsilon, por lo que

```
i = (int)(uniform() * N)
```

establecerá `i` en un número aleatorio uniforme dentro del rango de 0 a N - 1.

Desafortunadamente, hay una falla técnica, ya que se permite que RAND_MAX sea más grande de lo que una variable de tipo `double` puede representar con precisión. Esto significa que `RAND_MAX + 1.0` evalúa como RAND_MAX y la función ocasionalmente devuelve la unidad. Sin embargo, esto es poco probable.

Generación Xorshift

Una alternativa buena y fácil a los procedimientos defectuosos de `rand()` es *xorshift* , una clase de generadores de números pseudoaleatorios descubiertos por George Marsaglia . El generador xorshift se encuentra entre los generadores de números aleatorios no criptográficos seguros más rápidos. Más información y otras implementaciones de ejemplo están disponibles en la página de Wikipedia de xorshift.

Ejemplo de implementación

```
#include <stdint.h>

/* These state variables must be initialised so that they are not all zero. */
uint32_t w, x, y, z;

uint32_t xorshift128(void)
{
    uint32_t t = x;
```

```
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
    w ^= w >> 19U;
    w ^= t;
    return w;
}
```

Lea Generación de números aleatorios en línea: https://riptutorial.com/es/c/topic/365/generacion-de-numeros-aleatorios

Capítulo 33: Gestión de la memoria

Introducción

Para administrar la memoria asignada dinámicamente, la biblioteca C estándar proporciona las funciones `malloc()` , `calloc()` , `realloc()` y `free()` . En C99 y posteriores, también hay `aligned_alloc()` . Algunos sistemas también proporcionan `alloca()` .

Sintaxis

- void * align_alloc (alineación size_t, size_t size); / * Solo desde C11 * /
- void * calloc (size_t nelements, size_t size);
- vacío libre (void * ptr);
- void * malloc (size_t size);
- void * realloc (void * ptr, size_t size);
- void * alloca (size_t size); / * de alloca.h, no estándar, no portátil, peligroso. * /

Parámetros

nombre	descripción
tamaño (`malloc` , `realloc` y `aligned_alloc`)	Tamaño total de la memoria en bytes. Para `aligned_alloc` el tamaño debe ser un múltiplo integral de alineación.
tamaño (`calloc`)	tamaño de cada elemento
nelementos	número de elementos
ptr	puntero a la memoria asignada previamente devuelto por `malloc` , `calloc` , `realloc` o `aligned_alloc`
alineación	alineación de la memoria asignada

Observaciones

C11

Tenga en cuenta que `aligned_alloc()` solo se define para C11 o posterior.

Los sistemas como los basados en POSIX proporcionan otras formas de asignar memoria alineada (por ejemplo, `posix_memalign()`), y también tienen otras opciones de administración de memoria (por ejemplo, `mmap()`).

Examples

Liberando memoria

Es posible liberar memoria asignada dinámicamente llamando a free () .

```
int *p = malloc(10 * sizeof *p); /* allocation of memory */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* release of memory */
/* note that after free(p), even using the *value* of the pointer p
   has undefined behavior, until a new value is stored into it. */

/* reusing/re-purposing the pointer itself */
int i = 42;
p = &i; /* This is valid, has defined behaviour */
```

La memoria apuntada por `p` se reclama (ya sea por la implementación libc o por el sistema operativo subyacente) después de la llamada a `free()` , por lo que acceder al bloque de memoria liberado a través de `p` conducirá a un comportamiento indefinido . Los punteros que hacen referencia a elementos de memoria que se han liberado se denominan comúnmente punteros colgantes y presentan un riesgo de seguridad. Además, el estándar C indica que incluso el acceso al valor de un puntero colgante tiene un comportamiento indefinido. Tenga en cuenta que el puntero `p` se puede volver a usar como se muestra arriba.

Tenga en cuenta que solo puede llamar a `free()` en punteros que hayan sido devueltos directamente desde las funciones `malloc()` , `calloc()` , `realloc()` y `aligned_alloc()` , o donde la documentación le indique que la memoria se ha asignado de esa manera (funciones Como `strdup ()` son ejemplos notables). Liberando un puntero que es,

- obtenido usando el operador `&` en una variable, o
- en medio de un bloque asignado,

está prohibido. Por lo general, su compilador no diagnosticará dicho error, pero llevará la ejecución del programa a un estado indefinido.

Hay dos estrategias comunes para prevenir estos casos de comportamiento indefinido.

Lo primero y preferible es simple: dejar que el propio `p` deje de existir cuando ya no se necesita, por ejemplo:

```
if (something_is_needed())
{

    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
```

```
        return -1;
    }

    /* do whatever is needed with p */

    free(p);
}
```

Al llamar a `free()` directamente antes del final del bloque contenedor (es decir, el `}`), `p` sí mismo deja de existir. El compilador dará un error de compilación en cualquier intento de usar `p` después de eso.

Un segundo enfoque es invalidar también el puntero después de liberar la memoria a la que apunta:

```
free(p);
p = NULL;     // you may also use 0 instead of NULL
```

Argumentos a favor de este enfoque:

- En muchas plataformas, un intento de anular la referencia a un puntero nulo causará una falla instantánea: falla de segmentación. Aquí, obtenemos al menos un seguimiento de pila que apunta a la variable que se utilizó después de ser liberado.

  Sin establecer el puntero a `NULL` tenemos puntero colgante. Es muy probable que el programa aún se bloquee, pero más tarde, porque la memoria a la que apunta el puntero se corromperá silenciosamente. Dichos errores son difíciles de rastrear porque pueden resultar en una pila de llamadas que no guarda relación alguna con el problema inicial.

  Este enfoque, por lo tanto, sigue el concepto de falla rápida .

- Es seguro liberar un puntero nulo. El estándar C especifica que `free(NULL)` no tiene ningún efecto:

  > La función libre hace que el espacio al que apunta ptr se desasigne, es decir, que esté disponible para una asignación adicional. Si ptr es un puntero nulo, no se produce ninguna acción. De lo contrario, si el argumento no coincide con un puntero anterior devuelto por la función `calloc` , `malloc` o `realloc` , o si el espacio ha sido desasignado por una llamada a `free` o `realloc` , el comportamiento no está definido.

- A veces no se puede usar el primer enfoque (por ejemplo, la memoria se asigna en una función y se desasigna mucho más tarde en una función completamente diferente)

Asignación de memoria

Asignación estándar

Las funciones de asignación de memoria dinámica C se definen en el encabezado `<stdlib.h>` . Si

uno desea asignar espacio de memoria para un objeto dinámicamente, se puede usar el siguiente código:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

Esto calcula el número de bytes que ocupan diez `int` en la memoria, luego solicita la cantidad de bytes de `malloc` y asigna el resultado (es decir, la dirección de inicio de la porción de memoria que se creó con `malloc`) a un puntero llamado `p` .

Es una buena práctica usar `sizeof` para calcular la cantidad de memoria a solicitar, ya que el resultado de `sizeof` se define por implementación (excepto para los *tipos de caracteres* , que son `char` , `signed char` y `unsigned char` , para los cuales `sizeof` se define para dar siempre `1`).

Dado que es posible que `malloc` no pueda atender la solicitud, puede devolver un puntero nulo. Es importante comprobar esto para evitar intentos posteriores de anular la referencia al puntero nulo.

La memoria asignada dinámicamente con `malloc()` puede redimensionarse con `realloc()` o, cuando ya no se necesita, liberar con `free()` .

Alternativamente, declarando `int array[10];` Asignaría la misma cantidad de memoria. Sin embargo, si se declara dentro de una función sin la palabra clave `static` , solo será utilizable dentro de la función en la que se declara y las funciones que llama (porque la matriz se asignará a la pila y el espacio se liberará para su reutilización cuando la función vuelve). Alternativamente, si se define con `static` dentro de una función, o si se define fuera de cualquier función, entonces su vida útil es la vida útil del programa. Los punteros también pueden devolverse desde una función, sin embargo, una función en C no puede devolver una matriz.

Memoria de cero

La memoria devuelta por `malloc` no puede inicializarse a un valor razonable, y se debe tener cuidado de poner a cero la memoria con `memset` o copiar inmediatamente un valor adecuado en ella. Alternativamente, `calloc` devuelve un bloque del tamaño deseado donde todos los bits se inicializan a `0` . No es necesario que esto sea igual a la representación de cero en coma flotante o una constante de puntero nulo.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

Una nota sobre `calloc` : la mayoría de las implementaciones (comúnmente utilizadas) optimizarán `calloc()` para el rendimiento, por lo que serán más rápidas que llamar a `malloc()` , luego `memset()` , aunque el efecto neto sea

idéntico.

Memoria alineada

C11

C11 introdujo una nueva función `aligned_alloc()` que asigna espacio con la alineación dada. Se puede usar si la memoria que se asignará es necesaria para alinearse en ciertos límites que no se pueden satisfacer con `malloc()` o `calloc()` . `malloc()` funciones `malloc()` y `calloc()` asignan memoria que está adecuadamente alineada para *cualquier* tipo de objeto (es decir, la alineación es `alignof(max_align_t)`). Pero con `aligned_alloc()` se pueden solicitar mayores alineaciones.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

El estándar C11 impone dos restricciones: 1) el *tamaño* (segundo argumento) solicitado debe ser un múltiplo integral de la *alineación* (primer argumento) y 2) el valor de la *alineación* debe ser una alineación válida respaldada por la implementación. El incumplimiento de cualquiera de ellos da como resultado un comportamiento indefinido .

Reasignación de memoria

Es posible que deba expandir o reducir su espacio de almacenamiento de puntero después de haberle asignado memoria. La función `void *realloc(void *ptr, size_t size)` desasigna el objeto antiguo al que apunta `ptr` y devuelve un puntero a un objeto que tiene el tamaño especificado por `size` . `ptr` es el puntero a un bloque de memoria asignado previamente con `malloc` , `calloc` o `realloc` (o un puntero nulo) para ser reasignado. Se conservan los máximos contenidos posibles de la memoria original. Si el nuevo tamaño es más grande, cualquier memoria adicional más allá del tamaño anterior no estará inicializada. Si el nuevo tamaño es más corto, el contenido de la parte contraída se pierde. Si `ptr` es NULL, se asigna un nuevo bloque y la función devuelve un puntero a él.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    p[0] = 42;
    p[9] = 15;
```

```
    /* Reallocate array to a larger size, storing the result into a
     * temporary pointer in case realloc() fails. */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() failed, the original allocation was not free'd yet. */
        if (NULL == temporary)
        {
            perror("realloc() failed");
            free(p); /* Clean up. */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* From here on, array can be used with the new size it was
     * realloc'ed to, until it is free'd. */

    /* The values of p[0] to p[9] are preserved, so this will print:
       42 15
    */
    printf("%d %d\n", p[0], p[9]);

    free(p);

    return EXIT_SUCCESS;
}
```

El objeto reasignado puede o no tener la misma dirección que `*p` . Por lo tanto, es importante capturar el valor de retorno de `realloc` que contiene la nueva dirección si la llamada es exitosa.

Asegúrese de asignar el valor de retorno de `realloc` a un `temporary` lugar de a la `p` original. `realloc` devolverá el valor nulo en caso de que se `realloc` un error, lo que sobrescribiría el puntero. Esto perdería sus datos y crearía una pérdida de memoria.

Arreglos multidimensionales de tamaño variable.

C99

Desde C99, C tiene matrices de longitud variable, VLA, que matrices modelo con límites que solo se conocen en el momento de la inicialización. Si bien debe tener cuidado de no asignar VLA demasiado grande (podrían dañar su pila), usar los *punteros a VLA* y usarlos en el `sizeof` expresiones está bien.

```
double sumAll(size_t n, size_t m, double A[n][m]) {
    double ret = 0.0;
    for (size_t i = 0; i < n; ++i)
       for (size_t j = 0; j < m; ++j)
          ret += A[i][j]
    return ret;
}

int main(int argc, char *argv[argc+1]) {
   size_t n = argc*10;
```

```
    size_t m = argc*8;
    double (*matrix)[m] = malloc(sizeof(double[n][m]));
    // initialize matrix somehow
    double res = sumAll(n, m, matrix);
    printf("result is %g\n", res);
    free(matrix);
}
```

Aquí la `matrix` es un puntero a los elementos de tipo `double[m]` , y la expresión `sizeof` con `double[n][m]` garantiza que contenga espacio para `n` tales elementos.

Todo este espacio se asigna de forma contigua y, por lo tanto, puede ser desasignado por una sola llamada para `free` .

La presencia de VLA en el lenguaje también afecta las posibles declaraciones de matrices y punteros en encabezados de función. Ahora, se permite una expresión entera general dentro de `[]` de los parámetros de la matriz. Para ambas funciones, las expresiones en `[]` utilizan parámetros que se han declarado anteriormente en la lista de parámetros. Para `sumAll` estas son las longitudes que el código de usuario espera para la matriz. Como para todos los parámetros de la función de matriz en C, la dimensión más interna se reescribe en un tipo de puntero, por lo que esto es equivalente a la declaración

```
double sumAll(size_t n, size_t m, double (*A)[m]);
```

Es decir, `n` no es realmente parte de la interfaz de la función, pero la información puede ser útil para la documentación y también puede ser utilizada por compiladores de verificación de límites para advertir sobre el acceso fuera de los límites.

De manera similar, para `main` , la expresión `argc+1` es la longitud mínima que el estándar C prescribe para el argumento `argv` .

Tenga en cuenta que oficialmente el soporte de VLA es opcional en C11, pero no conocemos ningún compilador que implemente C11 y que no los tenga. Puedes probar con la macro `__STDC_NO_VLA__` si es necesario.

realloc (ptr, 0) no es equivalente a free (ptr)

`realloc` es *conceptualmente equivalente* a `malloc + memcpy + free` en el otro puntero.

Si el tamaño del espacio solicitado es cero, el comportamiento de `realloc` está definido por la implementación. Esto es similar para todas las funciones de asignación de memoria que reciben un parámetro de `size` de valor `0` . Dichas funciones pueden, de hecho, devolver un puntero que no sea nulo, pero eso nunca debe ser referenciado.

Por lo tanto, `realloc(ptr,0)` no es equivalente a `free(ptr)` . Puede

- ser una implementación "perezosa" y simplemente devolver `ptr`
- `free(ptr)` , asigne un elemento ficticio y devuelva ese
- `free(ptr)` y devuelve `0`
- Solo devuelve `0` por fallo y no hagas nada más.

Por lo tanto, en particular, los dos últimos casos son indistinguibles por el código de la aplicación.

Esto significa que `realloc(ptr,0)` puede que realmente no libere / desasigne la memoria, por lo que nunca debe usarse como un reemplazo `free` .

Gestión de memoria definida por el usuario

`malloc()` menudo llama a las funciones subyacentes del sistema operativo para obtener páginas de memoria. Pero la función no tiene nada de especial y puede implementarse en C directa declarando una matriz estática grande y asignándola (existe una ligera dificultad para garantizar la alineación correcta, en la práctica, la alineación con 8 bytes es casi siempre adecuada).

Para implementar un esquema simple, un bloque de control se almacena en la región de la memoria inmediatamente antes del puntero que se devolverá de la llamada. Esto significa que se puede implementar `free()` restando del puntero devuelto y leyendo la información de control, que generalmente es el tamaño del bloque más alguna información que le permite volver a la lista libre, una lista vinculada de bloques no asignados.

Cuando el usuario solicita una asignación, se busca la lista gratuita hasta que se encuentra un bloque de tamaño idéntico o mayor a la cantidad solicitada y, si es necesario, se divide. Esto puede llevar a la fragmentación de la memoria si el usuario realiza continuamente muchas asignaciones y libera de tamaño impredecible y en intervalos impredecibles (no todos los programas reales se comportan así, el esquema simple suele ser adecuado para programas pequeños).

```
/* typical control block */
struct block
{
   size_t size;         /* size of block */
   struct block *next;  /* next block in free list */
   struct block *prev;  /* back pointer to previous block in memory */
   void *padding;       /* need 16 bytes to make multiple of 8 */
}

static struct block arena[10000]; /* allocate from here */
static struct block *firstfree;
```

Muchos programas requieren grandes cantidades de asignaciones de objetos pequeños del mismo tamaño. Esto es muy fácil de implementar. Simplemente usa un bloque con el siguiente puntero. Así que si se requiere un bloque de 32 bytes:

```
union block
{
   union block * next;
   unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
```

```
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* last one, null */
    head = &block[0];
}

void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head - block;
}
```

Este esquema es extremadamente rápido y eficiente, y se puede hacer genérico con cierta pérdida de claridad.

alloca: asignar memoria en la pila

Advertencia: la `alloca` solo se menciona aquí en aras de la integridad. Es completamente no portátil (no está cubierto por ninguna de las normas comunes) y tiene una serie de características potencialmente peligrosas que lo hacen inseguro para quienes no lo saben. El código C moderno debe reemplazarlo con *matrices de longitud variable* (VLA).

Página de manual

```
#include <alloca.h>
// glibc version of stdlib.h include alloca.h by default

void foo(int size) {
    char *data = alloca(size);
    /*
      function body;
    */
    // data is automatically freed
}
```

Asigne memoria en el marco de pila de la persona que llama, el espacio al que hace referencia el puntero devuelto se libera automáticamente cuando finaliza la función de la persona que llama.

Si bien esta función es conveniente para la administración automática de la memoria, tenga en cuenta que solicitar una asignación grande puede provocar un desbordamiento de pila y que no puede utilizar `free` memoria `free` asignada con `alloca` (lo que podría causar más problemas con el desbordamiento de pila).

Por esta razón, no se recomienda utilizar `alloca` dentro de un bucle ni una función recursiva.

Y como la memoria está `free` cuando se devuelve la función, no se puede devolver el puntero como resultado de la función (el comportamiento sería indefinido).

Resumen

- llamada idéntica a `malloc`
- Se libera automáticamente al regresar de la función.
- incompatible con `free` funciones `free` , `realloc` (comportamiento indefinido)
- el puntero no se puede devolver como un resultado de función (comportamiento indefinido)
- el tamaño de la asignación está limitado por el espacio de pila, que (en la mayoría de las máquinas) es mucho más pequeño que el espacio de almacenamiento disponible para uso de `malloc()`
- evitar el uso de `alloca()` y VLA (matrices de longitud variable) en una sola función
- `alloca()` no es tan portátil como `malloc()` et al.

Recomendación

- No use `alloca()` en código nuevo

C99

Alternativa moderna.

```
void foo(int size) {
    char data[size];
    /*
      function body;
    */
    // data is automatically freed
}
```

Esto funciona donde `alloca()` funciona, y funciona en lugares donde `alloca()` no funciona (dentro de los bucles, por ejemplo). Supone una implementación C99 o una implementación C11 que no define `__STDC_NO_VLA__` .

Lea Gestión de la memoria en línea: https://riptutorial.com/es/c/topic/4726/gestion-de-la-memoria

Capítulo 34: Hilos (nativos)

Sintaxis

- `#ifndef __STDC_NO_THREADS__`
- `# include <threads.h>`
- `#endif`
- `void call_once(once_flag *flag, void (*func)(void));`
- `int cnd_broadcast(cnd_t *cond);`
- `void cnd_destroy(cnd_t *cond);`
- `int cnd_init(cnd_t *cond);`
- `int cnd_signal(cnd_t *cond);`
- `int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int cnd_wait(cnd_t *cond, mtx_t *mtx);`
- `void mtx_destroy(mtx_t *mtx);`
- `int mtx_init(mtx_t *mtx, int type);`
- `int mtx_lock(mtx_t *mtx);`
- `int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int mtx_trylock(mtx_t *mtx);`
- `int mtx_unlock(mtx_t *mtx);`
- `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`
- `thrd_t thrd_current(void);`
- `int thrd_detach(thrd_t thr);`
- `int thrd_equal(thrd_t thr0, thrd_t thr1);`
- `_Noreturn void thrd_exit(int res);`
- `int thrd_join(thrd_t thr, int *res);`
- `int thrd_sleep(const struct timespec *duration, struct timespec* remaining);`
- `void thrd_yield(void);`
- `int tss_create(tss_t *key, tss_dtor_t dtor);`
- `void tss_delete(tss_t key);`
- `void *tss_get(tss_t key);`
- `int tss_set(tss_t key, void *val);`

Observaciones

Los hilos C11 son una característica opcional. Su ausencia se puede probar con `__STDC__NO_THREAD__` . Actualmente (julio de 2016), esta función aún no está implementada en todas las bibliotecas de C que, de lo contrario, admiten C11.

Las bibliotecas de C que se sabe que admiten subprocesos C11 son:

- musl

Bibliotecas C que no admiten subprocesos C11, sin embargo:

- gnu libc

Examples

Iniciar varios hilos

```
#include <stdio.h>
#include <threads.h>
#include <stdlib.h>

struct my_thread_data {
   double factor;
};

int my_thread_func(void* a) {
   struct my_thread_data* d = a;
   // do something with d
   printf("we found %g\n", d->factor);
   // return an success or error code
   return d->factor > 1.0;
}


int main(int argc, char* argv[argc+1]) {
    unsigned n = 4;
    if (argc > 1) n = strtoull(argv[1], 0, 0);
    // reserve space for the arguments for the threads
    struct my_thread_data D[n];     // can't be initialized
    for (unsigned i = 0; i < n; ++i) {
         D[i] = (struct my_thread_data){ .factor = 0.5*i, };
    }
    // reserve space for the ID's of the threads
    thrd_t id[4];
    // launch the threads
    for (unsigned i = 0; i < n; ++i) {
         thrd_create(&id[i], my_thread_func, &D[i]);
    }
    // Wait that all threads have finished, but throw away their
    // return values
    for (unsigned i = 0; i < n; ++i) {
         thrd_join(id[i], 0);
    }
    return EXIT_SUCCESS;
}
```

Inicialización por un hilo

En la mayoría de los casos, todos los datos a los que acceden varios subprocesos deben inicializarse antes de crear los subprocesos. Esto garantiza que todos los subprocesos comiencen con un estado claro y no se produzca ninguna *condición de carrera* .

Si esto no es posible, `once_flag` `call_once` se pueden usar `once_flag` y `call_once`

```
#include <threads.h>
#include <stdlib.h>
```

```
// the user data for this example
double const* Big = 0;

// the flag to protect big, must be global and/or static
static once_flag onceBig = ONCE_INIT;

void destroyBig(void) {
   free((void*)Big);
}

void initBig(void) {
    // assign to temporary with no const qualification
    double* b = malloc(largeNum);
    if (!b) {
       perror("allocation failed for Big");
       exit(EXIT_FAILURE);
    }
    // now initialize and store Big
    initializeBigWithSophisticatedValues(largeNum, b);
    Big = b;
    // ensure that the space is freed on exit or quick_exit
    atexit(destroyBig);
    at_quick_exit(destroyBig);
}

// the user thread function that relies on Big
int myThreadFunc(void* a) {
   call_once(&onceBig, initBig);
   // only use Big from here on
   ...
   return 0;
}
```

El `once_flag` se utiliza para coordinar los diferentes hilos que quieran para inicializar los mismos datos `Big` . La llamada a `call_once` garantiza que

- `initBig` se llama exactamente una vez
- `call_once` bloquea hasta que dicha llamada a `initBig` se haya realizado, ya sea por el mismo u otro hilo.

Además de la asignación, una cosa típica que se hace en una función llamada antes es una inicialización dinámica de estructuras de datos de control de subprocesos como `mtx_t` o `cnd_t` que no se pueden inicializar de forma estática, usando `mtx_init` o `cnd_init` , respectivamente.

Lea Hilos (nativos) en línea: https://riptutorial.com/es/c/topic/4432/hilos--nativos-

Capítulo 35: Inicialización

Examples

Inicialización de variables en C

En ausencia de una inicialización explícita, se garantiza que las variables externas y `static` se inicialicen a cero; las variables automáticas (incluidas `register` variables de `register`) tienen valores iniciales [1] (es decir, basura) *indeterminados* .

Las variables escalares pueden inicializarse cuando se definen siguiendo el nombre con un signo igual y una expresión:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

Para variables externas y `static` , el inicializador debe ser una *expresión constante* [2] ; La inicialización se realiza una vez, conceptualmente antes de que el programa comience la ejecución.

Para `register` variables automáticas y de `register` , el inicializador no se limita a ser una constante: puede ser cualquier expresión que involucre valores previamente definidos, incluso llamadas a funciones.

Por ejemplo, vea el fragmento de código a continuación

```
int binsearch(int x, int v[], int n)
{
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

en lugar de

```
   int low, high, mid;

   low = 0;
   high = n - 1;
```

En efecto, la inicialización de las variables automáticas son solo una abreviatura de las declaraciones de asignación. Qué forma de preferencia es en gran medida una cuestión de gusto. Generalmente usamos asignaciones explícitas, porque los inicializadores en las declaraciones son más difíciles de ver y están más alejados del punto de uso. Por otro lado, las variables solo deben declararse cuando estén a punto de usarse siempre que sea posible.

Inicializando una matriz:

Una matriz puede inicializarse siguiendo su declaración con una lista de inicializadores entre llaves y separados por comas.

Por ejemplo, para inicializar una matriz de días con el número de días en cada mes:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Tenga en cuenta que enero se codifica como mes cero en esta estructura).

Cuando se omite el tamaño de la matriz, el compilador calculará la longitud contando los inicializadores, de los cuales hay 12 en este caso.

Si hay menos inicializadores para una matriz que el tamaño especificado, los demás serán cero para todos los tipos de variables.

Es un error tener demasiados inicializadores. No hay una manera estándar de especificar la repetición de un inicializador, pero GCC tiene una extensión para hacerlo.

C99

En C89 / C90 o versiones anteriores de C, no había manera de inicializar un elemento en medio de una matriz sin proporcionar también todos los valores anteriores.

C99

Con C99 y superior, los inicializadores designados le permiten inicializar elementos arbitrarios de una matriz, dejando los valores sin inicializar como ceros.

Inicializando matrices de caracteres:

Las matrices de caracteres son un caso especial de inicialización; Se puede utilizar una cadena en lugar de la notación de llaves y comas:

```
char chr_array[] = "hello";
```

Es una abreviatura para el más largo pero equivalente:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

En este caso, el tamaño de la matriz es seis (cinco caracteres más la terminación `'\0'`).

[1] ¿Qué le sucede a una variable declarada, sin inicializar en C? ¿Tiene un valor?

[2] Tenga en cuenta que una *expresión constante* se define como algo que se puede evaluar en tiempo de compilación. Entonces, `int global_var = f();` no es válido. Otro error común es pensar en una variable cualificada `const` como una *expresión constante* . En C, `const` significa "solo lectura", no "constante de tiempo de compilación". Entonces, las definiciones globales como `const`

`int SIZE = 10; int global_arr[SIZE];` y `const int SIZE = 10; int global_var = SIZE;` No son legales en C.

Inicializando estructuras y matrices de estructuras.

Las estructuras y matrices de estructuras pueden inicializarse mediante una serie de valores entre llaves, un valor por miembro de la estructura.

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { 1776, 7, 4 };

struct Date uk_battles[] =
{
    { 1066, 10, 14 },    // Battle of Hastings
    { 1815,  6, 18 },    // Battle of Waterloo
    { 1805, 10, 21 },    // Battle of Trafalgar
};
```

Tenga en cuenta que la inicialización de la matriz se podría escribir sin las llaves interiores, y en tiempos pasados (antes de 1990, por ejemplo) a menudo se habría escrito sin ellas:

```
struct Date uk_battles[] =
{
    1066, 10, 14,    // Battle of Hastings
    1815,  6, 18,    // Battle of Waterloo
    1805, 10, 21,    // Battle of Trafalgar
};
```

Aunque esto funciona, no es un buen estilo moderno; no debe intentar usar esta notación en un código nuevo y debe corregir las advertencias del compilador que generalmente produce.

Ver también los inicializadores designados .

Usando inicializadores designados

C99

C99 introdujo el concepto de *inicializadores designados.* Estos le permiten especificar qué elementos de una matriz, estructura o unión deben inicializarse con los valores siguientes.

Inicializadores designados para elementos de matriz

Para un tipo simple como llano `int` :

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

El término entre corchetes, que puede ser cualquier expresión de entero constante, especifica qué elemento de la matriz debe inicializarse con el valor del término después del signo `=` . Los elementos no especificados se inicializan por defecto, lo que significa que los ceros están definidos. El ejemplo muestra los inicializadores designados en orden; No tienen que estar en orden. El ejemplo muestra las lagunas; esos son legitimos El ejemplo no muestra dos inicializaciones diferentes para el mismo elemento; eso también está permitido (ISO / IEC 9899: 2011, §6.7.9 Inicialización, ¶19 *La inicialización se producirá en el orden de la lista de inicializadores, cada inicializador proporcionado para un subobjeto en particular anulará cualquier inicializador previamente listado para el mismo subobjeto*).

En este ejemplo, el tamaño de la matriz no se define explícitamente, por lo que el índice máximo especificado en los inicializadores designados determina el tamaño de la matriz, que sería 21 elementos en el ejemplo. Si se definió el tamaño, inicializar una entrada más allá del final de la matriz sería un error, como de costumbre.

Inicializadores designados para estructuras.

Puede especificar qué elementos de una estructura se inicializan utilizando el `.` notación del *`element`*

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

Si los elementos no están listados, están inicializados por defecto (a cero).

Inicializador designado para uniones

Puede especificar qué elemento de una unión se inicializa con un inicializador designado.

C89

Antes de la norma C, no había manera de inicializar una `union` . El estándar C89 / C90 le permite inicializar el primer miembro de una `union` , por lo que la elección de qué miembro aparece primero es lo primero.

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    } du;
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
```

```
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 }
};
```

C11

Tenga en cuenta que C11 le permite utilizar miembros anónimos de la unión dentro de una estructura, por lo que no necesita el nombre `du` en el ejemplo anterior:

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

Inicializadores designados para matrices de estructuras, etc.

Estas construcciones se pueden combinar para matrices de estructuras que contienen elementos que son matrices, etc. El uso de conjuntos completos de llaves asegura que la notación sea inequívoca.

```
typedef struct Date Date;  // See earlier in this example

struct date_range
{
    Date    dr_from;
    Date    dr_to;
    char    dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
            .dr_to   = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "Battle of Hastings to Coronation of William the Conqueror",
          },
    [2] = { .dr_from = { .month = 7, .day =  4, .year = 1776 },
            .dr_to   = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "US Declaration of Independence to Constitutional Convention",
          }
 };
```

Especificando rangos en inicializadores de matriz

GCC proporciona una extensión que le permite especificar un rango de elementos en una matriz que debe tener el mismo inicializador:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

Los puntos triples deben estar separados de los números para que uno de los puntos no se interprete como parte de un número de punto flotante (regla de *munch máxima*).

Lea Inicialización en línea: https://riptutorial.com/es/c/topic/4547/inicializacion

Capítulo 36: Instrumentos de cuerda

Introducción

En C, una cadena no es un tipo intrínseco. Una cadena en C es la convención de tener una matriz de caracteres unidimensional que termina con un carácter nulo, con un `'\0'` .

Esto significa que una cadena C con un contenido de `"abc"` tendrá cuatro caracteres `'a'` , `'b'` , `'c'` y `'\0'` .

Vea el ejemplo básico de introducción a cadenas .

Sintaxis

- char str1 [] = "¡Hola mundo!"; / * Modificable * /
- char str2 [14] = "¡Hola mundo!"; / * Modificable * /
- char * str3 = "¡Hola mundo!"; /* No modificable*/

Examples

Calcular la longitud: strlen ()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        puts("Argument missing.");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("The length of the second argument is %zu.\n", len);

    return EXIT_SUCCESS;
}
```

Este programa calcula la longitud de su segundo argumento de entrada y almacena el resultado en `len` . Luego imprime esa longitud al terminal. Por ejemplo, cuando se ejecuta con los parámetros `program_name "Hello, world!"` , el programa emitirá `The length of the second argument is 13.` porque la cadena `Hello, world!` Tiene 13 caracteres de longitud.

`strlen` cuenta todos los **bytes** desde el principio de la cadena hasta, pero sin incluir, el carácter NUL de terminación, `'\0'` . Como tal, solo se puede utilizar cuando se *garantiza* que la cadena

terminará en NUL.

También tenga en cuenta que si la cadena contiene caracteres Unicode, `strlen` no le dirá cuántos caracteres hay en la cadena (ya que algunos caracteres pueden tener varios bytes). En tales casos, debe contar usted mismo los caracteres (*es decir* , las unidades de código). Considere la salida del siguiente ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* "Hello World!" in Greek */

    printf("asciiString has %zu bytes in the array\n", sizeof(asciiString));
    printf("utf8String has %zu bytes in the array\n", sizeof(utf8String));
    printf("\"%s\" is %zu bytes\n", asciiString, strlen(asciiString));
    printf("\"%s\" is %zu bytes\n", utf8String, strlen(utf8String));
}
```

Salida:

```
asciiString has 50 bytes in the array
utf8String has 50 bytes in the array
"Hello world!" is 12 bytes
"Γειά σου Κόσμε!" is 27 bytes
```

Copia y concatenación: strcpy (), strcat ()

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  /* Always ensure that your string is large enough to contain the characters
   * and a terminating NUL character ('\0')!
   */
  char mystring[10];

  /* Copy "foo" into `mystring`, until a NUL character is encountered. */
  strcpy(mystring, "foo");
  printf("%s\n", mystring);

  /* At this point, we used 4 chars of `mystring`, the 3 characters of "foo",
   * and the NUL terminating byte.
   */

  /* Append "bar" to `mystring`. */
  strcat(mystring, "bar");
  printf("%s\n", mystring);

  /* We now use 7 characters of `mystring`: "foo" requires 3, "bar" requires 3
   * and there is a terminating NUL character ('\0') at the end.
   */
```

```
  /* Copy "bar" into `mystring`, overwriting the former contents. */
  strcpy(mystring, "bar");
  printf("%s\n", mystring);

  return 0;
}
```

Salidas:

```
foo
foobar
bar
```

Si agrega o copia desde una cadena existente, asegúrese de que esté terminada en NUL.

Los literales de cadena (por ejemplo, `"foo"`) siempre terminarán en NUL mediante el compilador.

Comparsion: strcmp (), strncmp (), strcasecmp (), strncasecmp ()

Las `strcase*` strcase no son estándar C, sino una extensión POSIX.

La función `strcmp` compara lexicográficamente dos matrices de caracteres terminadas en nulo. Las funciones devuelven un valor negativo si el primer argumento aparece antes del segundo en orden lexicográfico, cero si se comparan igual o positivo si el primer argumento aparece después del segundo en orden lexicográfico.

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "BBB");
    compare("BBB", "CCCCC");
    compare("BBB", "AAAAAA");
    return 0;
}
```

Salidas:

```
BBB equals BBB
BBB comes before CCCCC
```

```
BBB comes after AAAAAA
```

Como función `strcmp` , la función `strcasecmp` también compara lexicográficamente sus argumentos después de traducir cada carácter a su correspondiente en minúscula:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // compute case-insensitive comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "bBB");
    compare("BBB", "ccCCC");
    compare("BBB", "aaaaaa");
    return 0;
}
```

Salidas:

```
BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa
```

`strncmp` y `strncasecmp` comparan como máximo n caracteres:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncmp(lhs, rhs, n); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "Bb", 1);
    compare("BBB", "Bb", 2);
    compare("BBB", "Bb", 3);
    return 0;
```

```
}
```

Salidas:

```
BBB equals Bb
BBB comes before Bb
BBB comes before Bb
```

Tokenización: strtok (), strtok_r () y strtok_s ()

La función `strtok` rompe una cadena en cadenas o tokens más pequeños, utilizando un conjunto de delimitadores.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]\n", ++toknum, token);
        token = strtok(NULL, delimiters);
    }
    /* source is now "Hello\0, world\0\0" */
}
```

Salida:

```
1: [Hello]
2: [world]
```

La cadena de delimitadores puede contener uno o más delimitadores y se pueden usar diferentes cadenas delimitadoras con cada llamada a `strtok` .

Las llamadas a `strtok` para continuar tokenizando la misma cadena fuente no deben pasar la cadena fuente nuevamente, sino que pasan `NULL` como el primer argumento. Si *se* pasa la misma cadena de origen, el primer token se volverá a tokenizar. Es decir, dados los mismos delimitadores, `strtok` simplemente devolvería el primer token nuevamente.

Tenga en cuenta que como `strtok` no asigna nueva memoria para los tokens, *modifica la cadena de origen* . Es decir, en el ejemplo anterior, la cadena `src` se manipulará para producir los tokens a los que hace referencia el puntero devuelto por las llamadas a `strtok` . Esto significa que la cadena de origen no puede ser `const` (por lo que no puede ser un literal de cadena). También significa que la identidad del byte delimitador se pierde (es decir, en el ejemplo, "," y "!" Se eliminan de la cadena fuente de manera efectiva y no se puede saber qué carácter delimitador coincide).

Tenga en cuenta también que varios delimitadores consecutivos en la cadena de origen se tratan como uno solo; en el ejemplo, la segunda coma se ignora.

`strtok` no es seguro para subprocesos ni re-entrante porque usa un búfer estático mientras analiza. Esto significa que si una función llama a `strtok` , ninguna función a la que llama mientras está usando `strtok` también puede usar `strtok` , y no puede ser llamada por ninguna función que esté usando `strtok` .

Un ejemplo que demuestra los problemas causados por el hecho de que `strtok` no es re-entrante es el siguiente:

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* Nested calls to strtok do not work as desired */
    printf("[%s]\n", first);
    part = strtok(first, ".");
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok(NULL, ".");
    }
} while ((first = strtok(NULL, ",")) != NULL);
```

Salida:

```
[1.2]
 [1]
 [2]
```

La operación esperada es que el bucle `do while` externo debe crear tres tokens que constan de cada cadena de número decimal (`"1.2"` , `"3.5"` , `"4.2"`), para cada uno de los cuales `strtok` solicita el bucle interno debe dividirlo en partes separadas. cadenas de dígitos (`"1"` , `"2"` , `"3"` , `"5"` , `"4"` , `"2"`).

Sin embargo, debido a que `strtok` no es reingresante, esto no ocurre. En su lugar, el primer `strtok` crea correctamente el token "1.2 \ 0", y el bucle interno crea correctamente los tokens `"1"` y `"2"` . Pero entonces el `strtok` en el bucle externo está al final de la cadena utilizada por el bucle interno, y devuelve NULL inmediatamente. La segunda y tercera subcadenas de la matriz `src` no se analizan en absoluto.

C11

Las bibliotecas estándar de C no contienen una versión segura para subprocesos o de reingreso, pero otras sí, como el `strtok_r` POSIX. Tenga en cuenta que en el MSVC `strtok` equivalente, `strtok_s` es seguro para subprocesos.

C11

C11 tiene una parte opcional, el Anexo K, que ofrece una versión segura para subprocesos y re-entrant llamada `strtok_s` . Puede probar la función con `__STDC_LIB_EXT1__` . Esta parte opcional no es ampliamente soportada.

La función `strtok_s` difiere de la función `strtok_r` POSIX al protegerse contra el almacenamiento fuera de la cadena que está siendo tokenizada, y al verificar las restricciones de tiempo de ejecución. Sin embargo, en programas escritos correctamente, los `strtok_s` y `strtok_r` comportan igual.

El uso de `strtok_s` con el ejemplo ahora produce la respuesta correcta, así:

```
/* you have to announce that you want to use Annex K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif

char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]\n", first);
    part = strtok_s(first, ".", &posn);
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok_s(NULL, ".", &posn);
    }
}
while ((first = strtok_s(NULL, ",", &next)) != NULL);
```

Y la salida será:

```
[1.2]
 [1]
 [2]
[3.5]
 [3]
 [5]
[4.2]
 [4]
 [2]
```

Encuentre la primera / última aparición de un carácter específico: strchr (), strrchr ()

Las funciones `strchr` y `strrchr` encuentran un carácter en una cadena, que está en una matriz de caracteres terminada en NUL. `strchr` devuelve un puntero a la primera aparición y `strrchr` a la

última.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        printf("Argument missing.\n");
        return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("First position of %c in %s is %td.\n",
              toSearchFor, argv[1], firstOcc-argv[1]); /* A pointer difference's result
                                       is a signed integer and uses the length modifier 't'. */
        }
        else
        {
            printf("%c is not in %s.\n", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("Last position of %c in %s is %td.\n",
              toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }

    return EXIT_SUCCESS;
}
```

Salidas (después de haber generado un ejecutable llamado `pos`):

```
$ ./pos AAAAAAA
First position of A in AAAAAAA is 0.
Last position of A in AAAAAAA is 6.
$ ./pos BAbbbbbAccccAAAAzzz
First position of A in BAbbbbbAccccAAAAzzz is 1.
Last position of A in BAbbbbbAccccAAAAzzz is 15.
$  ./pos qwerty
A is not in qwerty.
```

Un uso común para `strrchr` es extraer un nombre de archivo de una ruta. Por ejemplo, para extraer `myfile.txt` de `C:\Users\eak\myfile.txt` :

```
char *getFileName(const char *path)
```

```
{
    char *pend;

    if ((pend = strrchr(path, '\')) != NULL)
        return pend + 1;

    return NULL;
}
```

Iterando sobre los personajes en una cadena

Si conocemos la longitud de la cadena, podemos usar un bucle for para iterar sobre sus caracteres:

```
char * string = "hello world"; /* This 11 chars long, excluding the 0-terminator. */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Alternativamente, podemos usar la función estándar `strlen()` para obtener la longitud de una cadena si no sabemos qué es la cadena:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Finalmente, podemos aprovechar el hecho de que se garantiza que las cadenas en C terminen en nulo (lo que ya hicimos al pasarlo a `strlen()` en el ejemplo anterior ;-)). Podemos iterar sobre la matriz sin importar su tamaño y dejar de iterar una vez que alcancemos un carácter nulo:

```
size_t i = 0;
while (string[i] != '\0') {       /* Stop looping when we reach the null-character. */
    printf("%c\n", string[i]);    /* Print each character of the string. */
    i++;
}
```

Introducción básica a las cuerdas.

En C, una **cadena** es una secuencia de caracteres que termina con un carácter nulo ('\ 0').

Podemos crear cadenas utilizando **literales de cadena** , que son secuencias de caracteres rodeadas por comillas dobles; por ejemplo, toma la cadena literal `"hello world"` . Los literales de cadena se terminan automáticamente en nulo.

Podemos crear cadenas utilizando varios métodos. Por ejemplo, podemos declarar un `char *` e inicializarlo para que apunte al primer carácter de una cadena:

```
char * string = "hello world";
```

Cuando se inicializa un `char *` a una constante de cadena como la anterior, la cadena misma suele asignarse en datos de solo lectura; `string` es un puntero al primer elemento de la matriz, que es el carácter `'h'` .

Dado que la cadena literal se asigna en la memoria de solo lectura, no es modificable [1] . Cualquier intento de modificarlo conducirá a un comportamiento indefinido , por lo que es mejor agregar `const` para obtener un error de compilación como este

```
char const * string = "hello world";
```

Tiene el mismo efecto [2] como

```
char const string_arr[] = "hello world";
```

Para crear una cadena modificable, puede declarar una matriz de caracteres e inicializar su contenido utilizando un literal de cadena, de esta manera:

```
char modifiable_string[] = "hello world";
```

Esto es equivalente a lo siguiente:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Dado que la segunda versión utiliza un inicializador con llave, la cadena no termina automáticamente en nulo a menos que se incluya explícitamente un carácter `'\0'` en la matriz de caracteres, generalmente como su último elemento.

---

1 No modificable implica que los caracteres en el literal de cadena no pueden modificarse, pero recuerde que la `string` puntero puede modificarse (puede apuntar a otra parte o puede incrementarse o disminuirse).

2 Ambas cadenas tienen un efecto similar en el sentido de que los caracteres de ambas cadenas no pueden modificarse. Debe tenerse en cuenta que la `string` es un puntero a `char` y es un valor l modificable, por lo que puede incrementarse o apuntar a otra ubicación, mientras que el array `string_arr` es un valor l no modificable, no se puede modificar.

Creando matrices de cuerdas

Una serie de cadenas puede significar un par de cosas:

1. Una matriz cuyos elementos son caracteres `char *`
2. Una matriz cuyos elementos son matrices de `char` s

Podemos crear una serie de punteros de caracteres así:

```
char * string_array[] = {
    "foo",
    "bar",
    "baz"
};
```

Recuerde: cuando asignamos literales de cadena a `char *` , las cadenas se asignan en la memoria de solo lectura. Sin embargo, la matriz `string_array` se asigna en la memoria de lectura / escritura. Esto significa que podemos modificar los punteros en la matriz, pero no podemos modificar las cadenas a las que apuntan.

En C, el parámetro a main `argv` (la matriz de argumentos de línea de comando que se pasó cuando se ejecutó el programa) es una matriz de `char *` : `char * argv[]` .

También podemos crear matrices de matrices de caracteres. Dado que las cadenas son matrices de caracteres, una matriz de cadenas es simplemente una matriz cuyos elementos son matrices de caracteres:

```
char modifiable_string_array_literals[][4] = {
    "foo",
    "bar",
    "baz"
};
```

Esto es equivalente a:

```
char modifiable_string_array[][4] = {
    {'f', 'o', 'o', '\0'},
    {'b', 'a', 'r', '\0'},
    {'b', 'a', 'z', '\0'}
};
```

Tenga en cuenta que especificamos `4` como el tamaño de la segunda dimensión de la matriz; Cada una de las cadenas en nuestra matriz es en realidad 4 bytes, ya que debemos incluir el carácter de terminación nula.

strstr

```
/* finds the next instance of needle in haystack
   zbpos: the zero-based position to begin searching from
   haystack: the string to search in
   needle: the string that must be found
   returns the next match of `needle` in `haystack`, or -1 if not found
*/
int findnext(int zbpos, const char *haystack, const char *needle)
{
    char *p;

    if (((p = strstr(haystack + zbpos, needle)) != NULL)
        return p - haystack;

    return -1;
}
```

`strstr` busca en el `haystack` (primero) el argumento de la cadena apuntada por la `needle` . Si se encuentra, `strstr` devuelve la dirección de la ocurrencia. Si no pudo encontrar la `needle` , devuelve NULL. Usamos `zbpos` para no seguir encontrando la misma aguja una y otra vez. Para omitir la primera instancia, agregamos un desplazamiento de `zbpos` . Un clon del Bloc de notas podría

llamar a `findnext` como este, para implementar su diálogo "Buscar siguiente":

```
/*
    Called when the user clicks "Find Next"
    doc: The text of the document to search
    findwhat: The string to find
*/
void onfindnext(const char *doc, const char *findwhat)
{
    static int i;

    if ((i = findnext(i, doc, findwhat)) != -1)
        /* select the text starting from i and ending at i + strlen(findwhat) */
    else
        /* display a message box saying "end of search" */
}
```

Literales de cuerda

Los literales de cadena representan terminados en cero, estática duración matrices de `char` . Debido a que tienen una duración de almacenamiento estático, una cadena literal o un puntero a la misma matriz subyacente se puede usar de manera segura de varias maneras que un puntero a una matriz automática no puede. Por ejemplo, devolver un literal de cadena desde una función tiene un comportamiento bien definido:

```
const char *get_hello() {
    return "Hello, World!";  /* safe */
}
```

Por razones históricas, los elementos de la matriz que corresponden a un literal de cadena no son formalmente `const` . Sin embargo, cualquier intento de modificarlos tiene un comportamiento indefinido . Normalmente, un programa que intenta modificar la matriz correspondiente a un literal de cadena se bloqueará o funcionará mal.

```
char *foo = "hello";
foo[0] = 'y';  /* Undefined behavior - BAD! */
```

Cuando un puntero apunta a una cadena literal, o donde a veces puede hacerlo, es aconsejable declarar la `const` referente de ese puntero para evitar la participación accidental de tal comportamiento indefinido.

```
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

Por otro lado, un puntero hacia o dentro de la matriz subyacente de un literal de cadena no es inherentemente especial; su valor se puede modificar libremente para señalar otra cosa:

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

Además, aunque inicializadores para `char` matrices pueden tener la misma forma que los literales

de cadena, el uso de un inicializador tales no confiere las características de una cadena literal en la matriz inicializada. El inicializador simplemente designa la longitud y los contenidos iniciales de la matriz. En particular, los elementos son modificables si no se declaran explícitamente `const` :

```
char foo[] = "hello";
foo[0] = 'y';  /* OK! */
```

Poniendo a cero una cuerda

Puede llamar a `memset` para `memset` a cero una cadena (o cualquier otro bloque de memoria).

Donde `str` es la cadena a cero, y `n` es el número de bytes en la cadena.

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


int main(void)
{
  char str[42] = "fortytwo";
  size_t n = sizeof str; /* Take the size not the length. */

  printf("'%s'\n", str);

  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Huellas dactilares:

```
'fortytwo'
''
```

Otro ejemplo:

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


#define FORTY_STR "forty"
#define TWO_STR "two"

int main(void)
{
  char str[42] = FORTY_STR TWO_STR;
  size_t n = sizeof str; /* Take the size not the length. */
  char * point_to_two = strstr(str, TWO_STR);

  printf("'%s'\n", str);
```

```
  memset(point_to_two, '\0', n);

  printf("'%s'\n", str);

  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Huellas dactilares:

```
'fortytwo'
'forty'
''
```

strspn y strcspn

Dada una cadena, `strspn` calcula la longitud de la subcadena inicial (intervalo) que consiste únicamente en una lista específica de caracteres. `strcspn` es similar, excepto que calcula la longitud de la subcadena inicial que consta de cualquier carácter excepto los listados:

```
/*
  Provided a string of "tokens" delimited by "separators", print the tokens along
  with the token separators that get skipped.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.,";
    char *s;
    int n;

    for (s = foo; *s != 0; /*empty*/) {
        /* Get the number of token separator characters. */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("skipping separators: << %.*s >> (length=%d)\n", n, s, n);

        /* Actually skip the separators now. */
        s += n;

        /* Get the number of token (non-separator) characters. */
        n = (int)strcspn(s, sepchars);

        if (n > 0)
            printf("token found: << %.*s >> (length=%d)\n", n, s, n);

        /* Skip the token now. */
        s += n;
    }
```

```
    printf("== token list exhausted ==\n");

    return 0;
}
```

Las funciones análogas que utilizan cadenas de caracteres `wcsspn` son `wcsspn` y `wcscspn` ; Se usan de la misma manera.

Copiando cuerdas

Las asignaciones de punteros no copian cadenas

Puede usar el operador `=` para copiar enteros, pero no puede usar el operador `=` para copiar cadenas en C. Las cadenas en C se representan como matrices de caracteres con un carácter nulo de terminación, por lo que usar el operador `=` solo guardará la dirección (puntero) de una cadena.

```
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* Integer is copied */
    a = 20; /* Modifying a leaves b unchanged - b is a 'deep copy' of a */
    printf("%d %d\n", a, b); /* "20 10" will be printed */

    d = c;
    /* Only copies the address of the string -
    there is still only one string stored in memory */

    c[1] = 'x';
    /* Modifies the original string - d[1] = 'x' will do exactly the same thing */

    printf("%s %s\n", c, d); /* "axc axc" will be printed */

    return 0;
}
```

El ejemplo anterior se compiló porque usamos `char *d` lugar de `char d[3]` . El uso de este último causaría un error del compilador. No se puede asignar a matrices en C.

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    b = a; /* compile error */
    printf("%s\n", b);
```

```
    return 0;
}
```

Copiando cadenas utilizando funciones estándar

`strcpy()`

Para copiar cadenas, la función `strcpy()` está disponible en `string.h` . Se debe asignar suficiente espacio para el destino antes de copiar.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    strcpy(b, a); /* think "b special equals a" */
    printf("%s\n", b); /* "abc" will be printed */

    return 0;
}
```

C99

`snprintf()`

Para evitar la saturación del búfer, se puede usar `snprintf()` . No es la mejor solución en cuanto a rendimiento, ya que tiene que analizar la cadena de la plantilla, pero es la única función segura de límite de búfer para copiar cadenas fácilmente disponibles en la biblioteca estándar, que se puede usar sin ningún paso adicional.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* causes buffer overrun (undefined behavior), so do not execute this here!
*/
#endif

    snprintf(b, sizeof(b), "%s", a); /* does not cause buffer overrun */
    printf("%s\n", b); /* "0123456" will be printed */

    return 0;
}
```

`strncat()`

Una segunda opción, con un mejor rendimiento, es utilizar `strncat()` (una versión de comprobación de desbordamiento de búfer de `strcat()`): requiere un tercer argumento que le indica el número máximo de bytes que se debe copiar:

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
    /* copies up to the first (sizeof(dest) - 1) elements of source into dest,
    then puts a \0 on the end of dest */
```

Tenga en cuenta que esta formulación utiliza `sizeof(dest) - 1` ; esto es crucial porque `strncat()` siempre agrega un byte nulo (bueno), pero no cuenta eso en el tamaño de la cadena (causa de confusión y sobrescritura del búfer).

También tenga en cuenta que la alternativa, concatenar después de una cadena no vacía, es aún más complicada. Considerar:

```
char dst[24] = "Clownfish: ";
char src[] = "Marvin and Nemo";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

La salida es:

```
23: [Clownfish: Marvin and N]
```

Tenga en cuenta, sin embargo, que el tamaño especificado como la longitud *no* era el tamaño de la matriz de destino, sino la cantidad de espacio que quedaba en ella, sin contar el byte nulo del terminal. Esto puede causar grandes problemas de sobrescritura. También es un poco derrochador; para especificar correctamente el argumento de longitud, conoce la longitud de los datos en el destino, por lo que podría especificar la dirección del byte nulo al final del contenido existente, evitando que `strncat()` vuelva a escanearlo:

```
    strcpy(dst, "Clownfish: ");
    assert(len < sizeof(dst) - 1);
    strncat(dst + len, src, sizeof(dst) - len - 1);
    printf("%zu: [%s]\n", strlen(dst), dst);
```

Esto produce la misma salida que antes, pero `strncat()` no tiene que escanear el contenido existente de `dst` antes de que comience a copiar.

`strncpy()`

La última opción es la función `strncpy()` . Aunque podría pensar que debería ser lo primero, es una función bastante engañosa que tiene dos objetivos principales:

1. Si la copia a través de `strncpy()` alcanza el límite del búfer, no se escribirá un carácter nulo

de terminación.
2. `strncpy()` siempre llena completamente el destino, con bytes nulos si es necesario.

(Tal implementación peculiar es histórica y fue inicialmente diseñada para manejar nombres de archivos UNIX)

La única forma correcta de usarlo es asegurar manualmente la terminación nula:

```
strncpy(b, a, sizeof(b)); /* the third parameter is destination buffer size */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* terminate the string */
printf("%s\n", b); /* "0123456" will be printed */
```

Incluso entonces, si tienes un búfer grande, se vuelve muy ineficiente usar `strncpy()` debido al relleno nulo adicional.

Convierta cadenas a números: atoi (), atof () (peligroso, no las use)

Advertencia: Las funciones `atoi` , `atol` , `atoll` y `atof` son intrínsecamente inseguras, porque: *si el valor del resultado no puede representarse, el comportamiento no está definido.* (7.20.1p1)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int val;
    if (argc < 2)
    {
        printf("Usage: %s <integer>\n", argv[0]);
        return 0;
    }

    val = atoi(argv[1]);

    printf("String value = %s, Int value = %d\n", argv[1], val);

    return 0;
}
```

Cuando la cadena a convertir es un entero decimal válido que está en el rango, la función funciona:

```
$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200
```

Para cadenas que comienzan con un número, seguidas de otra cosa, solo se analiza el número inicial:

```
$ ./atoi 0x200
0
$ ./atoi 0123x300
```

```
123
```

En todos los demás casos, el comportamiento es indefinido:

```
$ ./atoi hello
Formatting the hard disk...
```

Debido a las ambigüedades anteriores y este comportamiento indefinido, la familia de funciones `atoi` nunca debe usarse.

- Para convertir a `long int` , use `strtol()` lugar de `atol()` .
- Para convertir al `double` , use `strtod()` lugar de `atof()` .

C99

- Para convertir a `long long int` , use `strtoll()` lugar de `atoll()` .

cadena de datos con formato de lectura / escritura

Escribe datos formateados en una cadena

```
int sprintf ( char * str, const char * format, ... );
```

use la función `sprintf` para escribir datos flotantes en la cadena.

```
#include <stdio.h>
int main ()
{
  char buffer [50];
  double PI = 3.1415926;
  sprintf (buffer, "PI = %.7f", PI);
  printf ("%s\n",buffer);
  return 0;
}
```

Leer datos formateados de cadena

```
int sscanf ( const char * s, const char * format, ...);
```

Utilice la función `sscanf` para analizar datos formateados.

```
#include <stdio.h>
int main ()
{
  char sentence []="date : 06-06-2012";
  char str [50];
  int year;
  int month;
  int day;
  sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);
  printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);
  return 0;
```

```
}
```

Convertir con seguridad cadenas a números: funciones strtoX

C99

Desde C99, la biblioteca C tiene un conjunto de funciones de conversión seguras que interpretan una cadena como un número. Sus nombres son de la forma `strtoX` , donde `X` es uno de `l` , `ul` , `d` , etc. para determinar el tipo de destino de la conversión.

```
double strtod(char const* p, char** endptr);
long double strtold(char const* p, char** endptr);
```

Proporcionan la verificación de que una conversión tuvo un desbordamiento o desbordamiento:

```
double ret = strtod(argv[1], 0); /* attempt conversion */

/* check the conversion result. */
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return;  /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Si la cadena de hecho no contiene ningún número, este uso de `strtod` devuelve `0.0` .

Si esto no es satisfactorio, se puede usar el parámetro adicional `endptr` . Es un puntero a puntero que se señalará al final del número detectado en la cadena. Si se establece en `0` , como anteriormente, o `NULL` , simplemente se ignora.

Este parámetro `endptr` proporciona indica si ha habido una conversión exitosa y, de ser así, dónde terminó el número:

```
char *check = 0;
double ret = strtod(argv[1], &check); /* attempt conversion */

/* check the conversion result. */
if (argv[1] == check)
    return; /* No number was detected in string */
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return; /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Hay funciones análogas para convertir a los tipos enteros más anchos:

```
long strtol(char const* p, char** endptr, int nbase);
long long strtoll(char const* p, char** endptr, int nbase);
unsigned long strtoul(char const* p, char** endptr, int nbase);
```

```
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

Estas funciones tienen un tercer parámetro `nbase` que contiene la base numérica en la que se escribe el número.

```
long a = strtol("101",   0, 2 ); /* a = 5L */
long b = strtol("101",   0, 8 ); /* b = 65L */
long c = strtol("101",   0, 10); /* c = 101L */
long d = strtol("101",   0, 16); /* d = 257L */
long e = strtol("101",   0, 0 ); /* e = 101L */
long f = strtol("0101",  0, 0 ); /* f = 65L */
long g = strtol("0x101", 0, 0 ); /* g = 257L */
```

El valor especial `0` para `nbase` significa que la cadena se interpreta de la misma manera que los literales numéricos se interpretan en un programa C: un prefijo de `0x` corresponde a una representación hexadecimal, de lo contrario, un `0` inicial es octal y todos los demás números se ven como decimales.

Así, la forma más práctica de interpretar un argumento de línea de comando como un número sería

```
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* No number given. */

    /* use strtoull because size_t may be wide */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* then check conversion results. */

     ...

    return EXIT_SUCCESS;
}
```

Esto significa que se puede llamar al programa con un parámetro en octal, decimal o hexadecimal.

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/c/topic/1990/instrumentos-de-cuerda

Capítulo 37: Iteraciones / bucles de repetición: for, while, do-while

Sintaxis

- /* Todas las versiones */
- para ([expresión]; [expresión]; [expresión]) una declaración
- para ([expresión]; [expresión]; [expresión]) {cero o varias declaraciones}
- while (expresión) one_statement
- while (expresión) {cero o varias declaraciones}
- do one_statement while (expresión);
- hacer {una o más declaraciones} while (expresión);
- // desde C99 además del formulario anterior
- para (declaración; [expresión]; [expresión]) one_statement;
- para (declaración; [expresión]; [expresión]) {cero o varias declaraciones}

Observaciones

Iteración Declaración / Loops se dividen en dos categorías:

- instrucción de iteración controlada / bucles
- iteración controlada por el pie / bucles

Declaración de iteración controlada por la cabeza / Bucles

```
for ([<expression>]; [<expression>]; [<expression>]) <statement>
while (<expression>) <statement>
```

C99

```
for ([declaration expression]; [expression] [; [expression]]) statement
```

Declaración de iteración controlada por el pie / bucles

```
do <statement> while (<expression>);
```

Examples

En bucle

Para ejecutar un bloque de código a lo largo de una vez más, los bucles entran en la imagen. El bucle `for` se utiliza cuando un bloque de código se ejecuta un número fijo de veces. Por ejemplo,

para llenar una matriz de tamaño `n` con las entradas del usuario, necesitamos ejecutar `scanf()` por `n` veces.

C99

```
#include <stddef.h>          // for size_t

int array[10];               // array of 10 int

for (size_t i = 0; i < 10; i++)    // i starts at 0 and finishes with 9
{
   scanf("%d", &array[i]);
}
```

De esta manera, la llamada a la función `scanf()` se ejecuta `n` veces (10 veces en nuestro ejemplo), pero se escribe solo una vez.

Aquí, la variable `i` es el índice de bucle, y se declara mejor tal como se presenta. El tipo `size_t` (*tamaño*) debe usarse para todo lo que cuenta o recorre los objetos de datos.

Esta forma de declarar variables dentro de `for` solo está disponible para compiladores que se han actualizado al estándar C99. Si, por algún motivo, aún está atascado con un compilador anterior, puede declarar el índice de bucle antes del bucle `for` :

C99

```
#include <stddef.h>        /* for size_t */
size_t i;
int array[10];             /* array of 10 int */

for (i = 0; i < 10; i++)       /* i starts at 0 and finishes at 9 */
{
   scanf("%d", &array[i]);
}
```

Mientras bucle

A `while` bucle se utiliza para ejecutar una parte de código, mientras que una condición es verdadera. El `while` de bucle se va a utilizar cuando un bloque de código se va a ejecutar un número variable de veces. Por ejemplo, el código que se muestra obtiene la entrada del usuario, siempre que el usuario inserte números que no sean `0` . Si el usuario inserta `0` , la condición while ya no es verdadera, por lo que la ejecución saldrá del bucle y continuará con cualquier código posterior:

```
int num = 1;

while (num != 0)
{
    scanf("%d", &num);
}
```

Bucle Do-While

A diferencia de `for` y `while` bucles, `do-while` bucles comprobar la verdad de la condición al final del bucle, lo que significa la `do` bloque se ejecutará una vez, y a continuación, comprobar la condición de la `while` en la parte inferior del bloque. Lo que significa que un bucle `do-while` while *siempre se* ejecutará al menos una vez.

Por ejemplo, este bucle `do-while` while obtendrá números del usuario, hasta que la suma de estos valores sea mayor o igual a `50` :

```
int num, sum;
num = sum = 0;

do
{
  scanf("%d", &num);
  sum += num;

} while (sum < 50);
```

`do-while` bucles `do-while` while son relativamente raros en la mayoría de los estilos de programación.

Estructura y flujo de control en un bucle for.

```
for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}
```

En un bucle `for` , la condición de bucle tiene tres expresiones, todas opcionales.

- La primera expresión, `declaration-or-expression` , *inicializa* el bucle. Se ejecuta exactamente una vez al principio del bucle.

C99

Puede ser una declaración e inicialización de una variable de bucle o una expresión general. Si es una declaración, el alcance de la variable declarada está restringido por la declaración `for` .

C99

Las versiones históricas de C solo permitían una expresión, aquí, y la declaración de una variable de bucle tenía que colocarse antes de la `for` .

- La segunda expresión, `expression2` , es la *condición de prueba* . Primero se ejecuta después de la inicialización. Si la condición es `true` , entonces el control entra en el cuerpo del bucle. Si no, se desplaza al exterior del cuerpo del bucle al final del bucle. Posteriormente, esta condición se verifica después de cada ejecución del cuerpo, así como la instrucción de actualización. Cuando es `true` , el control vuelve al principio del cuerpo del bucle. La condición generalmente pretende ser una verificación del número de veces que se ejecuta el cuerpo del bucle. Esta es la forma principal de salir de un bucle, la otra forma es usar sentencias de salto .

- La tercera expresión, `expression3` , es la *instrucción de actualización* . Se ejecuta después de cada ejecución del cuerpo del bucle. A menudo se usa para incrementar el número de veces que se ha ejecutado el bucle, y esta variable se denomina *iterador* .

Cada instancia de ejecución del cuerpo del bucle se llama una *iteración* .

Ejemplo:

C99

```
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}
```

La salida es:

```
0123456789
```

En el ejemplo anterior, primero se ejecuta `i = 0` , inicializando `i` . Luego, se comprueba la condición `i < 10` , que se evalúa como `true` . El control entra en el cuerpo del bucle y se imprime el valor de `i` . Luego, el control cambia a `i++` , actualizando el valor de `i` de 0 a 1. Luego, la condición se verifica nuevamente y el proceso continúa. Esto continúa hasta que el valor de `i` convierte en 10. Luego, la condición `i < 10` evalúa como `false` , después de lo cual el control sale del bucle.

Bucles infinitos

Se dice que un *bucle* es un *bucle infinito* si el control ingresa pero nunca abandona el cuerpo del bucle. Esto sucede cuando la condición de prueba del bucle nunca se evalúa como `false` .

Ejemplo:

C99

```
for (int i = 0; i >= 0; )
{
    /* body of the loop where i is not changed*/
}
```

En el ejemplo anterior, la variable `i` , el iterador, se inicializa a 0. La condición de prueba es inicialmente `true` . Sin embargo, `i` no se modifica en cualquier parte del cuerpo y la expresión de actualización está vacía. Por lo tanto, `i` seguiré siendo 0, y la condición de prueba nunca será evaluada como `false` , lo que lleva a un bucle infinito.

Suponiendo que no hay declaraciones de salto, otra forma en que se podría formar un bucle infinito es manteniendo explícitamente la condición verdadera:

```
while (true)
{
```

```
    /* body of the loop */
}
```

En un bucle `for` , la declaración de condición opcional. En este caso, la condición siempre se `true` vacía, lo que lleva a un bucle infinito.

```
for (;;)
{
    /* body of the loop */
}
```

Sin embargo, en ciertos casos, la condición podría mantenerse `true` intencionalmente, con la intención de salir del bucle utilizando una instrucción de salto como `break` .

```
while (true)
{
    /* statements */
    if (condition)
    {
         /* more statements */
         break;
    }
}
```

Loop desenrollado y dispositivo de Duff

A veces, el bucle directo no puede estar completamente contenido dentro del cuerpo del bucle. Esto se debe a que el bucle debe estar cebado por algunas instrucciones ***B.*** Luego, la iteración comienza con algunas declaraciones ***A*** , que luego son seguidas por ***B*** nuevamente antes de hacer un bucle.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

Para evitar posibles problemas de cortar / pegar con la repetición de ***B*** dos veces en el código, dispositivo de Duff podría ser aplicada para iniciar el bucle desde el medio del `while` del cuerpo, utilizando una sentencia switch y caer a través del comportamiento.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
default:    do_B(); /* FALL THROUGH */
}
```

El dispositivo de Duff fue realmente inventado para implementar el desenrollado de bucles. Imagínese aplicando una máscara a un bloque de memoria, donde `n` es un tipo integral con signo con un valor positivo.

```
do {
```

```
    *ptr++ ^= mask;
} while (--n > 0);
```

Si `n` siempre fuera divisible por 4, podría desenrollar esto fácilmente como:

```
do {
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
} while ((n -= 4) > 0);
```

Pero, con el Dispositivo de Duff, el código puede seguir este lenguaje desenrollado que salta al lugar correcto en el medio del bucle si `n` no es divisible por 4.

```
switch (n % 4) do {
case 0: *ptr++ ^= mask; /* FALL THROUGH */
case 3: *ptr++ ^= mask; /* FALL THROUGH */
case 2: *ptr++ ^= mask; /* FALL THROUGH */
case 1: *ptr++ ^= mask; /* FALL THROUGH */
} while ((n -= 4) > 0);
```

Este tipo de desenrollado manual rara vez se requiere con compiladores modernos, ya que el motor de optimización del compilador puede desenrollar bucles en nombre del programador.

Lea Iteraciones / bucles de repetición: for, while, do-while en línea:
https://riptutorial.com/es/c/topic/5151/iteraciones---bucles-de-repeticion--for--while--do-while

Capítulo 38: Lenguajes comunes de programación C y prácticas de desarrollador.

Examples

Comparando literal y variable

Supongamos que estás comparando valor con alguna variable

```
if ( i  == 2) //Bad-way
{
    doSomething;
}
```

Ahora supongamos que te has equivocado con `==` con `=` . Entonces tomará su dulce momento para averiguarlo.

```
if( 2 == i) //Good-way
{
    doSomething;
}
```

Luego, si se omite accidentalmente un signo igual, el compilador se quejará de un "intento de asignación a un literal". Esto no lo protegerá cuando compare dos variables, pero cada pequeña ayuda.

Vea aquí para más información.

No deje en blanco la lista de parámetros de una función - use void

Supongamos que está creando una función que no requiere argumentos cuando se le llama y se enfrenta al dilema de cómo debe definir la lista de parámetros en el prototipo de función y la definición de función.

- Tiene la opción de mantener la lista de parámetros vacía tanto para el prototipo como para la definición. Por lo tanto, se ven igual que la declaración de llamada de función que necesitará.
- Lees en alguna parte que uno de los usos de la palabra clave ***void*** (solo hay unos pocos), es definir la lista de parámetros de funciones que no aceptan ningún argumento en su llamada. Por lo tanto, esta es también una opción.

Entonces, ¿cuál es la elección correcta?

RESPUESTA: usando la palabra clave ***void***

CONSEJOS GENERALES: Si un idioma proporciona cierta característica para usar con un propósito especial, es mejor que lo use en su código. Por ejemplo, usar `enum` s en lugar de `#define` macros (eso es para otro ejemplo).

C11 sección 6.7.6.3 "Declaradores de función", párrafo 10, establece:

> El caso especial de un parámetro sin nombre de tipo void como el único elemento en la lista especifica que la función no tiene parámetros.

El párrafo 14 de esa misma sección muestra la única diferencia:

> ... Una lista vacía en un declarador de función que forma parte de una definición de esa función especifica que la función no tiene parámetros. La lista vacía en un declarador de función que no forma parte de una definición de esa función especifica que no se proporciona información sobre el número o los tipos de parámetros.

Una explicación simplificada provista por K&R (pgs- 72-73) para lo anterior:

> Además, si una declaración de función no incluye argumentos, como en `double atof();` , eso también significa que no se debe asumir nada sobre los argumentos de `atof` ; Todas las comprobaciones de parámetros están desactivadas. Este significado especial de la lista de argumentos vacíos tiene la intención de permitir que los programas C más antiguos se compilen con compiladores nuevos. Pero es una mala idea usarlo con nuevos programas. Si la función toma argumentos, declararlos; Si no toma argumentos, use `void` .

Así es como debería verse su prototipo de función:

```
int foo(void);
```

Y así es como debería ser la definición de la función:

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

Una ventaja de usar el tipo de declaración anterior, sobre el tipo `int foo()` (es decir, sin usar la palabra clave ***void***), es que el compilador puede detectar el error si llama a su función utilizando una declaración errónea como `foo(42)` . Este tipo de declaración de llamada de función no causará ningún error si deja la lista de parámetros en blanco. El error pasaría en silencio, sin ser detectado y el código aún se ejecutaría.

Esto también significa que debe definir la función `main()` esta manera:

```
int main(void)
{
```

```
    ...
    <statements>
    ...
    return 0;
}
```

Tenga en cuenta que aunque una función definida con una lista de parámetros vacía no toma argumentos, no proporciona un prototipo para la función, por lo que el compilador no se quejará si la función se llama posteriormente con argumentos. Por ejemplo:

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called\n", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
    return 0;
}
```

Si ese código se guarda en el archivo `proto79.c` , se puede compilar en Unix con GCC (la versión 7.1.0 en macOS Sierra 10.12.5 se usa para demostración) de esta manera:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o
proto79
$
```

Si compilas con opciones más estrictas, obtienes errores:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c:3:13: error: function declaration isn't a prototype [-Werror=strict-prototypes]
 static void parameterless()
             ^~~~~~~~~~~~~
proto79.c: In function 'parameterless':
proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]
cc1: all warnings being treated as errors
$
```

Si le da a la función el prototipo formal `static void parameterless(void)` , entonces la compilación genera errores:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c: In function 'main':
proto79.c:10:5: error: too many arguments to function 'parameterless'
     parameterless(3, "arguments", "provided");
     ^~~~~~~~~~~~~
proto79.c:3:13: note: declared here
 static void parameterless(void)
             ^~~~~~~~~~~~~
$
```

Moraleja: siempre asegúrese de tener prototipos y asegúrese de que su compilador le diga cuándo no está obedeciendo las reglas.

Lea Lenguajes comunes de programación C y prácticas de desarrollador. en línea: https://riptutorial.com/es/c/topic/10543/lenguajes-comunes-de-programacion-c-y-practicas-de-desarrollador-

Capítulo 39: Listas enlazadas

Observaciones

El lenguaje C no define una estructura de datos de lista enlazada. Si está utilizando C y necesita una lista enlazada, debe usar una lista enlazada de una biblioteca existente (como GLib) o escribir su propia interfaz de lista enlazada. Este tema muestra ejemplos de listas vinculadas y listas dobles vinculadas que pueden utilizarse como punto de partida para escribir sus propias listas vinculadas.

Lista enlazada individualmente

La lista contiene nodos que se componen de un enlace llamado siguiente.

Estructura de datos

```
struct singly_node
{
  struct singly_node * next;
};
```

Lista doblemente enlazada

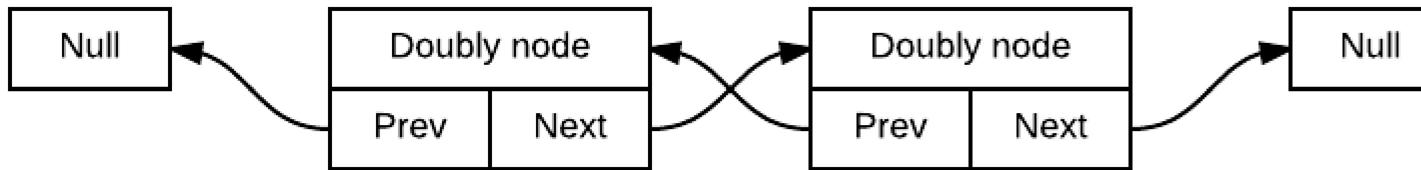
La lista contiene nodos que se componen de dos enlaces llamados anterior y siguiente. Los enlaces normalmente hacen referencia a un nodo con la misma estructura.

Estructura de datos

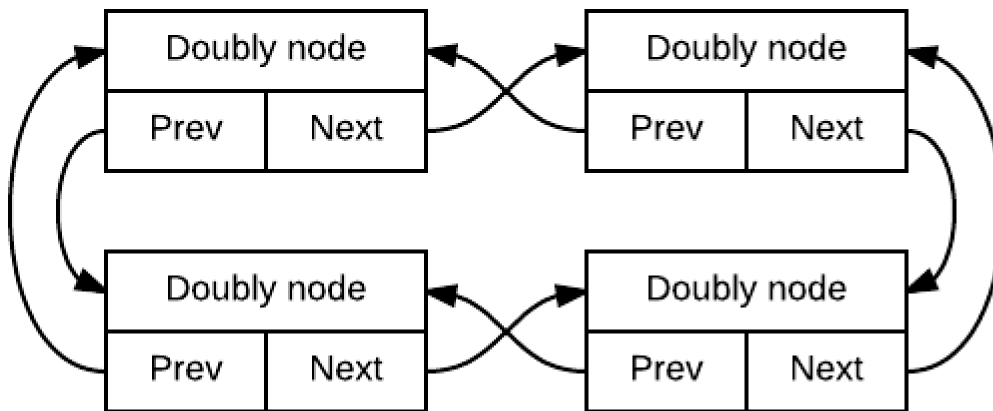
```
struct doubly_node
{
  struct doubly_node * prev;
  struct doubly_node * next;
};
```

Topoliges

Lineal o abierto



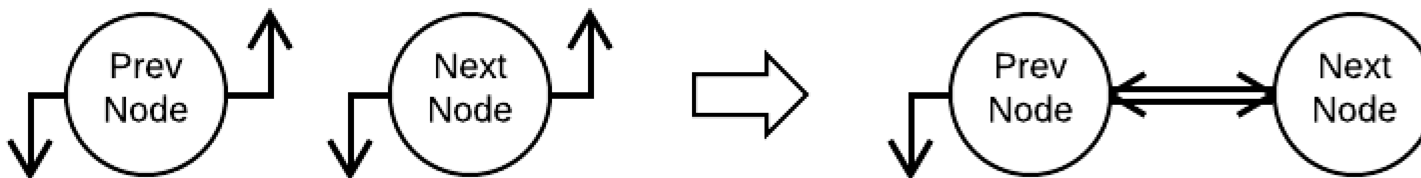
Circular o anillo



Procedimientos

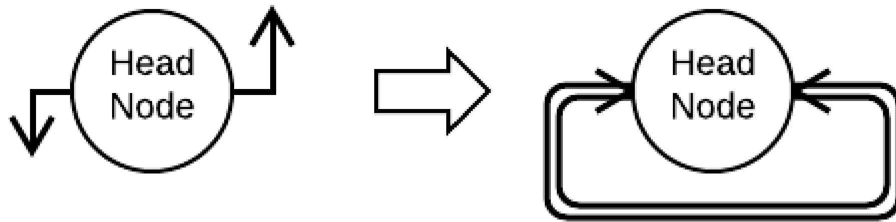
Enlazar

Unir dos nodos juntos.



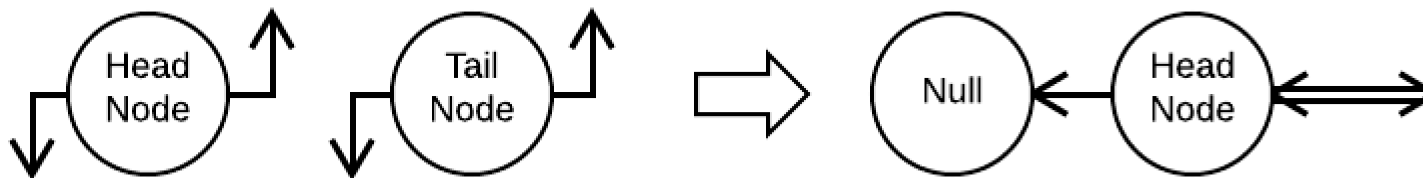

```
void doubly_node_bind (struct doubly_node * prev, struct doubly_node * next)
{
  prev->next = next;
  next->prev = prev;
}
```

Hacer una lista enlazada circularmente




```
void doubly_node_make_empty_circularly_list (struct doubly_node * head)
{
  doubly_node_bind (head, head);
}
```

Hacer una lista enlazada linealmente




```
void doubly_node_make_empty_linear_list (struct doubly_node * head, struct doubly_node * tail)
{
  head->prev = NULL;
  tail->next = NULL;
  doubly_node_bind (head, tail);
}
```

Inserción

Supongamos que una lista vacía siempre contiene un nodo en lugar de NULL. Entonces, los procedimientos de inserción no tienen que tomar NULL en consideración.

```
void doubly_node_insert_between
(struct doubly_node * prev, struct doubly_node * next, struct doubly_node * insertion)
{
  doubly_node_bind (prev, insertion);
  doubly_node_bind (insertion, next);
}

void doubly_node_insert_before
(struct doubly_node * tail, struct doubly_node * insertion)
{
  doubly_node_insert_between (tail->prev, tail, insertion);
}

void doubly_node_insert_after
```

```
(struct doubly_node * head, struct doubly_node * insertion)
{
  doubly_node_insert_between (head, head->next, insertion);
}
```

Examples

Insertar un nodo al comienzo de una lista enlazada individualmente

El código a continuación solicitará números y continuará agregándolos al comienzo de una lista vinculada.

```
/* This program will demonstrate inserting a node at the beginning of a linked list */

#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};


void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
  struct Node* headNode;
  headNode = NULL; /* Initialize our first node pointer to be NULL. */
  size_t listSize, i;
  do {
    printf("How many numbers would you like to input?\n");
  } while(1 != scanf("%zu", &listSize));

  for (i = 0; i < listSize; i++) {
    int numToAdd;
    do {
      printf("Enter a number:\n");
    } while (1 != scanf("%d", &numToAdd));

    insert_node (&headNode, numToAdd);
    printf("Current list after your inserted node: \n");
    print_list(headNode);
  }

  return 0;
}

void print_list (struct Node *head) {
  struct node* currentNode = head;

  /* Iterate through each link. */
  while (currentNode != NULL) {
      printf("Value: %d\n", currentNode->data);
      currentNode = currentNode -> next;
  }
}
```

```
void insert_node (struct Node **head, int nodeValue) {
  struct Node *currentNode = malloc(sizeof *currentNode);
  currentNode->data = nodeValue;
  currentNode->next = (*head);

  *head = currentNode;
}
```

Explicación para la inserción de nodos

Para entender cómo agregamos nodos al principio, echemos un vistazo a los posibles escenarios:

1. La lista está vacía, por lo que necesitamos agregar un nuevo nodo. En cuyo caso, nuestra memoria se ve así donde `HEAD` es un puntero al primer nodo:

```
| HEAD | --> NULL
```

La línea `currentNode->next = *headNode;` asignará el valor de `currentNode->next` a ser `NULL` desde que `headNode` originalmente comienza con un valor de `NULL` .

Ahora, queremos configurar nuestro puntero de nodo principal para que apunte a nuestro nodo actual.

```
  -----      -------------
 |HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */
  -----      -------------
```

Esto se hace con `*headNode = currentNode;`

2. La lista ya está poblada; Necesitamos agregar un nuevo nodo al principio. En aras de la simplicidad, comencemos con 1 nodo:

```
 -----    -----------
 HEAD --> FIRST NODE --> NULL
 -----    -----------
```

Con `currentNode->next = *headNode` , la estructura de datos se ve así:

```
 ---------        -----    ---------------------
 currentNode -->  HEAD --> POINTER TO FIRST NODE --> NULL
 ---------        -----    ---------------------
```

Lo cual, obviamente, necesita ser modificado ya que `*headNode` debería apuntar a `currentNode` .

```
 ----    -----------    ---------------
 HEAD -> currentNode -->     NODE       -> NULL
 ----    -----------    ---------------
```

Esto se hace con `*headNode = currentNode;`

Insertando un nodo en la posición n.

Hasta ahora, hemos observado la inserción de un nodo al principio de una lista enlazada individualmente . Sin embargo, la mayoría de las veces también querrá poder insertar nodos en otros lugares. El código escrito a continuación muestra cómo es posible escribir una función `insert()` para insertar nodos en *cualquier lugar* de las listas vinculadas.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};

struct Node* insert(struct Node* head, int value, size_t position);
void print_list (struct Node* head);

int main(int argc, char *argv[]) {
  struct Node *head = NULL; /* Initialize the list to be empty */

  /* Insert nodes at positions with values: */
  head = insert(head, 1, 0);
  head = insert(head, 100, 1);
  head = insert(head, 21, 2);
  head = insert(head, 2, 3);
  head = insert(head, 5, 4);
  head = insert(head, 42, 2);

  print_list(head);
  return 0;
}

struct Node* insert(struct Node* head, int value, size_t position) {
  size_t i = 0;
  struct Node *currentNode;

  /* Create our node */
  currentNode = malloc(sizeof *currentNode);
  /* Check for success of malloc() here! */

  /* Assign data */
  currentNode->data = value;

  /* Holds a pointer to the 'next' field that we have to link to the new node.
     By initializing it to &head we handle the case of insertion at the beginning. */
  struct Node **nextForPosition = &head;
  /* Iterate to get the 'next' field we are looking for.
     Note: Insert at the end if position is larger than current number of elements. */
  for (i = 0; i < position && *nextForPosition != NULL; i++) {
      /* nextForPosition is pointing to the 'next' field of the node.
         So *nextForPosition is a pointer to the next node.
         Update it with a pointer to the 'next' field of the next node. */
      nextForPosition = &(*nextForPosition)->next;
  }

  /* Here, we are taking the link to the next node (the one our newly inserted node should
  point to) by dereferencing nextForPosition, which points to the 'next' field of the node
  that is in the position we want to insert our node at.
```

```
  We assign this link to our next value. */
  currentNode->next = *nextForPosition;

  /* Now, we want to correct the link of the node before the position of our
  new node: it will be changed to be a pointer to our new node. */
  *nextForPosition = currentNode;

  return head;
}

void print_list (struct Node* head) {
  /* Go through the list of nodes and print out the data in each node */
  struct Node* i = head;
  while (i != NULL) {
    printf("%d\n", i->data);
    i = i->next;
  }
}
```

Invertir una lista enlazada

También puede realizar esta tarea de forma recursiva, pero en este ejemplo he elegido utilizar un enfoque iterativo. Esta tarea es útil si está insertando todos sus nodos al comienzo de una lista vinculada . Aquí hay un ejemplo:

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
  int data;
  struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
void print_list(struct Node *headNode);
void reverse_list(struct Node **headNode);


int main(void) {
  int i;
  struct Node *head = NULL;

  for(i = 1; i <= NUM_ITEMS; i++) {
    insert_node(&head, i, i);
  }
  print_list(head);

  printf("I will now reverse the linked list\n");
  reverse_list(&head);
  print_list(head);
  return 0;
}

void print_list(struct Node *headNode) {
  struct Node *iterator;
```

```
  for(iterator = headNode; iterator != NULL; iterator = iterator->next) {
    printf("Value: %d\n", iterator->data);
  }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
  int i;
  struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *nodeBeforePosition = *headNode;

  currentNode->data = nodeValue;

  if(position == 1) {
    currentNode->next = *headNode;
    *headNode = currentNode;
    return;
  }

  for (i = 0; i < position - 2; i++) {
    nodeBeforePosition = nodeBeforePosition->next;
  }

  currentNode->next = nodeBeforePosition->next;
  nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
  struct Node *iterator = *headNode;
  struct Node *previousNode = NULL;
  struct Node *nextNode = NULL;

  while (iterator != NULL) {
    nextNode = iterator->next;
    iterator->next = previousNode;
    previousNode = iterator;
    iterator = nextNode;
  }

  /* Iterator will be NULL by the end, so the last node will be stored in
  previousNode.  We will set the last node to be the headNode */
  *headNode = previousNode;
}
```

Explicación para el método de lista inversa

Comenzamos el Nodo `previousNode` como `NULL` , ya que sabemos en la primera iteración del bucle, si estamos buscando el nodo antes del primer nodo principal, será `NULL` . El primer nodo se convertirá en el último nodo de la lista, y la siguiente variable, naturalmente, debería ser `NULL` .

Básicamente, el concepto de revertir la lista enlazada aquí es que en realidad revertimos los enlaces en sí mismos. El siguiente miembro de cada nodo se convertirá en el nodo anterior, así:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Donde cada número representa un nodo. Esta lista se convertiría en:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Finalmente, la cabeza debe apuntar al quinto nodo en su lugar, y cada nodo debe apuntar al nodo anterior de este.

El nodo 1 debe apuntar a `NULL` ya que no había nada antes de él. El nodo 2 debe apuntar al nodo 1, el nodo 3 debe apuntar al nodo 2, etcétera.

Sin embargo, hay *un pequeño problema* con este método. Si rompemos el enlace al siguiente nodo y lo cambiamos al nodo anterior, no podremos pasar al siguiente nodo en la lista ya que el enlace a él desapareció.

La solución a este problema es simplemente almacenar el siguiente elemento en una variable (`nextNode`) antes de cambiar el enlace.

Una lista doblemente enlazada.

Un ejemplo de código que muestra cómo se pueden insertar los nodos en una lista con doble enlace, cómo se puede revertir fácilmente la lista y cómo se puede imprimir a la inversa.

```
#include <stdio.h>
#include <stdlib.h>

/* This data is not always stored in a structure, but it is sometimes for ease of use */
struct Node {
  /* Sometimes a key is also stored and used in the functions */
  int data;
  struct Node* next;
  struct Node* previous;
};

void insert_at_beginning(struct Node **pheadNode, int value);
void insert_at_end(struct Node **pheadNode, int value);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);

int main(void) {
  /* Sometimes in a doubly linked list the last node is also stored */
  struct Node *head = NULL;

  printf("Insert a node at the beginning of the list.\n");
  insert_at_beginning(&head, 5);
  print_list(head);

  printf("Insert a node at the beginning, and then print the list backwards\n");
  insert_at_beginning(&head, 10);
  print_list_backwards(head);

  printf("Insert a node at the end, and then print the list forwards.\n");

  insert_at_end(&head, 15);
  print_list(head);
```

```
  free_list(head);

  return 0;
}

void print_list_backwards(struct Node *headNode) {
  if (NULL == headNode)
  {
    return;
  }
  /*
  Iterate through the list, and once we get to the end, iterate backwards to print
  out the items in reverse order (this is done with the pointer to the previous node).
  This can be done even more easily if a pointer to the last node is stored.
  */
  struct Node *i = headNode;
  while (i->next != NULL) {
    i = i->next; /* Move to the end of the list */
  }

  while (i != NULL) {
    printf("Value: %d\n", i->data);
    i = i->previous;
  }
}

void print_list(struct Node *headNode) {
  /* Iterate through the list and print out the data member of each node */
  struct Node *i;
  for (i = headNode; i != NULL; i = i->next) {
    printf("Value: %d\n", i->data);
  }
}

void insert_at_beginning(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }
  /*
  This is done similarly to how we insert a node at the beginning of a singly linked
  list, instead we set the previous member of the structure as well
  */
  currentNode = malloc(sizeof *currentNode);

  currentNode->next = NULL;
  currentNode->previous = NULL;
  currentNode->data = value;

  if (*pheadNode == NULL) { /* The list is empty */
    *pheadNode = currentNode;
    return;
  }

  currentNode->next = *pheadNode;
  (*pheadNode)->previous = currentNode;
  *pheadNode = currentNode;
}
```

```
void insert_at_end(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }

  /*
  This can, again be done easily by being able to have the previous element.  It
  would also be even more useful to have a pointer to the last node, which is commonly
  used.
  */

  currentNode = malloc(sizeof *currentNode);
  struct Node *i = *pheadNode;

  currentNode->data = value;
  currentNode->next = NULL;
  currentNode->previous = NULL;

  if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
  }

  while (i->next != NULL) { /* Go to the end of the list */
    i = i->next;
  }

  i->next = currentNode;
  currentNode->previous = i;
}

void free_list(struct Node *node) {
  while (node != NULL) {
    struct Node *next = node->next;
    free(node);
    node = next;
  }
}
```

Tenga en cuenta que a veces, almacenar un puntero al último nodo es útil (es más eficiente simplemente poder saltar directamente al final de la lista que tener que recorrer hasta el final):

```
struct Node *lastNode = NULL;
```

En cuyo caso, es necesario actualizarlo tras los cambios en la lista.

A veces, una clave también se utiliza para identificar elementos. Es simplemente un miembro de la estructura del Nodo:

```
struct Node {
  int data;
  int key;
  struct Node* next;
  struct Node* previous;
};
```

La clave se utiliza cuando se realizan tareas en un elemento específico, como eliminar elementos.

Lea Listas enlazadas en línea: https://riptutorial.com/es/c/topic/560/listas-enlazadas

Capítulo 40: Literales compuestos

Sintaxis

- (tipo) {lista de inicializadores}

Observaciones

La norma C dice en C11-§6.5.2.5 / 3:

> Una expresión de posfijo que consiste en un nombre de tipo entre paréntesis seguido de una lista de inicializadores de corchetes es un *literal compuesto* . Proporciona un objeto sin nombre cuyo valor viene dado por la lista de inicializadores. [99)]

y la nota de pie de página 99 dice:

> Tenga en cuenta que esto difiere de una expresión de reparto. Por ejemplo, una conversión de caracteres especifica una conversión a tipos escalares o solo **anulados** , y el resultado de una expresión de conversión no es un valor de l.

Tenga en cuenta que:

> Los literales de cadena, y los literales compuestos con tipos const-calificados, no necesitan designar objetos distintos. [101)]
>
> 101) Esto permite que las implementaciones compartan almacenamiento para literales de cadena y literales compuestos constantes con representaciones iguales o superpuestas.

El ejemplo se da en el estándar:
C11-§6.5.2.5 / 13:

> Al igual que los literales de cadena, los literales compuestos con calificación constante se pueden colocar en la memoria de solo lectura e incluso se pueden compartir. Por ejemplo,
>
> ```
> (const char []){"abc"} == "abc"
> ```
>
> podría dar 1 si el almacenamiento de los literales es compartido.

Examples

Definición / Inicialización de Literales Compuestos

Un literal compuesto es un objeto sin nombre que se crea en el ámbito donde se define. El concepto fue introducido por primera vez en el estándar C99. Un ejemplo para literal compuesto es

Ejemplos de la norma C, C11-§6.5.2.5 / 9:

```
int *p = (int [2]){ 2, 4 };
```

`p` se inicializa en la dirección del primer elemento de una matriz sin nombre de dos ints.

El literal compuesto es un lvalor. La duración de almacenamiento del objeto sin nombre es estática (si el literal aparece en el ámbito del archivo) o automática (si el literal aparece en el ámbito del bloque), y en este último caso, la vida útil del objeto finaliza cuando el control abandona el bloque que lo contiene.

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){ *p };
    /*...*/
}
```

> `p` se le asigna la dirección del primer elemento de una matriz de dos ints, el primero tiene el valor apuntado previamente por `p` y el segundo, cero. [...]

Aquí `p` permanece vigente hasta el final del bloque.

Compuesto literal con designadores.

(también de C11)

```
  struct point {
    unsigned x;
    unsigned y;
  };

  extern void drawline(struct point, struct point);

 // used somewhere like this
 drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4});
```

Una línea de `drawline` función ficticia recibe dos argumentos de tipo `struct point` . El primero tiene valores de coordenadas `x == 1` y `y == 1` , mientras que el segundo tiene `x == 3` y `y == 4`

Literal compuesto sin especificar la longitud de la matriz

```
int *p = (int []){ 1, 2, 3};
```

En este caso, el tamaño de la matriz no está especificado, entonces estará determinado por la longitud del inicializador.

Literal compuesto que tiene una longitud de inicializador menor que el tamaño de matriz especificado

```
int *p = (int [10]){1, 2, 3};
```

El resto de los elementos del literal compuesto se inicializarán a `0` implícitamente.

Literal compuesto de solo lectura

Tenga en cuenta que un literal compuesto es un lvalor y, por lo tanto, sus elementos pueden ser modificables. Un literal compuesto de *solo lectura* se puede especificar usando el calificador `const` como `(const int[]){1,2}` .

Literal compuesto que contiene expresiones arbitrarias

Dentro de una función, un literal compuesto, como para cualquier inicialización desde C99, puede tener expresiones arbitrarias.

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

Lea Literales compuestos en línea: https://riptutorial.com/es/c/topic/4135/literales-compuestos

Capítulo 41: Literales para números, caracteres y cadenas.

Observaciones

El término **literal** se usa comúnmente para describir una secuencia de caracteres en un código C que designa un valor constante como un número (por ejemplo, `0`) o una cadena (por ejemplo, `"C"`). Estrictamente hablando, el estándar usa el término **constante** para **constantes** enteras, constantes flotantes, constantes de enumeración y constantes de caracteres, reservando el término 'literal' para literales de cadena, pero esto no es un uso común.

Los literales pueden tener **prefijos** o **sufijos** (pero no ambos) que son caracteres adicionales que pueden iniciar o finalizar un literal para cambiar su tipo predeterminado o su representación.

Examples

Literales enteros

Los literales enteros se utilizan para proporcionar valores integrales. Se admiten tres bases numéricas, indicadas por prefijos:

Base	Prefijo	Ejemplo
Decimal	Ninguna	`5`
Octal	`0`	`0345`
Hexadecimal	`0x` o `0X`	`0x12AB` , `0X12AB` , `0x12ab` , `0x12Ab`

Tenga en cuenta que esta escritura no incluye ningún signo, por lo que los literales enteros son siempre positivos. Algo así como `-1` se trata como una expresión que tiene un entero literal (`1`) que se niega con un `-`

El tipo de un entero decimal decimal es el primer tipo de datos que puede ajustarse al valor de `int` y `long` . Desde C99, `long long` también es compatible con literales muy grandes.

El tipo de literal entero octal o hexadecimal es el primer tipo de datos que puede ajustarse al valor de `int` , `unsigned` , `long` y `unsigned long` . Desde C99, también se admiten largos literales `long long` y `unsigned long long` para literales muy grandes.

Usando varios sufijos, se puede cambiar el tipo predeterminado de un literal.

Sufijo	Explicación
`L` , `l`	`long int`

Sufijo	Explicación
`LL` , `ll` (desde C99)	`long long int`
`U` , `u`	`unsigned`

Los sufijos U y L / LL se pueden combinar en cualquier orden y caso. Es un error duplicar los sufijos (por ejemplo, proporcionar dos sufijos `U`) incluso si tienen casos diferentes.

Literales de cuerda

Los literales de cadena se utilizan para especificar matrices de caracteres. Son secuencias de caracteres encerrados entre comillas dobles (por ejemplo, `"abcd"` y tienen el tipo `char*`).

El prefijo `L` hace que el literal sea una matriz de caracteres anchos, de tipo `wchar_t*` . Por ejemplo, `L"abcd"` .

Desde C11, hay otros prefijos de codificación, similares a `L` :

prefijo	tipo de base	codificación
ninguna	`char`	dependiente de la plataforma
`L`	`wchar_t`	dependiente de la plataforma
`u8`	`char`	UTF-8
`u`	`char16_t`	usualmente UTF-16
`U`	`char32_t`	usualmente UTF-32

Para los dos últimos, se puede consultar con macros de prueba de características si la codificación es efectivamente la codificación UTF correspondiente.

Literales de punto flotante

Los literales de punto flotante se utilizan para representar números reales firmados. Los siguientes sufijos se pueden usar para especificar el tipo de un literal:

Sufijo	Tipo	Ejemplos
ninguna	`double`	`3.1415926` `-3E6`
`f` , `F`	`float`	`3.1415926f` `2.1E-6F`
`l` , `L`	`long double`	`3.1415926L` `1E126L`

Para usar estos sufijos, el literal *debe* ser un literal de punto flotante. Por ejemplo, `3f` es un error,

ya que `3` es un literal entero, mientras que `3.f` o `3.0f` son correctos. Para el `long double` , la recomendación es usar siempre `L` mayúscula para facilitar la lectura.

Literales de personajes

Los literales de caracteres son un tipo especial de literales enteros que se utilizan para representar un carácter. Están entre comillas simples, por ejemplo, `'a'` y tienen el tipo `int` . El valor del literal es un valor entero de acuerdo con el conjunto de caracteres de la máquina. No permiten sufijos.

El prefijo `L` antes de un literal de carácter lo convierte en un carácter ancho del tipo `wchar_t` . Del mismo modo, ya que los prefijos C11 `u` y `U` convierten en caracteres anchos de tipo `char16_t` y `char32_t` , respectivamente.

Cuando se pretende representar ciertos caracteres especiales, como un carácter que no se puede imprimir, se utilizan secuencias de escape. Las secuencias de escape utilizan una secuencia de caracteres que se traducen en otro carácter. Todas las secuencias de escape constan de dos o más caracteres, el primero de los cuales es una barra invertida `\` . Los caracteres que siguen inmediatamente a la barra invertida determinan qué carácter literal se interpreta la secuencia.

Secuencia de escape	Personaje representado
`\b`	Retroceso
`\f`	Alimentación de forma
`\n`	Avance de línea (nueva línea)
`\r`	Retorno de carro
`\t`	Pestaña horizontal
`\v`	Pestaña vertical
`\\`	Barra invertida
`\'`	Comilla simple
`\"`	Comilla doble
`\?`	Signo de interrogación
`\nnn`	Valor octal
`\xnn` ...	Valor hexadecimal

C89

Secuencia de escape	Personaje representado
`\a`	Alerta (pitido, campana)

C99

Secuencia de escape	Personaje representado
`\unnnn`	Nombre del personaje universal
`\Unnnnnnnn`	Nombre del personaje universal

Un nombre de carácter universal es un punto de código Unicode. Un nombre de carácter universal puede asignarse a más de un carácter. Los dígitos `n` se interpretan como dígitos hexadecimales. Dependiendo de la codificación UTF en uso, una secuencia de nombre de carácter universal puede resultar en un punto de código que consta de varios caracteres, en lugar de un solo `char` carácter normal.

Cuando se utiliza la secuencia de escape de alimentación de línea en el modo de texto I / O, se convierte a la secuencia de bytes o bytes de nueva línea específicos del sistema operativo.

La secuencia de escape de signo de interrogación se utiliza para evitar los trigrafos . Por ejemplo, `??/` se compila como el trigraph que representa un carácter de barra invertida `'\'` , pero usar `?\?/` Daría como resultado la *cadena* `"??/"` .

Puede haber uno, dos o tres números octales `n` en la secuencia de escape del valor octal.

Lea Literales para números, caracteres y cadenas. en línea:
https://riptutorial.com/es/c/topic/3455/literales-para-numeros--caracteres-y-cadenas-

Capítulo 42: Los operadores

Introducción

Un operador en un lenguaje de programación es un símbolo que le dice al compilador o intérprete que realice una operación matemática, relacional o lógica específica y produzca un resultado final.

C tiene muchos operadores poderosos. Muchos operadores de C son operadores binarios, lo que significa que tienen dos operandos. Por ejemplo, en `a / b` , `/` es un operador binario que acepta dos operandos (`a` , `b`). ¿Hay algunos operadores unarios que toman un operando (por ejemplo: `~` , `++`) y solo un operador ternario `? :` .

Sintaxis

- operador expr1
- operador expr2
- expr1 operador expr2
- expr1? expr2: expr3

Observaciones

Los operadores tienen una *aridad* , una *precedencia* y una *asociatividad* .

- *Arity* indica el número de operandos. En C, existen tres aridades de operadores diferentes:
  - Unario (1 operando)
  - Binario (2 operandos)
  - Ternario (3 operandos)
- *La precedencia* indica qué operadores "enlazan" primero a sus operandos. Es decir, qué operador tiene prioridad para operar en sus operandos. Por ejemplo, el lenguaje C obedece a la convención de que la multiplicación y la división tienen prioridad sobre la suma y la resta:

  ```
  a * b + c
  ```

  Da el mismo resultado que

  ```
  (a * b) + c
  ```

  Si esto no es lo que se quería, se puede forzar la precedencia utilizando paréntesis, porque tienen la precedencia *más alta* de todos los operadores.

```
a * (b + c)
```

Esta nueva expresión producirá un resultado que difiere de las dos expresiones anteriores.

El lenguaje C tiene muchos niveles de precedencia; A continuación se muestra una tabla de todos los operadores, en orden descendente de prioridad.

Tabla de precedencia

<table>
<tr><th>Los operadores</th><th>Asociatividad</th></tr>
<tr><td>() [] -> .</td><td>de izquierda a derecha</td></tr>
<tr><td>! ~ ++ -- + - * (desreferencia) (type) sizeof</td><td>De derecha a izquierda</td></tr>
<tr><td>* (multiplicación) / %</td><td>de izquierda a derecha</td></tr>
<tr><td>+ -</td><td>de izquierda a derecha</td></tr>
<tr><td><< >></td><td>de izquierda a derecha</td></tr>
<tr><td>< <= > >=</td><td>de izquierda a derecha</td></tr>
<tr><td>== !=</td><td>de izquierda a derecha</td></tr>
<tr><td>&</td><td>de izquierda a derecha</td></tr>
<tr><td>^</td><td>de izquierda a derecha</td></tr>
<tr><td>|</td><td>de izquierda a derecha</td></tr>
<tr><td>&&</td><td>de izquierda a derecha</td></tr>
<tr><td>||</td><td>de izquierda a derecha</td></tr>
<tr><td>?:</td><td>De derecha a izquierda</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>De derecha a izquierda</td></tr>
<tr><td>,</td><td>de izquierda a derecha</td></tr>
</table>

- *La asociatividad* indica cómo se vinculan los operadores de igual precedencia de manera predeterminada, y hay dos tipos: de *izquierda a derecha* y de *derecha a izquierda* . Un ejemplo de enlace de *izquierda a derecha* es el operador de resta (`-`). La expresion

```
a - b - c - d
```

tiene tres restas de precedencia idéntica, pero da el mismo resultado que

```
((a - b) - c) - d
```

porque el más a la izquierda `-` se une primero a sus dos operandos.

Un ejemplo de asociatividad de *derecha a izquierda* son los operadores de referencia `*` y post-incremento `++` . Ambos tienen la misma prioridad, por lo que si se usan en una expresión como

```
* ptr ++
```

, esto es equivalente a

```
* (ptr ++)
```

porque el operador unario más a la derecha (`++`) se une primero a su único operando.

Examples

Operadores relacionales

Los operadores relacionales comprueban si una relación específica entre dos operandos es verdadera. El resultado se evalúa como 1 (lo que significa *verdadero*) o 0 (lo que significa *falso*). Este resultado se usa a menudo para afectar el flujo de control (a través de `if` , `while` , `for`), pero también se puede almacenar en variables.

Es igual a "=="

Comprueba si los operandos suministrados son iguales.

```
1 == 0;         /* evaluates to 0. */
1 == 1;         /* evaluates to 1. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr == yptr;   /* evaluates to 0, the operands hold different location addresses. */
*xptr == *yptr; /* evaluates to 1, the operands point at locations that hold the same value.
*/
```

Atención: ¡Este operador no debe confundirse con el operador de asignación (`=`)!

No es igual a "! ="

Comprueba si los operandos suministrados no son iguales.

```
1 != 0;         /* evaluates to 1. */
1 != 1;         /* evaluates to 0. */
```

```
int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr != yptr;   /* evaluates to 1, the operands hold different location addresses. */
*xptr != *yptr; /* evaluates to 0, the operands point at locations that hold the same value.
*/
```

Este operador devuelve efectivamente el resultado opuesto al del operador igual (`==`).

No "!"

Compruebe si un objeto es igual a `0` .

El `!` También se puede utilizar directamente con una variable de la siguiente manera:

```
!someVal
```

Esto tiene el mismo efecto que:

```
someVal == 0
```

Mayor que ">"

Comprueba si el operando de la izquierda tiene un valor mayor que el operando de la derecha

```
5 > 4      /* evaluates to 1. */
4 > 5      /* evaluates to 0. */
4 > 4      /* evaluates to 0. */
```

Menos que "<"

Comprueba si el operando de la izquierda tiene un valor más pequeño que el operando de la derecha

```
5 < 4      /* evaluates to 0. */
4 < 5      /* evaluates to 1. */
4 < 4      /* evaluates to 0. */
```

Mayor o igual que "> ="

Comprueba si el operando de la mano izquierda tiene un valor mayor o igual que el operando de la derecha.

```
5 >= 4      /* evaluates to 1. */
4 >= 5      /* evaluates to 0. */
4 >= 4      /* evaluates to 1. */
```

Menor o igual que "<="

Comprueba si el operando de la izquierda tiene un valor más pequeño o igual que el operando de la derecha.

```
5 <= 4      /* evaluates to 0. */
4 <= 5      /* evaluates to 1. */
4 <= 4      /* evaluates to 1. */
```

Operadores de Asignación

Asigna el valor del operando de la derecha a la ubicación de almacenamiento nombrada por el operando de la izquierda y devuelve el valor.

```
int x = 5;      /* Variable x holds the value 5. Returns 5. */
char y = 'c';   /* Variable y holds the value 99. Returns 99
                 * (as the character 'c' is represented in the ASCII table with 99).
                 */
float z = 1.5;  /* variable z holds the value 1.5. Returns 1.5. */
char const* s = "foo"; /* Variable s holds the address of the first character of the string
'foo'. */
```

Varias operaciones aritméticas tienen un operador de *asignación compuesto* .

```
a += b  /* equal to: a = a + b */
a -= b  /* equal to: a = a - b */
a *= b  /* equal to: a = a * b */
a /= b  /* equal to: a = a / b */
a %= b  /* equal to: a = a % b */
a &= b  /* equal to: a = a & b */
a |= b  /* equal to: a = a | b */
a ^= b  /* equal to: a = a ^ b */
a <<= b /* equal to: a = a << b */
a >>= b /* equal to: a = a >> b */
```

Una característica importante de estas asignaciones compuestas es que la expresión en el lado izquierdo (`a`) solo se evalúa una vez. Por ejemplo, si `p` es un puntero

```
*p += 27;
```

desreferencias `p` solo una vez, mientras que lo siguiente lo hace dos veces.

```
*p = *p + 27;
```

También se debe tener en cuenta que el resultado de una asignación como `a = b` es lo que se conoce como un valor de *r* . Por lo tanto, la asignación realmente tiene un valor que luego se puede asignar a otra variable. Esto permite el encadenamiento de asignaciones para establecer múltiples variables en una sola declaración.

Este *valor* puede usarse en las expresiones de control de las sentencias `if` (o bucles o sentencias

de `switch`) que guardan algún código en el resultado de otra expresión o llamada de función. Por ejemplo:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Debido a esto, se debe tener cuidado para evitar un error tipográfico común que puede conducir a errores misteriosos.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

Esto tendrá resultados desastrosos, ya que `a = 1` siempre se evaluará como `1` y, por lo tanto, la expresión de control de la sentencia `if` siempre será verdadera (lea más sobre este escollo común aquí). El autor casi seguramente quiso usar el operador de igualdad (`==`) como se muestra a continuación:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

Asociatividad de operadores

```
int a, b = 1, c = 2;
a = b = c;
```

Esto asigna `c` a `b` , que devuelve `b` , que es lo que se asigna a `a` . Esto sucede porque todos los operadores de asignación tienen asociatividad derecha, lo que significa que la operación más a la derecha en la expresión se evalúa primero, y procede de derecha a izquierda.

Operadores aritméticos

Aritmética básica

Devuelva un valor que sea el resultado de aplicar el operando de la mano izquierda al operando de la derecha, utilizando la operación matemática asociada. Se aplican reglas matemáticas normales de conmutación (es decir, la suma y la multiplicación son conmutativas, la resta, la división y el módulo no lo son).

Operador de adición

El operador de suma (`+`) se utiliza para agregar dos operandos juntos. Ejemplo:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a + b; /* c now holds the value 12 */

    printf("%d + %d = %d",a,b,c); /* will output "5 + 7 = 12" */

    return 0;
}
```

Operador de resta

El operador de sustracción (`-`) se usa para restar el segundo operando del primero. Ejemplo:

```
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c now holds the value 3 */

    printf("%d - %d = %d",a,b,c); /* will output "10 - 7 = 3" */

    return 0;
}
```

Operador de multiplicación

El operador de multiplicación (`*`) se utiliza para multiplicar ambos operandos. Ejemplo:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c now holds the value 35 */

    printf("%d * %d = %d",a,b,c); /* will output "5 * 7 = 35" */

    return 0;
}
```

No debe confundirse con el operador de `*` desreferencia.

Operador de la división

El operador de división (`/`) divide el primer operando por el segundo. Si ambos operandos de la división son enteros, devolverá un valor entero y descartará el resto (use el operador de módulo `%` para calcular y adquirir el resto).

Si uno de los operandos es un valor de punto flotante, el resultado es una aproximación de la fracción.

Ejemplo:

```
#include <stdio.h>

int main (void)
{
    int a = 19 / 2 ; /* a holds value 9   */
    int b = 18 / 2 ; /* b holds value 9   */
    int c = 255 / 2; /* c holds value 127 */
    int d = 44 / 4 ; /* d holds value 11  */
    double e = 19 / 2.0 ; /* e holds value 9.5   */
    double f = 18.0 / 2 ; /* f holds value 9.0   */
    double g = 255 / 2.0; /* g holds value 127.5 */
    double h = 45.0 / 4 ; /* h holds value 11.25 */

    printf("19 / 2 = %d\n", a);    /* Will output "19 / 2 = 9"    */
    printf("18 / 2 = %d\n", b);    /* Will output "18 / 2 = 9"    */
    printf("255 / 2 = %d\n", c);   /* Will output "255 / 2 = 127" */
    printf("44 / 4 = %d\n", d);    /* Will output "44 / 4 = 11"   */
    printf("19 / 2.0 = %g\n", e);  /* Will output "19 / 2.0 = 9.5"    */
    printf("18.0 / 2 = %g\n", f);  /* Will output "18.0 / 2 = 9"      */
    printf("255 / 2.0 = %g\n", g); /* Will output "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g\n", h);  /* Will output "45.0 / 4 = 11.25"  */

    return 0;
}
```

Operador Modulo

El operador de módulo (`%`) recibe solo operandos enteros, y se utiliza para calcular el resto después de que el primer operando se divide por el segundo. Ejemplo:

```
#include <stdio.h>

int main (void) {
    int a = 25 % 2;    /* a holds value 1  */
    int b = 24 % 2;    /* b holds value 0  */
    int c = 155 % 5;   /* c holds value 0  */
    int d = 49 % 25;   /* d holds value 24 */

    printf("25 % 2 = %d\n", a);     /* Will output "25 % 2 = 1"    */
    printf("24 % 2 = %d\n", b);     /* Will output "24 % 2 = 0"    */
    printf("155 % 5 = %d\n", c);    /* Will output "155 % 5 = 0"   */
```

```
    printf("49 % 25 = %d\n", d);    /* Will output "49 % 25 = 24"  */

    return 0;
}
```

Operadores de Incremento / Decremento

Los operadores de incremento (`a++`) y decremento (`a--`) son diferentes porque cambian el valor de la variable a la que se aplican sin un operador de asignación. Puede usar los operadores de incremento y decremento antes o después de la variable. La ubicación del operador cambia el tiempo de la incrementación / disminución del valor antes o después de asignarlo a la variable. Ejemplo:

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;
    int c = 1;
    int d = 4;

    a++;
    printf("a = %d\n",a);    /* Will output "a = 2" */
    b--;
    printf("b = %d\n",b);    /* Will output "b = 3" */

    if (++c > 1) { /* c is incremented by 1 before being compared in the condition */
        printf("This will print\n");    /* This is printed */
    } else {
        printf("This will never print\n");    /* This is not printed */
    }

    if (d-- < 4) {  /* d is decremented after being compared */
        printf("This will never print\n");    /* This is not printed */
    } else {
        printf("This will print\n");    /* This is printed */
    }
}
```

Como muestra el ejemplo de `c` y `d` muestra, ambos operadores tienen dos formas, como notación de prefijo y la notación postfix. Ambos tienen el mismo efecto al incrementar (`++`) o disminuir (`--`) la variable, pero difieren según el valor que devuelven: las operaciones de prefijo realizan la operación primero y luego devuelven el valor, mientras que las operaciones de posfijo primero determinan el valor que es ser devuelto, y luego hacer la operación.

Debido a este comportamiento potencialmente contraintuitivo, el uso de operadores de incremento / decremento dentro de expresiones es controvertido.

Operadores logicos

Y lógico

Realiza un AND lógico booleano de los dos operandos que devuelven 1 si ambos operandos son distintos de cero. El operador lógico AND es de tipo `int` .

```
0 && 0  /* Returns 0. */
0 && 1  /* Returns 0. */
2 && 0  /* Returns 0. */
2 && 3  /* Returns 1. */
```

O lógico

Realiza un OR lógico booleano de los dos operandos que devuelven 1 si alguno de los operandos no es cero. El operador lógico OR es de tipo `int` .

```
0 || 0  /* Returns 0. */
0 || 1  /* Returns 1.  */
2 || 0  /* Returns 1.  */
2 || 3  /* Returns 1.  */
```

Lógica NO

Realiza una negación lógica. El operador lógico NOT es de tipo `int` . El operador NO verifica si al menos un bit es igual a 1, si es así devuelve 0. En caso contrario, devuelve 1;

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

Evaluación de corto circuito

Hay algunas propiedades cruciales comunes a `&&` y `||` :

- el operando de la izquierda (LHS) se evalúa completamente antes de que se evalúe el operando de la derecha (RHS),
- hay un punto de secuencia entre la evaluación del operando de la izquierda y el operando de la derecha,
- y, lo más importante, el operando de la derecha no se evalúa en absoluto si el resultado del operando de la izquierda determina el resultado general.

Esto significa que:

- si el LHS se evalúa como 'verdadero' (distinto de cero), el RHS de `||` no se evaluará (porque el resultado de "verdadero O cualquier cosa" es "verdadero"),
- si el LHS se evalúa como "falso" (cero), el RHS de `&&` no se evaluará (porque el resultado de "falso Y cualquier cosa" es "falso").

Esto es importante ya que le permite escribir código como:

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

Si se pasa un valor negativo a la función, el `value >= 0` término se evalúa como falso y el `value < NUM_NAMES` term no se evalúa.

Incremento / Decremento

Los operadores de incremento y decremento existen en forma de *prefijo* y *postfijo* .

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;        /* increments a by one, and returns new value; a == 2, tmp == 2  */
tmp = a++;        /* increments a by one, but returns old value; a == 3, tmp == 2 */
tmp = --b;        /* decrements b by one, and returns new value; b == 0, tmp == 0 */
tmp = b--;        /* decrements b by one, but returns old value; b == -1, tmp == 0 */
```

Tenga en cuenta que las operaciones aritméticas no introducen puntos de secuencia , por lo que ciertas expresiones con `++` o `--` operadores pueden introducir un comportamiento indefinido .

Operador Condicional / Operador Ternario

Evalúa su primer operando y, si el valor resultante no es igual a cero, evalúa su segundo operando. De lo contrario, evalúa su tercer operando, como se muestra en el siguiente ejemplo:

```
a = b ? c : d;
```

es equivalente a:

```
if (b)
    a = c;
else
    a = d;
```

Este pseudo-código lo representa: `condition ? value_if_true : value_if_false` . Cada valor puede ser el resultado de una expresión evaluada.

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

El operador condicional puede estar anidado. Por ejemplo, el siguiente código determina el mayor de tres números:

```
big= a > b ? (a > c ? a : c)
```

```
          : (b > c ? b : c);
```

El siguiente ejemplo escribe enteros pares en un archivo y enteros impares en otro archivo:

```
#include<stdio.h>

int main()
{
    FILE *even, *odds;
    int n = 10;
    size_t k = 0;

    even = fopen("even.txt", "w");
    odds = fopen("odds.txt", "w");

    for(k = 1; k < n + 1; k++)
    {
        k%2==0 ? fprintf(even, "\t%5d\n", k)
               : fprintf(odds, "\t%5d\n", k);
    }
    fclose(even);
    fclose(odds);

    return 0;
}
```

El operador condicional se asocia de derecha a izquierda. Considera lo siguiente:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

Como la asociación es de derecha a izquierda, la expresión anterior se evalúa como

```
exp1 ? exp2 : ( exp3 ? exp4 : exp5 )
```

Operador de coma

Evalúa su operando izquierdo, descarta el valor resultante y luego evalúa sus operandos de derechos y el resultado produce el valor de su operando más a la derecha.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

El operador de coma introduce un punto de secuencia entre sus operandos.

Tenga en cuenta que la *coma* utilizada en las llamadas de funciones que separan argumentos NO es el *operador de coma* , sino que se llama *separador,* que es diferente del *operador de coma* . Por lo tanto, no tiene las propiedades del *operador de coma* .

La llamada anterior `printf()` contiene el *operador de coma* y el *separador* .

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
/*           ^        ^ this is a comma operator */
```

```
/*          this is a separator */
```

El operador de coma se utiliza a menudo en la sección de inicialización, así como en la sección de actualización de un bucle `for` . Por ejemplo:

```
for(k = 1; k < 10; printf("\%d\\n", k), k += 2);   /*outputs the odd numbers below 9/*

/* outputs sum to first 9 natural numbers */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\\n", k, sumk);
```

Operador de Reparto

Realiza una conversión *explícita* en el tipo dado del valor resultante de evaluar la expresión dada.

```
int x = 3;
int y = 4;
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

Aquí, el valor de `x` se convierte en un `double` , la división también promueve el valor de `y` al `double` , y el resultado de la división, se pasa un `double` a `printf` para imprimir.

operador de tamaño

Con un tipo como operando.

Se evalúa en el tamaño en bytes, de tipo `size_t` , de objetos del tipo dado. Requiere paréntesis alrededor del tipo.

```
printf("%zu\n", sizeof(int)); /* Valid, outputs the size of an int object, which is platform-dependent. */
printf("%zu\n", sizeof int); /* Invalid, types as arguments need to be surrounded by parentheses! */
```

Con una expresión como operando.

Se evalúa en el tamaño en bytes, de tipo `size_t` , de objetos del tipo de la expresión dada. La expresión en sí no es evaluada. No se requieren paréntesis; sin embargo, debido a que la expresión dada debe ser única, se considera la mejor práctica usarlos siempre.

```
char ch = 'a';
printf("%zu\n", sizeof(ch)); /* Valid, will output the size of a char object, which is always 1 for all platforms. */
printf("%zu\n", sizeof ch);  /* Valid, will output the size of a char object, which is always 1 for all platforms. */
```

Aritmética de puntero

Además de puntero

Dado un puntero y un tipo escalar `N` , se evalúa como un puntero al elemento `N` th del tipo apuntado a que se corresponde directamente con el objeto apuntado en la memoria.

```
int arr[] = {1, 2, 3, 4, 5};
printf("*(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

No importa si el puntero se usa como el valor de operando o el valor escalar. Esto significa que cosas como `3 + arr` son válidas. Si `arr[k]` es el miembro `k+1` de una matriz, `arr+k` es un puntero a `arr[k]` . En otras palabras, `arr` o `arr+0` es un puntero a `arr[0]` , `arr+1` es un puntero a `arr[2]` , y así sucesivamente. En general, `*(arr+k)` es igual que `arr[k]` .

A diferencia de la aritmética habitual, la adición de `1` a un puntero a un `int` agregará `4` bytes al valor de la dirección actual. Como los nombres de matriz son punteros constantes, `+` es el único operador que podemos usar para acceder a los miembros de una matriz mediante notación de puntero utilizando el nombre de la matriz. Sin embargo, al definir un puntero a una matriz, podemos obtener más flexibilidad para procesar los datos en una matriz. Por ejemplo, podemos imprimir los miembros de una matriz de la siguiente manera:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", *(arr + k));
    }
    return 0;
}
```

Al definir un puntero a la matriz, el programa anterior es equivalente a lo siguiente:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* or int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", ptr[k]);
        /* or   printf("\n\t%d", *(ptr + k)); */
        /* or   printf("\n\t%d", *ptr++); */
    }
    return 0;
}
```

Ver que se acceda a los miembros de la matriz `arr` utilizando los operadores `+` y `++` . Los otros operadores que se pueden usar con el puntero `ptr` son `-` y `--` .

Resta de puntero

Dados dos punteros al mismo tipo, se evalúa en un objeto de tipo `ptrdiff_t` que contiene el valor escalar que debe agregarse al segundo puntero para obtener el valor del primer puntero.

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti\n", diff); /* Outputs "1". */
printf("*(p + (q - p)) = %d\n", *(p + diff)); /* Outputs "4". */
```

Operadores de Acceso

Los operadores de acceso miembro (DOT `.` Y la flecha `->`) se utilizan para acceder a un miembro de una `struct` .

Miembro de objeto

Se evalúa en el lvalor que denota el objeto que es miembro del objeto accedido.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Outputs ".x = 42, .y = 123". */
```

Miembro de objeto apuntado

Azúcar sintáctica para la desreferenciación seguida por el acceso de miembros. Efectivamente, una expresión de la forma `x->y` es una abreviatura de `(*x).y` - pero el operador de flecha es mucho más claro, especialmente si los punteros de la estructura están anidados.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
```

```
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i\n", p->x, p->y); /* Outputs ".x = 42, .y = 123". */
printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Also outputs ".x = 42, .y = 123". */
```

Dirección de

El unario `&` operador es la dirección del operador. Evalúa la expresión dada, donde el objeto resultante debe ser un valor l. Luego, se evalúa en un objeto cuyo tipo es un puntero al tipo del objeto resultante y contiene la dirección del objeto resultante.

```
int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-
defined A. */
```

Desreferencia

El operador unario `*` referencia a un puntero. Se evalúa en el lvalor resultante de la desreferenciación del puntero que resulta de evaluar la expresión dada.

```
int x = 42;
int *p = &x;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 42, *p = 42". */

*p = 123;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 123, *p = 123". */
```

Indexación

La indexación es azúcar sintáctica para la adición de punteros seguida de la eliminación de referencias. Efectivamente, una expresión de la forma `a[i]` es equivalente a `*(a + i)` , pero se prefiere la notación explícita del subíndice.

```
int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */
```

Intercambiabilidad de la indexación.

Agregar un puntero a un entero es una operación conmutativa (es decir, el orden de los operandos no cambia el resultado), por lo tanto, `pointer + integer == integer + pointer` .

Una consecuencia de esto es que `arr[3]` y `3[arr]` son equivalentes.

```
printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */
```

El uso de una expresión `3[arr]` lugar de `arr[3]` generalmente no se recomienda, ya que afecta la legibilidad del código. Tiende a ser popular en concursos de programación ofuscados.

Operador de llamada de función

El primer operando debe ser un puntero de función (un designador de función también es aceptable porque se convertirá en un puntero a la función), identificando la función a llamar, y todos los demás operandos, si los hay, se conocen colectivamente como los argumentos de la llamada de función. . Se evalúa en el valor de retorno que resulta de llamar a la función apropiada con los argumentos respectivos.

```
int myFunction(int x, int y)
{
    return x * 2 + y;
}

int (*fn)(int, int) = &myFunction;
int x = 42;
int y = 123;

printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereference
explicitly */
```

Operadores de Bitwise

Los operadores bitwise se pueden usar para realizar operaciones a nivel de bit en variables.
A continuación se muestra una lista de los seis operadores bitwise admitidos en C:

Símbolo	Operador
Y	en el bit y
|	bitwise inclusive O
^	OR exclusivo a nivel de bit (XOR)
~	bitwise no (complemento de uno)
<<	desplazamiento lógico a la izquierda
>>	cambio lógico a la derecha

El siguiente programa ilustra el uso de todos los operadores bitwise:

```
#include <stdio.h>

int main(void)
{
   unsigned int a = 29;    /* 29 = 0001 1101 */
   unsigned int b = 48;    /* 48 = 0011 0000 */
```

```
    int c = 0;

    c = a & b;              /* 32 = 0001 0000 */
    printf("%d & %d = %d\n", a, b, c );

    c = a | b;              /* 61 = 0011 1101 */
    printf("%d | %d = %d\n", a, b, c );

    c = a ^ b;              /* 45 = 0010 1101 */
    printf("%d ^ %d = %d\n", a, b, c );

    c = ~a;                 /* -30 = 1110 0010 */
    printf("~%d = %d\n", a, c );

    c = a << 2;             /* 116 = 0111 0100 */
    printf("%d << 2 = %d\n", a, c );

    c = a >> 2;             /* 7 = 0000 0111 */
    printf("%d >> 2 = %d\n", a, c );

    return 0;
}
```

Las operaciones bitwise con tipos firmados deben evitarse porque el bit de signo de tal representación de bit tiene un significado particular. Se aplican restricciones particulares a los operadores de turno:

- El desplazamiento a la izquierda de un bit 1 en el bit firmado es erróneo y conduce a un comportamiento indefinido.
- El desplazamiento a la derecha de un valor negativo (con el bit de signo 1) se define por la implementación y, por lo tanto, no es portátil.
- Si el valor del operando derecho de un operador de cambio es negativo o es mayor o igual que el ancho del operando izquierdo promovido, el comportamiento no está definido.

Enmascaramiento:

Enmascaramiento se refiere al proceso de extracción de los bits deseados de (o la transformación de los bits deseados en) una variable mediante el uso de operaciones lógicas a nivel de bits. El operando (una constante o variable) que se utiliza para realizar el enmascaramiento se denomina *máscara* .

El enmascaramiento se utiliza de muchas maneras diferentes:

- Para decidir el patrón de bits de una variable entera.
- Para copiar una parte de un patrón de bits dado a una nueva variable, mientras que el resto de la nueva variable se rellena con 0s (utilizando AND a nivel de bits)
- Para copiar una parte de un patrón de bits dado a una nueva variable, mientras que el resto de la nueva variable se rellena con 1s (usando bitwise OR).
- Para copiar una parte de un patrón de bits dado a una nueva variable, mientras que el resto del patrón de bits original se invierte dentro de la nueva variable (utilizando OR exclusivo en modo de bits).

La siguiente función utiliza una máscara para mostrar el patrón de bits de una variable:

```
#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;
    word = CHAR_BIT * sizeof(int);
    mask = mask << (word - 1);    /* shift 1 to the leftmost position */
    for(i = 1; i <= word; i++)
    {
        x = (u & mask) ? 1 : 0;  /* identify the bit */
        printf("%d", x);         /* print bit value */
        mask >>= 1;              /* shift mask to the right by 1 bit */
    }
}
```

_ Alineación de

C11

Consulta el requisito de alineación para el tipo especificado. El requisito de alineación es una potencia integral positiva de 2 que representa el número de bytes entre los cuales se pueden asignar dos objetos del tipo. En C, el requisito de alineación se mide en `size_t` .

El nombre de tipo no puede ser un tipo incompleto ni un tipo de función. Si se utiliza una matriz como tipo, se usa el tipo del elemento de matriz.

Con frecuencia se accede a este operador a través de la macro de conveniencia `alignof` desde `<stdalign.h>` .

```
int main(void)
{
    printf("Alignment of char = %zu\n", alignof(char));
    printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
    printf("alignof(float[10]) = %zu\n", alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
            alignof(struct {char c; int n;}));
}
```

Salida posible:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://en.cppreference.com/w/c/language/_Alignof

Comportamiento a corto circuito de operadores lógicos.

El cortocircuito es una funcionalidad que omite la evaluación de partes de una condición (if / while / ...) cuando puede. En el caso de una operación lógica en dos operandos, el primer operando se

evalúa (a verdadero o falso) y si hay un veredicto (es decir, el primer operando es falso cuando se usa &&, el primer operando es verdadero cuando se usa ||) el segundo operando es no evaluado.

Ejemplo:

```
#include <stdio.h>

int main(void) {
  int a = 20;
  int b = -5;

  /* here 'b == -5' is not evaluated,
     since a 'a != 20' is false. */
  if (a != 20 && b == -5) {
    printf("I won't be printed!\n");
  }

  return 0;
}
```

Compruébelo usted mismo:

```
#include <stdio.h>

int print(int i) {
  printf("print function %d\n", i);
  return i;
}

int main(void) {
  int a = 20;

  /* here 'print(a)' is not called,
     since a 'a != 20' is false. */
  if (a != 20 && print(a)) {
    printf("I won't be printed!\n");
  }

  /* here 'print(a)' is called,
     since a 'a == 20' is true. */
  if (a == 20 && print(a)) {
    printf("I will be printed!\n");
  }

  return 0;
}
```

Salida:

```
$ ./a.out
print function 20
I will be printed!
```

El cortocircuito es importante cuando se quiere evitar evaluar términos que son (computacionalmente) costosos. Además, puede afectar considerablemente el flujo de su

programa como en este caso: ¿Por qué este programa se imprime "bifurcado"? ¿4 veces?

Lea Los operadores en línea: https://riptutorial.com/es/c/topic/256/los-operadores

Capítulo 43: Macros x

Introducción

Las macros X son una técnica basada en preprocesador para minimizar el código repetitivo y mantener las correspondencias de datos / código. Se admiten múltiples expansiones de macros distintas basadas en un conjunto común de datos al representar todo el grupo de expansiones a través de una sola macro maestra, con el texto de reemplazo de esa macro que consiste en una secuencia de expansiones de una macro interna, una para cada dato. La macro interna se denomina tradicionalmente `X()` , de ahí el nombre de la técnica.

Observaciones

Se espera que el usuario de una macro maestra de estilo X-macro proporcione su propia definición para la macro interna `X()` , y dentro de su alcance para expandir la macro maestra. Las referencias macro internas del maestro se expanden de acuerdo con la definición de `X()` del usuario. De esta manera, la cantidad de código repetitivo repetitivo en el archivo de origen se puede reducir (aparece solo una vez, en el texto de reemplazo de `X()`), como lo favorecen los adherentes a la filosofía de "No repetirse" (DRY).

Además, al redefinir `X()` y expandir la macro maestra una o más veces, las macros X pueden facilitar el mantenimiento de los datos y el código correspondientes; una expansión de la macro declara los datos (como elementos de la matriz o miembros de la enumeración, por ejemplo), y Las otras expansiones producen el código correspondiente.

Aunque el nombre de "X-macro" proviene del nombre tradicional de la macro interna, la técnica no depende de ese nombre en particular. Cualquier nombre de macro válido puede ser usado en su lugar.

Las criticas incluyen

- Los archivos de origen que se basan en macros X son más difíciles de leer;
- como todas las macros, las macros X son estrictamente textuales, no proporcionan ningún tipo de seguridad; y
- Las macros X proporcionan la *generación de código* . En comparación con las alternativas basadas en funciones de llamada, las macros X hacen que el código sea más grande.

Una buena explicación de las macros de X se puede encontrar en el artículo de Randy Meyers [X-Macros] en Dr. Dobbs (http://www.drdobbs.com/the-new-cx-macros/184401387) .

Examples

Uso trivial de X-macros para printfs

```
/* define a list of preprocessor tokens on which to call X */
```

```
#define X_123 X(1) X(2) X(3)

/* define X to use */
#define X(val) printf("X(%d) made this print\n", val);
X_123
#undef X
/* good practice to undef X to facilitate reuse later on */
```

Este ejemplo dará como resultado que el preprocesador genere el siguiente código:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Enum valor e identificador

```
/* declare items of the enum */
#define FOREACH \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */

/* define the enum values */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* convert an enum value to its identifier */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

A continuación, puede usar el valor enumerado en su código e imprimir fácilmente su identificador utilizando:

```
printf("%s\n", enum2string(MyEnum_item2));
```

Extensión: Da la macro X como argumento

El enfoque de la X-macro se puede generalizar un poco al hacer que el nombre de la macro "X" sea un argumento de la macro maestra. Esto tiene las ventajas de ayudar a evitar colisiones de nombres de macro y de permitir el uso de una macro de propósito general como la macro "X".

Como siempre con las macros X, la macro maestra representa una lista de elementos cuyo significado es específico de esa macro. En esta variación, tal macro podría definirse así:

```
/* declare list of items */
```

```
#define ITEM_LIST(X) \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */
```

Uno podría entonces generar un código para imprimir los nombres de los artículos de esta manera:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

Eso se expande a este código:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

A diferencia de las macros X estándar, donde el nombre "X" es una característica incorporada de la macro maestra, con este estilo puede ser innecesario o incluso no deseable volver a definir la macro utilizada como argumento (`PRINTSTRING` en este ejemplo).

Codigo de GENERACION

Las macros X se pueden usar para la generación de código, al escribir código repetitivo: iterar sobre una lista para realizar algunas tareas o declarar un conjunto de constantes, objetos o funciones.

Aquí usamos X-macros para declarar una enumeración que contiene 4 comandos y un mapa de sus nombres como cadenas

Luego podemos imprimir los valores de cadena de la enumeración.

```
/* All our commands */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
  COMMANDS(ENUM_NAME)
};
#undef ENUM_NAME

/* generate the string table */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
  COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP
```

```
/* the following prints "Quit\n": */
printf("%s\n", commandNames[cmdQuit]());
```

De manera similar, podemos generar una tabla de salto para llamar a funciones mediante el valor enum.

Esto requiere que todas las funciones tengan la misma firma. Si no toman argumentos y devuelven un int, pondríamos esto en un encabezado con la definición de enumeración:

```
/* declare all functions as extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* declare the function pointer type and the jump table  */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

Todo lo siguiente puede estar en diferentes unidades de compilación, asumiendo que la parte anterior se incluye como encabezado:

```
/* generate the jump table */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
  COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* call the save command like this: */
int result = commandJumpTable[cmdSave]();

/* somewhere else, we need the implementations of the commands */
int doCmdOpen(void) {/* code performing open command */}
int doCmdClose(void) {/* code performing close command */}
int doCmdSave(void) {/* code performing save command */}
int doCmdQuit(void) {/* code performing quit command */}
```

Un ejemplo de esta técnica utilizada en código real es el envío de comandos de GPU en Chromium .

Lea Macros x en línea: https://riptutorial.com/es/c/topic/628/macros-x

Capítulo 44: Manejo de errores

Sintaxis

- #include <errno.h>
- int errno / * implementación definida * /
- #include <string.h>
- char * strerror (int errnum);
- #include <stdio.h>
- vacío perror (const char * s);

Observaciones

Tenga en cuenta que `errno` no es necesariamente una variable, pero que la sintaxis es solo una indicación de cómo *podría* ser declarada. En muchos sistemas modernos con interfaces de hilo, `errno` es una macro que se resuelve en un objeto que es local al hilo actual.

Examples

errno

Cuando falla una función de biblioteca estándar, a menudo establece errno en el código de error apropiado. El estándar C requiere que se establezcan al menos 3 valores para errno:

Valor	Sentido
EDOM	Error de dominio
ERANGE	Error de rango
EILSEQ	Secuencia de caracteres multi-byte ilegal

estridente

Si `perror` no es lo suficientemente flexible, puede obtener una descripción de error legible por el `strerror` llamando a `strerror` desde <string.h> .

```
int main(int argc, char *argv[])
{
    FILE *fout;
    int last_error = 0;

    if ((fout = fopen(argv[1], "w")) == NULL) {
        last_error = errno;
         /* reset errno and continue */
         errno = 0;
```

```
    }

    /* do some processing and try opening the file differently, then */


    if (last_error) {
        fprintf(stderr, "fopen: Could not open %s for writing: %s",
                argv[1], strerror(last_error));
        fputs("Cross fingers and continue", stderr);
    }

    /* do some other processing */

    return EXIT_SUCCESS;
}
```

perror

Para imprimir un mensaje de error legible por el usuario a `stderr` , llame a `perror` desde <stdio.h>.

```
int main(int argc, char *argv[])
{
   FILE *fout;

   if ((fout = fopen(argv[1], "w")) == NULL) {
      perror("fopen: Could not open file for writing");
      return EXIT_FAILURE;
   }
return EXIT_SUCCESS;
}
```

Esto imprimirá un mensaje de error referente al valor actual de `errno` .

Lea Manejo de errores en línea: https://riptutorial.com/es/c/topic/2486/manejo-de-errores

Capítulo 45: Manejo de señales

Sintaxis

- void (* señal (int sig, void (* func) (int))) (int);

Parámetros

Parámetro	Detalles
sig	La señal para configurar el manejador de señales en, uno de `SIGABRT` , `SIGFPE` , `SIGILL` , `SIGTERM` , `SIGINT` , `SIGSEGV` o algún valor definido de implementación
función	El controlador de señales, que es uno de los siguientes: `SIG_DFL` , para el controlador predeterminado, `SIG_IGN` para ignorar la señal, o un puntero de función con la firma `void foo(int sig);` .

Observaciones

El uso de manejadores de señales con solo las garantías del estándar C impone varias limitaciones que pueden o no pueden hacerse en el manejador de señales definido por el usuario.

- Si la función definida por el usuario regresa mientras se maneja `SIGSEGV` , `SIGFPE` , `SIGILL` o cualquier otra interrupción de hardware definida por la implementación, el comportamiento no está definido por el estándar C. Esto se debe a que la interfaz de C no proporciona medios para cambiar el estado defectuoso (p. Ej., Después de una división por `0`) y, por lo tanto, cuando el programa regresa se encuentra exactamente en el mismo estado erróneo que antes de que ocurriera la interrupción del hardware.
- Si se llamó a la función definida por el usuario como resultado de una llamada para `abort` o `raise` , el manejador de señal no tiene permitido llamar `raise` , nuevamente.
- Las señales pueden llegar a la mitad de cualquier operación y, por lo tanto, la indivisibilidad de las operaciones generalmente no puede garantizarse ni el manejo de la señal funciona bien con la optimización. Por lo tanto, todas las modificaciones a los datos en un controlador de señales deben ser a variables
    - de tipo `sig_atomic_t` (todas las versiones) o un tipo atómico sin bloqueo (desde C11, opcional)
    - que son `volatile` calificados.
- Otras funciones de la biblioteca estándar de C generalmente no respetarán estas restricciones, ya que pueden cambiar las variables en el estado global del programa. El estándar de C sólo hace garantías para `abort` , `_Exit` (desde C99), `quick_exit` (desde C11), `signal` (para el mismo número de señal), y algunas operaciones atómicas (desde C11).

El comportamiento no está definido por el estándar C si se viola alguna de las reglas anteriores. Las plataformas pueden tener extensiones específicas, pero estas generalmente no son portátiles más allá de esa plataforma.

- Por lo general, los sistemas tienen su propia lista de funciones que son *seguras para señales asíncronas* , es decir, funciones de biblioteca C que pueden usarse desde un controlador de señales. Por ejemplo, a menudo `printf` está entre estas funciones.
- En particular, el estándar C no define mucho sobre la interacción con su interfaz de subprocesos (desde C11) o cualquier biblioteca de subprocesos específica de la plataforma, como los subprocesos POSIX. Dichas plataformas deben especificar la interacción de dichas bibliotecas de hilos con señales por sí mismas.

Examples

Manejo de señales con “señal ()”

Los números de señal pueden ser sincrónicos (como `SIGSEGV` - falla de segmentación) cuando se activan por un mal funcionamiento del programa en sí mismo o asíncronos (como `SIGINT` - atención interactiva) cuando se inician desde fuera del programa, por ejemplo, al `Cntrl-C` una tecla como `Cntrl-C` .

La función `signal()` es parte del estándar ISO C y se puede usar para asignar una función para manejar una señal específica

```
#include <stdio.h>  /* printf() */
#include <stdlib.h> /* abort()  */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Catched: %d\n", sig);

    /* abort is safe to call */
    abort();
}

sig_atomic_t volatile finished = 0;

void handler(int sig)
{
    switch (sig) {
    /* hardware interrupts should not return */
    case SIGSEGV:
    case SIGFPE:
    case SIGILL:
```

C11

```
      /* quick_exit is safe to call */
      quick_exit(EXIT_FAILURE);
```

```
      /* use _Exit in pre-C11 */
      _Exit(EXIT_FAILURE);
```

```
    default:
       /* Reset the signal to the default handler,
          so we will not be called again if things go
          wrong on return. */
      signal(sig, SIG_DFL);
      /* let everybody know that we are finished */
      finished = sig;
      return;
   }
}

int main(void)
{

    /* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access */
    if (signal(SIGSEGV, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGSEGV");
        return EXIT_FAILURE;
    }

    /* Catch the SIGTERM signal, termination request */
    if (signal(SIGTERM, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGTERM");
        return EXIT_FAILURE;
    }

    /* Ignore the SIGINT signal, by setting the handler to `SIG_IGN`. */
    signal(SIGINT, SIG_IGN);


    /* Do something that takes some time here, and leaves
       the time to terminate the program from the keyboard. */

    /* Then: */

    if (finished) {
       fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);
        return EXIT_FAILURE;
    }


    /* Try to force a segmentation fault, and raise a SIGSEGV */
    {
      char* ptr = 0;
      *ptr = 0;
    }

    /* This should never be executed */
    return EXIT_SUCCESS;
}
```

El uso de `signal()` impone importantes limitaciones a lo que se le permite hacer dentro de los manejadores de señales, consulte las observaciones para obtener más información.

POSIX recomienda el uso de `sigaction()` lugar de `signal()` , debido a su comportamiento

subespecificado y variaciones significativas en la implementación. POSIX también define muchas más señales que el estándar ISO C, incluyendo `SIGUSR1` y `SIGUSR2` , que el programador puede usar libremente para cualquier propósito.

Lea Manejo de señales en línea: https://riptutorial.com/es/c/topic/453/manejo-de-senales

Capítulo 46: Marcos de prueba

Introducción

Muchos desarrolladores utilizan pruebas unitarias para verificar que su software funciona como se espera. Las pruebas unitarias verifican pequeñas unidades de piezas más grandes de software y aseguran que las salidas coincidan con las expectativas. Los marcos de prueba facilitan las pruebas unitarias al proporcionar servicios de configuración / desmontaje y coordinar las pruebas.

Hay muchos marcos de prueba de unidad disponibles para C. Por ejemplo, Unity es un marco de C puro. La gente a menudo usa marcos de prueba de C ++ para probar el código C; Hay muchos marcos de prueba de C ++ también.

Observaciones

Arnés de prueba:

TDD - Test Driven Development:

Prueba mecanismos dobles en C:

1. Sustitución de tiempo de enlace
2. Función de puntero de sustitución.
3. Sustitución de preprocesador
4. Combinación de tiempo de enlace y sustitución de puntero de función

Nota sobre los marcos de prueba de C ++ utilizados en C: el uso de marcos de C ++ para probar un programa de C es una práctica bastante común como se explica aquí .

Examples

CppUTest

CppUTest es un marco de estilo xUnit para pruebas de unidad C y C ++. Está escrito en C ++ y tiene como objetivo la portabilidad y la simplicidad en el diseño. Es compatible con la detección de fugas de memoria, la creación de simulacros y la ejecución de sus pruebas junto con Google Test. Viene con scripts de ayuda y proyectos de muestra para Visual Studio y Eclipse CDT.

```
#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>


TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}

/* Test runner may be provided options, such
```

```
   as to enable colored output, to run only a
   specific test or a group of tests, etc. This
   will return the number of failed tests. */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}
```

Un grupo de prueba puede tener un método `setup()` y un método `teardown()` . El método de `setup` se llama antes de cada prueba y el método `teardown()` se llama después. Ambos son opcionales y pueden omitirse independientemente. Otros métodos y variables también pueden declararse dentro de un grupo y estarán disponibles para todas las pruebas de ese grupo.

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```

Unity Test Framework

Unity es un marco de prueba de estilo xUnit para pruebas de unidad C. Está escrito completamente en C y es portátil, rápido, simple, expresivo y extensible. Está diseñado para ser especialmente útil también para pruebas unitarias de sistemas integrados.

Un caso de prueba simple que verifica el valor de retorno de una función, podría tener el siguiente aspecto

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

Un archivo de prueba completo podría verse como:

```
#include "unity.h"
#include "UnitUnderTest.h" /* The unit to be tested. */
```

```
void setUp (void) {} /* Is run before every test, put unit init calls here. */
void tearDown (void) {} /* Is run after every test, put unit clean-up calls here. */

void test_TheFirst(void)
{
    TEST_IGNORE_MESSAGE("Hello world!"); /* Ignore this test but print a message. */
}

int main (void)
{
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst); /* Run the test. */
    return UNITY_END();
}
```

Unity viene con algunos proyectos de ejemplo, archivos make y algunos scripts de rake de Ruby que ayudan a que la creación de archivos de prueba más largos sea un poco más fácil.

CMocka

CMocka es un marco de prueba de unidad elegante para C con soporte para objetos simulados. Solo requiere la biblioteca C estándar, funciona en una variedad de plataformas informáticas (incluidas las integradas) y con diferentes compiladores. Tiene un tutorial sobre pruebas con simulacros, documentación de API y una variedad de ejemplos .

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}

void null_test_fail (void ** state)
{
    assert_true (0);
}

/* These functions will be used to initialize
   and clean resources up after each test run */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
}


int main (void)
{
    const struct CMUnitTest tests [] =
    {
        cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
    };
```

```
    /* If setup and teardown functions are not
       needed, then NULL may be passed instead */

    int count_fail_tests =
        cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}
```

Lea Marcos de prueba en línea: https://riptutorial.com/es/c/topic/6779/marcos-de-prueba

Capítulo 47: Matemáticas estándar

Sintaxis

- #include <math.h>
- doble pow (doble x, doble y);
- float powf (float x, float y);
- powl doble largo (doble x largo, doble y largo);

Observaciones

1. Para enlazar con la biblioteca matemática use `-lm` con gcc flags.
2. Un programa portátil que necesita verificar un error de una función matemática debe establecer `errno` en cero y hacer la siguiente llamada `feclearexcept(FE_ALL_EXCEPT);` Antes de llamar a una función matemática. Al regresar de la función matemática, si `errno` es distinto de cero, o la siguiente llamada devuelve `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW);` distinto de `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW);` entonces se produjo un error en la función matemática. Lea la página del manual de math_error para más información.

Examples

Resto de punto flotante de doble precisión: fmod ()

Esta función devuelve el resto de punto flotante de la división de `x/y` . El valor devuelto tiene el mismo signo que x.

```
#include <math.h> /* for fmod() */
#include <stdio.h> /* for printf() */

int main(void)
{
    double x = 10.0;
    double y = 5.1;

    double modulus = fmod(x, y);

    printf("%lf\n", modulus); /* f is the same as lf. */

    return 0;
}
```

Salida:

```
4.90000
```

Importante: use esta función con cuidado, ya que puede devolver valores inesperados debido a

la operación de valores de punto flotante.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf("%f\n", fmod(1, 0.1));
    printf("%19.17f\n", fmod(1, 0.1));
    return 0;
}
```

Salida:

```
0.1
0.09999999999999995
```

Resto de punto flotante de precisión simple y doble precisión: fmodf (), fmodl ()

C99

Estas funciones devuelven el resto en coma flotante de la división de `x/y` . El valor devuelto tiene el mismo signo que x.

Precisión simple:

```
#include <math.h> /* for fmodf() */
#include <stdio.h> /* for printf() */

int main(void)
{
    float x = 10.0;
    float y = 5.1;

    float modulus = fmodf(x, y);

    printf("%f\n", modulus); /* lf would do as well as modulus gets promoted to double. */
}
```

Salida:

```
4.90000
```

Doble Precision Doble:

```
#include <math.h> /* for fmodl() */
#include <stdio.h> /* for printf() */

int main(void)
{
    long double x = 10.0;
    long double y = 5.1;
```

```
    long double modulus = fmodl(x, y);

    printf("%Lf\n", modulus); /* Lf is for long double. */
}
```

Salida:

```
4.90000
```

Funciones de energía: pow (), powf (), powl ()

El siguiente código de ejemplo calcula la suma de las series *1 + 4 (3 + 3 ^ 2 + 3 ^ 3 + 3 ^ 4 + ... + 3 ^ N)* usando la familia pow () de la biblioteca matemática estándar.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>

int main()
{
        double pwr, sum=0;
        int i, n;

        printf("\n1+4(3+3^2+3^3+3^4+...+3^N)=?\nEnter N:");
        scanf("%d",&n);
        if (n<=0) {
                printf("Invalid power N=%d", n);
                return -1;
        }

        for (i=0; i<n+1; i++) {
                errno = 0;
                feclearexcept(FE_ALL_EXCEPT);
                pwr = powl(3,i);
                if (fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
                        FE_UNDERFLOW)) {
                        perror("Math Error");
                }
                sum += i ? pwr : 0;
                printf("N= %d\tS= %g\n", i, 1+4*sum);
        }

        return 0;
}
```

Ejemplo de salida:

```
1+4(3+3^2+3^3+3^4+...+3^N)=?
Enter N:10
N= 0    S= 1
N= 1    S= 13
N= 2    S= 49
N= 3    S= 157
N= 4    S= 481
```

```
N= 5     S= 1453
N= 6     S= 4369
N= 7     S= 13117
N= 8     S= 39361
N= 9     S= 118093
N= 10     S= 354289
```

Lea Matemáticas estándar en línea: https://riptutorial.com/es/c/topic/3170/matematicas-estandar

Capítulo 48: Montaje en línea

Observaciones

El ensamblaje en línea es la práctica de agregar instrucciones de ensamblaje en medio del código fuente de C. Ninguna norma ISO C requiere soporte de montaje en línea. Como no es obligatorio, la sintaxis para el ensamblaje en línea varía de compilador a compilador. Aunque normalmente se admite, hay muy pocas razones para usar el ensamblaje en línea y muchas razones para no hacerlo.

Pros

1. **Rendimiento** Al escribir las instrucciones de ensamblaje específicas para una operación, puede lograr un mejor rendimiento que el código de ensamblaje generado por el compilador. Tenga en cuenta que estas ganancias de rendimiento son raras. En la mayoría de los casos, puede lograr mejores ganancias de rendimiento simplemente reorganizando su código C para que el optimizador pueda hacer su trabajo.
2. **Interfaz de hardware** El controlador de dispositivo o el código de inicio del procesador pueden necesitar algún código de ensamblaje para acceder a los registros correctos y garantizar que ciertas operaciones se realicen en un orden específico con un retraso específico entre las operaciones.

Contras

1. No se garantiza que la sintaxis de **portabilidad del compilador** sea el mismo de un compilador a otro. Si está escribiendo código con ensamblaje en línea que debe ser compatible con compiladores diferentes, use macros de preprocesador (`#ifdef`) para verificar qué compilador se está utilizando. Luego, escriba una sección de ensamblaje en línea por separado para cada compilador compatible.
2. **Portabilidad del procesador** No puede escribir un ensamblaje en línea para un procesador x86 y esperar que funcione en un procesador ARM. El ensamblaje en línea está diseñado para ser escrito para un procesador específico o una familia de procesadores. Si tiene el ensamblaje en línea que desea que sea compatible con diferentes procesadores, use macros de preprocesador para verificar en qué procesador se está compilando el código y para seleccionar la sección de código de ensamblaje correspondiente.
3. **Cambios futuros de rendimiento** El ensamblaje en línea se puede escribir esperando demoras basadas en una cierta velocidad de reloj del procesador. Si el programa se compila para un procesador con un reloj más rápido, es posible que el código de ensamblaje no funcione como se espera.

Examples

Soporte básico de asc gcc

El soporte de ensamblaje básico con gcc tiene la siguiente sintaxis:

```
asm [ volatile ] ( AssemblerInstructions )
```

donde `AssemblerInstructions` es el código de ensamblaje directo para el procesador dado. La palabra clave volátil es opcional y no tiene efecto, ya que gcc no optimiza el código dentro de una declaración básica de asm. `AssemblerInstructions` puede contener varias instrucciones de montaje. Se usa una declaración básica de asm si tiene una rutina de asm que debe existir fuera de una función de C. El siguiente ejemplo es del manual de GCC:

```
 /* Note that this code will not compile with -masm=intel */
 #define DebugBreak() asm("int $3")
```

En este ejemplo, podría usar `DebugBreak()` en otros lugares de su código y ejecutará la instrucción de ensamblaje `int $3` . Tenga en cuenta que aunque gcc no modificará ningún código en una declaración básica de asm, el optimizador aún puede mover las declaraciones de asm consecutivas. Si tiene varias instrucciones de ensamblaje que deben ocurrir en un orden específico, inclúyalas en una declaración asm.

Soporte de asm gcc extendido

El soporte extendido de asm en gcc tiene la siguiente sintaxis:

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ])

 asm [volatile] goto ( AssemblerTemplate
                       :
                       : InputOperands
                       : Clobbers
                       : GotoLabels)
```

donde `AssemblerTemplate` es la plantilla para la instrucción del ensamblador, `OutputOperands` son las variables C que pueden modificarse mediante el código de ensamblaje, `InputOperands` son las variables C utilizadas como parámetros de entrada, `Clobbers` son una lista o registros modificados por el código de ensamblaje y `GotoLabels` son las etiquetas de instrucciones goto que se pueden usar en el código de ensamblaje.

El formato extendido se usa dentro de las funciones C y es el uso más típico del ensamblaje en línea. A continuación se muestra un ejemplo del kernel de Linux para el intercambio de bytes de números de 16 y 32 bits por un procesador ARM:

```
/* From arch/arm/include/asm/swab.h in Linux kernel version 4.6.4 */
#if __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
```

```
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

Cada sección de asm usa la variable `x` como su parámetro de entrada y salida. La función C devuelve el resultado manipulado.

Con el formato extendido de asm, gcc puede optimizar las instrucciones de ensamblaje en un bloque de asm siguiendo las mismas reglas que utiliza para optimizar el código C. Si desea que su sección de asm permanezca intacta, use la palabra clave `volatile` para la sección de asm.

Gcc Inline Assembly en macros

Podemos colocar instrucciones de ensamblaje dentro de una macro y usar la macro como llamaríamos a una función.

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// some definition and assignment
unsigned char sbox[size][size];
unsigned char sbox[size][size];

///Using
mov(state[0][1], sbox[si][sj]);
```

El uso de instrucciones de ensamblaje integradas en el código C puede mejorar el tiempo de ejecución de un programa. Esto es muy útil en situaciones críticas como algoritmos criptográficos como AES. Por ejemplo, para una operación de cambio simple que se necesita en el algoritmo AES, podemos sustituir una instrucción de montaje directo `Rotate Right` con el operador de cambio C `>>` .

En una implementación de 'AES256', en la función 'AddRoundKey ()' tenemos algunas declaraciones como esta:

```
unsigned int w;          // 32-bit
unsigned char subkey[4]; // 8-bit, 4*8 = 32

subkey[0] = w >> 24;     // hold 8 bit, MSB, leftmost group of 8-bits
subkey[1] = w >> 16;     // hold 8 bit, second group of 8-bit from left
subkey[2] = w >> 8;      // hold 8 bit, second group of 8-bit from right
subkey[3] = w;           // hold 8 bit, LSB, rightmost group of 8-bits
```

```
/// subkey <- w
```

Simplemente asignan el valor de bit de `w` a la matriz de `subkey` .

Podemos cambiar tres Mayús + asignar y una Asignar C expresión con solo un ensamblaje de operación `Rotate Right` .

```
__asm__ ("l.ror  %0,%1,%2" : "=r" (* (unsigned int *) subkey)  : "r" (w), "r" (0x10));
```

El resultado final es exactamente igual.

Lea Montaje en línea en línea: https://riptutorial.com/es/c/topic/4263/montaje-en-linea

Capítulo 49: Multihilo

Introducción

En C11 hay una biblioteca de subprocesos estándar, `<threads.h>` , pero no hay un compilador conocido que aún lo implemente. Por lo tanto, para usar los subprocesos múltiples en C, debe usar implementaciones específicas de la plataforma como la biblioteca de subprocesos POSIX (a menudo denominada pthreads) utilizando el encabezado `pthread.h` .

Sintaxis

- thrd_t // Tipo de objeto completo definido por la implementación que identifica un hilo
- int thrd_create (thrd_t * thr, thrd_start_t func, void * arg); // crea un hilo
- int thrd_equal (thrd_t thr0, thrd_t thr1); // Verificar si los argumentos se refieren al mismo hilo.
- thr_t thrd_current (void); // Devuelve el identificador del hilo que lo llama
- int thrd_sleep (const struct timespec * duration, struct timespec * restante); // Suspender la ejecución del hilo de llamada por al menos un tiempo dado
- void thrd_yield (void); // Permitir que se ejecuten otros subprocesos en lugar del subproceso que lo llama
- _Noreturn void thrd_exit (int res); // Termina el hilo el hilo que lo llama
- int thrd_detatch (thrd_t thr; // separa un hilo dado del entorno actual
- int thrd_join (thrd_t thr, int * res); // Bloquea el hilo actual hasta que el hilo dado termine

Observaciones

El uso de subprocesos puede introducir un comportamiento adicional indefinido, como http://www.riptutorial.com/c/example/2622/data-race . Para el acceso sin carreras a las variables que se comparten entre los distintos subprocesos, C11 proporciona la funcionalidad de `mtx_lock()` o el http://www.riptutorial.com/c/topic/4924/atomics y las funciones asociadas (opcional) en `stdatomic.h` .

Examples

C11 hilos ejemplo simple

```
#include <threads.h>
#include <stdio.h>

int run(void *arg)
{
    printf("Hello world of C11 threads.");

    return 0;
}
```

```
int main(int argc, const char *argv[])
{
    thrd_t thread;
    int result;

    thrd_create(&thread, run, NULL);

    thrd_join(&thread, &result);

    printf("Thread return %d at the end\n", result);
}
```

Lea Multihilo en línea: https://riptutorial.com/es/c/topic/10489/multihilo

Capítulo 50: Parámetros de función

Observaciones

En C, es común usar valores de retorno para denotar errores que ocurren; y para devolver datos mediante el uso de punteros pasados. Esto se puede hacer por múltiples razones; incluyendo no tener que asignar memoria en el montón o usar una asignación estática en el punto donde se llama la función.

Examples

Usando parámetros de puntero para devolver múltiples valores

Un patrón común en C, para imitar fácilmente la devolución de múltiples valores de una función, es usar punteros.

```
#include <stdio.h>

void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}
```

Pasando en Arrays a Funciones

```
int getListOfFriends(size_t size, int friend_indexes[]) {
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = i;
  }
}
```

C99 C11

```
/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
   In C11 VLAs are optional. */
void getListOfFriends(size_t size, int friend_indexes[static size]) {
```

```
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = 1;
  }
}
```

Aquí, la `static` dentro de `[]` del parámetro de función, solicita que la matriz de argumentos tenga al menos la cantidad de elementos especificados (es decir, elementos de `size`). Para poder usar esa función, debemos asegurarnos de que el parámetro de `size` aparezca antes del parámetro de matriz en la lista.

Utilice `getListOfFriends()` esta manera:

```
#define LIST_SIZE (50)

int main(void) {
  size_t size_of_list = LIST_SIZE;
  int friends_indexes[size_of_list];

  getListOfFriends(size_of_list, friend_indexes); /* friend_indexes decays to a pointer to the
                                                     address of its 1st element:
                                                                      &friend_indexes[0] */

  /* Here friend_indexes carries: {0, 1, 2, ..., 49}; */

  return 0;
}
```

Ver también

Pasar matrices multidimensionales a una función.

Los parámetros se pasan por valor

En C, todos los parámetros de función se pasan por valor, por lo que la modificación de lo que se pasa en funciones de llamada no afectará las variables locales de las funciones de llamada.

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
    return 0;
}
```

Puede usar punteros para permitir que las funciones de los destinatarios modifiquen las variables locales de las funciones de los que llaman. Tenga en cuenta que esto no se *pasa por referencia,* sino que se *pasan* los *valores de* puntero *que* apuntan a las variables locales.

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

Sin embargo, devolver la dirección de una variable local a los resultados del llamante en un comportamiento indefinido. Consulte Desreferenciación de un puntero a una variable más allá de su duración .

Orden de ejecución de parámetros de función

El orden de ejecución de los parámetros no está definido en la programación en C. Aquí puede ejecutarse de izquierda a derecha o de derecha a izquierda. El orden depende de la implementación.

```
#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d\n", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}
```

Ejemplo de función que devuelve la estructura que contiene valores con códigos de error

La mayoría de los ejemplos de una función que devuelve un valor implican proporcionar un puntero como uno de los argumentos para permitir que la función modifique el valor apuntado, similar al siguiente. El valor de retorno real de la función suele ser algún tipo, como un `int` para indicar el estado del resultado, ya sea que haya funcionado o no.

```
int func (int *pIvalue)
{
    int iRetStatus = 0;             /* Default status is no change */

    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45;   /* Modify the value pointed to */
        iRetStatus = 1;             /* indicate value was changed */
    }

    return iRetStatus;              /* Return an error code */
}
```

Sin embargo, también puede usar una `struct` como valor de retorno que le permite devolver tanto un estado de error como otros valores. Por ejemplo.

```
typedef struct {
    int    iStat;      /* Return status */
    int    iValue;     /* Return value */
}  RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}
```

Esta función podría ser utilizada como la siguiente muestra.

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

O podría ser utilizado como el siguiente.

```
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Lea Parámetros de función en línea: https://riptutorial.com/es/c/topic/1006/parametros-de-funcion

Capítulo 51: Pasar arrays 2D a funciones

Examples

Pasar una matriz 2D a una función

Pasar una matriz 2d a una función parece simple y obvio y con gusto escribimos:

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Pero el compilador, aquí GCC en la versión 4.9.4, no lo aprecia bien.

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c11 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:16:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, n, m);
        ^
passarr.c:8:6: note: expected ‘int **’ but argument is of type ‘int (*)[2]’
 void fun1(int **, int, int);
```

Las razones para esto son dobles: el problema principal es que las matrices no son punteros y el segundo inconveniente es la llamada *decadencia del puntero* . Pasar una matriz a una función hará que decaiga la matriz a un puntero al primer elemento de la matriz; en el caso de una matriz 2d, decae a un puntero a la primera fila porque en C las matrices se ordenan primero en la fila.

```
#include <stdio.h>
```

```
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Es necesario pasar el número de filas, que no se pueden calcular.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int rows = ROWS;

  /* works here because array_2d is still in scope and still an array */
  printf("MAIN: %zu\n",sizeof(array_2D)/sizeof(array_2D[0]));

  fun1(array_2D, rows);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int rows)
{
  int i, j;
  int n, m;

  n = rows;
  /* Works, because that information is passed (as "COLS").
     It is also redundant because that value is known at compile time (in "COLS"). */
  m = (int) (sizeof(a[0])/sizeof(a[0][0]));
```

```
  /* Does not work here because the "decay" in "pointer decay" is meant
     literally--information is lost. */
  printf("FUN1: %zu\n",sizeof(a)/sizeof(a[0]));

  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

C99

El número de columnas está predefinido y, por lo tanto, se fija en tiempo de compilación, pero el predecesor del estándar C actual (que era ISO / IEC 9899: 1999, actual es ISO / IEC 9899: 2011) implementó VLAs (TODO: enlace) y aunque el estándar actual lo hizo opcional, casi todos los compiladores C modernos lo admiten (TODO: verifica si MS Visual Studio lo admite ahora).

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];

  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = (i + 1) * (j + 1);
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;

  n = rows;
  /* Does not work anymore, no sizes are specified anymore
```

```
  m = (int) (sizeof(a[0])/sizeof(a[0][0])); */
  m = cols;


  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Esto no funciona, el compilador se queja:

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘fun1’:
passarr.c:168:7: error: invalid use of array with unspecified bounds
       printf("array[%d][%d]=%d\n", i, j, a[i][j]);
```

Se vuelve un poco más claro si intencionalmente cometemos un error en la llamada de la función al cambiar la declaración a `void fun1(int **a, int rows, int cols)` . Eso hace que el compilador se queje de una manera diferente, pero igualmente nebulosa

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:208:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, rows, cols);
        ^
passarr.c:185:6: note: expected ‘int **’ but argument is of type ‘int (*)[(sizetype)(cols)]’
 void fun1(int **, int rows, int cols);
```

Podemos reaccionar de varias maneras, una de ellas es ignorar todo y hacer un malabarismo de punteros ilegibles:

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
```

```
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, *( (*a) + (i * cols + j)));
    }
  }
}
```

O lo hacemos bien y pasamos la información necesaria a `fun1` . Para hacerlo necesitamos reorganizar los argumentos para `fun1` : el tamaño de la columna debe venir antes de la declaración de la matriz. Para mantenerlo más legible, la variable que mantiene el número de filas también ha cambiado su lugar, y ahora es la primera.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int (*)[]);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }
```

```
  fun1(rows, cols, array_2D);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int (*a)[cols])
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Esto parece incómodo para algunas personas, que opinan que el orden de las variables no debería importar. Eso no es un gran problema, simplemente declare un puntero y deje que apunte a la matriz.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int **);

int main(int argc, char **argv)
{
  int rows, cols, i, j;


  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }
  // a "rows" number of pointers to "int". Again a VLA
  int *a[rows];
  // initialize them to point to the individual rows
  for (i = 0; i < rows; i++) {
      a[i] = array_2D[i];
  }
```

```
  fun1(rows, cols, a);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int **a)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Usando matrices planas como matrices 2D

A menudo, la solución más sencilla es simplemente pasar matrices 2D y superiores como memoria plana.

```
/* create 2D array with dimensions determined at runtime */
double *matrix = malloc(width * height * sizeof(double));

/* initialise it (for the sake of illustration we want 1.0 on the diagonal) */
int x, y;
for (y = 0; y < height; y++)
{
    for (x = 0; x < width; x++)
    {
        if (x == y)
            matrix[y * width + x] = 1.0;
        else
            matrix[y * width + x] = 0.0;
    }
}

/* pass it to a subroutine */
 manipulate_matrix(matrix, width, height);


/* do something with the matrix, e.g. scale by 2 */
void manipulate_matrix(double *matrix, int width, int height)
{
    int x, y;

    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            matrix[y * width + x] *= 2.0;
        }
    }
```

```
}
```

Lea Pasar arrays 2D a funciones en línea: https://riptutorial.com/es/c/topic/6862/pasar-arrays-2d-a-funciones

Capítulo 52: Preprocesador y Macros

Introducción

Todos los comandos del preprocesador comienzan con el símbolo `#` hash (libra). La macro AC es solo un comando de preprocesador que se define mediante la directiva `#define` preprocesador. Durante la etapa de preprocesamiento, el preprocesador de C (una parte del compilador de C) simplemente sustituye el cuerpo de la macro donde aparece su nombre.

Observaciones

Cuando un compilador encuentra una macro en el código, realiza un reemplazo de cadena simple, no se realizan operaciones adicionales. Debido a esto, los cambios realizados por el preprocesador no respetan el alcance de los programas en C; por ejemplo, una definición de macro no se limita a estar dentro de un bloque, por lo que no se ve afectada por un `'}'` que termina una declaración de bloque.

El preprocesador, por diseño, no se ha completado, hay varios tipos de cálculo que no puede realizar solo el preprocesador.

Por lo general, los compiladores tienen un indicador de línea de comando (o configuración) que nos permite detener la compilación después de la fase de preprocesamiento e inspeccionar el resultado. En las plataformas POSIX esta bandera es `-E` . Por lo tanto, ejecutar gcc con este indicador imprime la fuente expandida a la salida estándar:

```
$ gcc -E cprog.c
```

A menudo, el preprocesador se implementa como un programa separado, que es invocado por el compilador, el nombre común para ese programa es `cpp` . Varios preprocesadores emiten información de apoyo, como información sobre números de línea, que se utiliza en las fases posteriores de la compilación para generar información de depuración. En el caso de que el preprocesador se base en gcc, la opción -P suprime dicha información.

```
$ cpp -P cprog.c
```

Examples

Inclusión condicional y modificación de la firma de la función condicional.

Para incluir condicionalmente un bloque de código, el preprocesador tiene varias directivas (por ejemplo `#if` , `#ifdef` , `#else` , `#endif` , etc).

```
/* Defines a conditional `printf` macro, which only prints if `DEBUG`
 * has been defined
```

```
 */
#ifdef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Se pueden usar operadores relacionales C normales para la condición `#if`

```
#if __STDC_VERSION__ >= 201112L
/* Do stuff for C11 or higher */
#elif __STDC_VERSION__ >= 199901L
/* Do stuff for C99 */
#else
/* Do stuff for pre C99 */
#endif
```

Las directivas `#if` comportan de manera similar a la instrucción C `if` , solo contendrá expresiones constantes integrales y no emitirá. Admite un operador unario adicional, `defined( identifier )` , que devuelve `1` si el identificador está definido, y `0` contrario.

```
#if defined(DEBUG) && !defined(QUIET)
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Modificación de la firma de la función condicional

En la mayoría de los casos, se espera que la versión de lanzamiento de una aplicación tenga la menor sobrecarga posible. Sin embargo, durante la prueba de una compilación provisional, pueden ser útiles los registros adicionales y la información sobre los problemas encontrados.

Por ejemplo, supongamos que hay alguna función `SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` que al realizar una compilación de prueba se generará un registro sobre su uso. Sin embargo, esta función se usa en múltiples lugares y se desea que al generar el registro, parte de la información sea para saber de dónde se llama la función.

Entonces, al usar la compilación condicional, puede tener algo como lo siguiente en el archivo de inclusión que declara la función. Esto reemplaza la versión estándar de la función con una versión de depuración de la función. El preprocesador se usa para reemplazar las llamadas a la función `SerOpPluAllRead()` con llamadas a la función `SerOpPluAllRead_Debug()` con dos argumentos adicionales, el nombre del archivo y el número de línea donde se usa la función.

La compilación condicional se utiliza para elegir si se reemplaza o no la función estándar con una versión de depuración.

```
#if 0
// function declaration and prototype for our debug version of the function.
SHORT   SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// macro definition to replace function call using old name with debug function with
```

```
additional arguments.
#define SerOpPluAllRead(pPif,usLock) SerOpPluAllRead_Debug(pPif,usLock,__FILE__,__LINE__)
#else
// standard function declaration that is normally used with builds.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

Esto le permite anular la versión estándar de la función `SerOpPluAllRead()` con una versión que proporcionará el nombre del archivo y el número de línea en el archivo donde se llama la función.

Hay una consideración importante: *cualquier archivo que use esta función debe incluir el archivo de encabezado donde se utiliza este método para que el preprocesador modifique la función. De lo contrario verá un error de vinculador.*

La definición de la función sería similar a la siguiente. Lo que hace esta fuente es solicitar que el preprocesador cambie el nombre de la función `SerOpPluAllRead()` para que sea `SerOpPluAllRead_Debug()` y para modificar la lista de argumentos para incluir dos argumentos adicionales, un puntero al nombre del archivo donde se llamó la función y el número de línea. en el archivo en el que se utiliza la función.

```
#if defined(SerOpPluAllRead)
// forward declare the replacement function which we will call once we create our log.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT    SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char  xBuffer[256];

    // only print the last 30 characters of the file name to shorten the logs.
    iLen = strlen (aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf (xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, File %s, lineno = %d", pPif-
>husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // now that we have issued the log, continue with standard processing.
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// our special replacement function name for when we are generating logs.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// standard, normal function name (signature) that is replaced with our debug version.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
#endif
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
```

```
}
```

Inclusión de archivo fuente

Los usos más comunes de las directivas de preprocesamiento de `#include` son las siguientes:

```
#include <stdio.h>
#include "myheader.h"
```

`#include` reemplaza la declaración con el contenido del archivo al que se hace referencia. Los corchetes angulares (<>) hacen referencia a los archivos de cabecera instalados en el sistema, mientras que las comillas ("") corresponden a los archivos proporcionados por el usuario.

Las propias macros pueden expandir otras macros una vez, como lo ilustra este ejemplo:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"
    /*  and so on */
#else
    #define INCFILE  "versN.h"
#endif
/* ... */
#include INCFILE
```

Reemplazo de macros

La forma más simple de reemplazo de macro es definir una `manifest constant` , como en

```
#define ARRSIZE 100
int array[ARRSIZE];
```

Esto define una macro *similar a* una *función* que multiplica una variable por `10` y almacena el nuevo valor:

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
int c = 23;

TIMES10(b);   // good: ((b) *= 10);
TIMES10(c);   // good: ((c) *= 10);
TIMES10(5);   // bad:  ((5) *= 10);
```

El reemplazo se realiza antes de cualquier otra interpretación del texto del programa. En la primera llamada a `TIMES10` el nombre `A` de la definición se reemplaza por `b` y el texto expandido se coloca en lugar de la llamada. Tenga en cuenta que esta definición de `TIMES10` no es equivalente a

```
#define TIMES10(A) ((A) = (A) * 10)
```

porque esto podría evaluar el reemplazo de `A` , dos veces, lo que puede tener efectos secundarios no deseados.

Lo siguiente define una macro similar a una función cuyo valor es el máximo de sus argumentos. Tiene las ventajas de trabajar con cualquier tipo compatible de argumentos y de generar código en línea sin la sobrecarga de la función de llamada. Tiene las desventajas de evaluar uno u otro de sus argumentos por segunda vez (incluidos los efectos secundarios) y de generar más código que una función si se invoca varias veces.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43);              /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* Should not be done, due to expression being evaluated twice */
int j = 0, i = 0;
int sideEffect = max(++i, ++j);       /* i == 4 */
```

Debido a esto, las macros que evalúan sus argumentos varias veces generalmente se evitan en el código de producción. Desde C11 existe la característica `_Generic` que permite evitar tales invocaciones múltiples.

Los abundantes paréntesis en las expansiones de macro (lado derecho de la definición) aseguran que los argumentos y la expresión resultante se unen correctamente y se ajusten bien al contexto en el que se llama la macro.

Directiva de error

Si el preprocesador encuentra una directiva `#error` , la compilación se detiene y se imprime el mensaje de diagnóstico incluido.

```
#define DEBUG

#ifdef DEBUG
#error "Debug Builds Not Supported"
#endif

int main(void) {
    return 0;
}
```

Salida posible:

```
$ gcc error.c
error.c: error: #error "Debug Builds Not Supported"
```

#if 0 para bloquear secciones de código

Si hay secciones de código que está considerando eliminar o desea desactivar temporalmente, puede comentarlo con un comentario de bloqueo.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
*/
```

Sin embargo, si el código fuente que ha rodeado con un comentario de bloque tiene comentarios de estilo de bloque en la fuente, el final * / de los comentarios de bloque existentes puede hacer que su nuevo comentario de bloque no sea válido y causar problemas de compilación.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;

    /* Return 5 */
    return i;
}
*/
```

En el ejemplo anterior, el compilador ve las últimas dos líneas de la función y el último '* /', por lo que se compilaría con errores. Un método más seguro es usar una directiva `#if 0` alrededor del código que desea bloquear.

```
#if 0
/* #if 0 evaluates to false, so everything between here and the #endif are
 * removed by the preprocessor. */
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
#endif
```

Un beneficio de esto es que cuando quiere regresar y encontrar el código, es mucho más fácil hacer una búsqueda de "#if 0" que buscar todos sus comentarios.

Otro beneficio muy importante es que puede anidar el código de comentario con `#if 0` . Esto no se puede hacer con comentarios.

Una alternativa al uso de `#if 0` es usar un nombre que no estará `#defined` pero es más descriptivo de por qué se está bloqueando el código. Por ejemplo, si hay una función que parece ser un código muerto inútil, podría usar `#if defined(POSSIBLE_DEAD_CODE)` o `#if defined(FUTURE_CODE_REL_020201)` para el código necesario una vez que haya otra funcionalidad o algo similar. Luego, al volver para eliminar o habilitar esa fuente, esas secciones de la fuente son fáciles de encontrar.

Pegado de ficha

El pegado de token permite pegar dos argumentos de macro. Por ejemplo, `front##back` produce `frontback` . Un ejemplo famoso es el encabezado `<TCHAR.H>` Win32. En el estándar C, se puede escribir `L"string"` para declarar una cadena de caracteres amplia. Sin embargo, la API de Windows le permite a uno convertir entre cadenas de caracteres amplias y cadenas de caracteres estrechas simplemente con `#define` ing `UNICODE` . Para implementar los literales de cadena, `TCHAR.H` usa esto

```
#ifdef UNICODE
#define TEXT(x) L##x
#endif
```

Cada vez que un usuario escribe `TEXT("hello, world")` y UNICODE se define, el preprocesador de C concatena `L` y el argumento de la macro. `L` concatenado con `"hello, world"` da `L"hello, world"` .

Macros predefinidas

Una macro predefinida es una macro que el preprocesador C ya entiende sin necesidad de que el programa la defina. Ejemplos incluyen

Macros obligatorias predefinidas

- `__FILE__` , que proporciona el nombre del archivo fuente actual (un literal de cadena),
- `__LINE__` para el número de línea actual (una constante entera),
- `__DATE__` para la fecha de compilación (un literal de cadena),
- `__TIME__` para el tiempo de compilación (un literal de cadena).

También hay un identificador predefinido relacionado, `__func__` (ISO / IEC 9899: 2011 §6.4.2.2), que *no* es una macro:

> El identificador `__func__` será declarado implícitamente por el traductor como si, inmediatamente después de la llave de apertura de cada definición de función, la declaración:
>
> ```
> static const char __func__[] = "function-name";
> ```
>
> apareció, donde *nombre-función* es el nombre de la función que encierra léxicamente.

`__FILE__` , `__LINE__` y `__func__` son especialmente útiles para propósitos de depuración. Por ejemplo:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", __FILE__, __func__, __LINE__);
```

Los compiladores anteriores a C99, pueden o no ser compatibles con `__func__` o pueden tener una macro que actúa de la misma manera que se nombra de manera diferente. Por ejemplo, gcc usó `__FUNCTION__` en modo C89.

Las siguientes macros permiten solicitar detalles sobre la implementación:

- `__STDC_VERSION__` La versión del estándar C implementada. Este es un entero constante que usa el formato `yyyymmL` (el valor `201112L` para C11, el valor `199901L` para C99; no se definió para C89 / C90)
- `__STDC_HOSTED__` `1` si es una implementación hospedada, de lo contrario `0` .
- `__STDC__` Si es `1` , la implementación cumple con el estándar C.

Otras macros predefinidas (no obligatorio)

ISO / IEC 9899: 2011 §6.10.9.2 Macros de entorno:

- `__STDC_ISO_10646__` Una constante entera de la forma `yyyymmL` (por ejemplo, 199712L). Si se define este símbolo, entonces cada carácter en el conjunto requerido de Unicode, cuando se almacena en un objeto de tipo `wchar_t` , tiene el mismo valor que el identificador corto de ese carácter. El conjunto requerido de Unicode consta de todos los caracteres que se definen en ISO / IEC 10646, junto con todas las enmiendas y correcciones técnicas, a partir del año y mes especificados. Si se utiliza alguna otra codificación, la macro no se definirá y la codificación real utilizada se definirá por la implementación.

- `__STDC_MB_MIGHT_NEQ_WC__` La constante entera 1, destinada a indicar que, en la codificación de `wchar_t` , un miembro del conjunto de caracteres básico no necesita tener un valor de código igual a su valor cuando se usa como carácter único en una constante de carácter entero.

- `__STDC_UTF_16__` La constante entera 1, que pretende indicar que los valores de tipo `char16_t` están codificados en UTF-16. Si se utiliza alguna otra codificación, la macro no se definirá y la codificación real utilizada se definirá por la implementación.

- `__STDC_UTF_32__` La constante entera 1, que pretende indicar que los valores de tipo `char32_t` están codificados en UTF – 32. Si se utiliza alguna otra codificación, la macro no se definirá y la codificación real utilizada se definirá por la implementación.

ISO / IEC 9899: 2011 §6.10.8.3 Macros de funciones condicionales

- `__STDC_ANALYZABLE__` La constante entera 1, con la intención de indicar conformidad con las especificaciones en el anexo L (Análisis).
- `__STDC_IEC_559__` La constante entera 1, que pretende indicar la conformidad con las especificaciones en el anexo F (IEC 60559 aritmética de punto flotante).
- `__STDC_IEC_559_COMPLEX__` La constante entera 1, destinada a indicar el cumplimiento de las especificaciones en el anexo G (aritmética compleja compatible con IEC 60559).
- `__STDC_LIB_EXT1__` La constante entera `201112L` , diseñada para indicar el soporte para las extensiones definidas en el anexo K (Interfaces de verificación de límites).
- `__STDC_NO_ATOMICS__` La constante entera 1, que pretende indicar que la

implementación no admite tipos atómicos (incluido el _Atomic tipo _Atomic) y el encabezado <stdatomic.h> .

- __STDC_NO_COMPLEX__ La constante entera 1, que pretende indicar que la implementación no admite tipos complejos o el encabezado <complex.h> .
- __STDC_NO_THREADS__ La constante entera 1, que pretende indicar que la implementación no admite el encabezado <threads.h> .
- __STDC_NO_VLA__ La constante entera 1, que pretende indicar que la implementación no admite matrices de longitud variable o tipos modificados de forma variable.

El encabezado incluye guardias

Casi todos los archivos de encabezado deben seguir el lenguaje incluido en la guarda :

mi-encabezado-archivo.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

Esto garantiza que cuando #include "my-header-file.h" en varios lugares, no obtenga declaraciones duplicadas de funciones, variables, etc. Imagine la siguiente jerarquía de archivos:

encabezado-1.h

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);
```

encabezado-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

C Principal

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Este código tiene un problema grave: el contenido detallado de MyStruct se define dos veces, lo que no está permitido. Esto podría resultar en un error de compilación que puede ser difícil de

rastrear, ya que un archivo de encabezado incluye otro. Si en cambio lo hiciste con guardias de cabecera:

encabezado-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

encabezado-2.h

```
#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

C Principal

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Esto luego se expandiría a:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H

typedef struct {
```

```
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
}
```

Cuando el compilador alcanza la segunda inclusión de **header-1.h** , `HEADER_1_H` ya estaba definido por la inclusión anterior. Ergo, se reduce a lo siguiente:

```
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

Y así no hay error de compilación.

Nota: Existen múltiples convenciones diferentes para nombrar los protectores de encabezado. A algunas personas les gusta `HEADER_2_H_` , algunas incluyen el nombre del proyecto como `MY_PROJECT_HEADER_2_H` . Lo importante es asegurarse de que la convención que sigue lo haga para que cada archivo en su proyecto tenga un protector de encabezado único.

Si los detalles de la estructura no se incluyeran en el encabezado, el tipo declarado sería incompleto o un tipo opaco . Tales tipos pueden ser útiles, ocultando los detalles de implementación de los usuarios de las funciones. Para muchos propósitos, el tipo de `FILE` en la biblioteca C estándar puede considerarse como un tipo opaco (aunque generalmente no es opaco, por lo que las implementaciones de macro de las funciones de E / S estándar pueden hacer uso de los elementos internos de la estructura). En ese caso, el `header-1.h` podría contener:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);
```

```
#endif
```

Tenga en cuenta que la estructura debe tener un nombre de etiqueta (aquí `MyStruct` , que está en el espacio de nombres de las etiquetas, separado del espacio de nombres de los identificadores ordinarios del nombre typedef `MyStruct`), y que se omite `{ … }` . Esto dice "hay un tipo de estructura `struct MyStruct` y hay un alias para `MyStruct` ".

En el archivo de implementación, los detalles de la estructura se pueden definir para completar el tipo:

```
struct MyStruct {
    …
};
```

Si está utilizando C11, puede repetir la `typedef struct MyStruct MyStruct;` Declaración sin causar un error de compilación, pero las versiones anteriores de C se quejarían. Por consiguiente, aún es mejor utilizar el lenguaje de inclusión de protección, aunque en este ejemplo, sería opcional si el código solo se compilara con compiladores que admitieran C11.

Muchos compiladores admiten la directiva `#pragma once` , que tiene los mismos resultados:

mi-encabezado-archivo.h

```
#pragma once

// Code for header file
```

Sin embargo, `#pragma once` no es parte del estándar C, por lo que el código es menos portátil si lo usas.

Unos pocos encabezados no usan el modismo de inclusión. Un ejemplo específico es el encabezado estándar `<assert.h>` . Puede incluirse varias veces en una sola unidad de traducción, y el efecto de hacerlo depende de si la macro `NDEBUG` se define cada vez que se incluye el encabezado. Ocasionalmente puede tener un requisito análogo; Tales casos serán pocos y distantes entre sí. Por lo general, sus encabezados deben estar protegidos por el lenguaje de inclusión de la guardia.

Implementación FOREACH

También podemos usar macros para hacer que el código sea más fácil de leer y escribir. Por ejemplo, podemos implementar macros para implementar el constructo `foreach` en C para algunas estructuras de datos como listas, colas, etc. vinculadas de manera simple y doble.

Aquí hay un pequeño ejemplo.

```
#include <stdio.h>
```

```
#include <stdlib.h>

struct LinkedListNode
{
    int data;
    struct LinkedListNode *next;
};

#define FOREACH_LIST(node, list) \
     for (node=list; node; node=node->next)

/* Usage */
int main(void)
{
    struct LinkedListNode *list, **plist = &list, *node;
    int i;

    for (i=0; i<10; i++)
    {
         *plist = malloc(sizeof(struct LinkedListNode));
         (*plist)->data = i;
         (*plist)->next = NULL;
         plist          = &(*plist)->next;
    }

    /* printing the elements here */
    FOREACH_LIST(node, list)
    {
        printf("%d\n", node->data);
    }
}
```

Puede crear una interfaz estándar para dichas estructuras de datos y escribir una implementación genérica de `FOREACH` como:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct CollectionItem_
{
    int data;
    struct CollectionItem_ *next;
} CollectionItem;

typedef struct Collection_
{
    /* interface functions */
    void* (*first)(void *coll);
    void* (*last) (void *coll);
    void* (*next) (void *coll, CollectionItem *currItem);

    CollectionItem *collectionHead;
    /* Other fields */
} Collection;

/* must implement */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
}
```

```
/* must implement */
void *last(void *coll)
{
    return NULL;
}

/* must implement */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
{
    CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
    item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
    CollectionItem **item = &coll->collectionHead;
    while(*item)
        item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
    Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
    nc->last  = last;
    nc->next  = next;
    return nc;
}

/* generic implementation */
#define FOREACH(node, collection)                      \
    for (node  = (collection)->first(collection);      \
         node != (collection)->last(collection);       \
         node  = (collection)->next(collection, node))

int main(void)
{
    Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
    {
         Add_Collection(coll, i);
    }

    /* printing the elements here */
    FOREACH(node, coll)
    {
        printf("%d\n", node->data);
    }
}
```

Para usar esta implementación genérica, simplemente implemente estas funciones para su estructura de datos.

```
1.  void* (*first)(void *coll);
2.  void* (*last) (void *coll);
3.  void* (*next) (void *coll, CollectionItem *currItem);
```

__cplusplus para usar C externos en el código C ++ compilado con C ++ - denominación de nombres

Hay ocasiones en que un archivo de inclusión tiene que generar una salida diferente del preprocesador dependiendo de si el compilador es un compilador de C o un compilador de C ++ debido a las diferencias de idioma.

Por ejemplo, una función u otra externa se define en un archivo fuente C, pero se usa en un archivo fuente C ++. Como C ++ utiliza la denominación de nombres (o la decoración de nombres) para generar nombres de función únicos basados en los tipos de argumento de función, una declaración de función C utilizada en un archivo fuente de C ++ causará errores de enlace. El compilador de C ++ modificará el nombre externo especificado para la salida del compilador usando las reglas de manipulación de nombres para C ++. El resultado son errores de enlace debido a elementos externos que no se encuentran cuando la salida del compilador de C ++ está vinculada con la salida del compilador de C.

Dado que los compiladores de C no modifican los nombres, los compiladores de C ++ lo hacen para todas las etiquetas externas (nombres de funciones o nombres de variables) generadas por el compilador de C ++, se introdujo una macro preprocesadora predefinida, `__cplusplus` , para permitir la detección del compilador.

Para evitar este problema de salida de compilador incompatible para nombres externos entre C y C ++, la macro `__cplusplus` se define en el preprocesador de C ++ y no se define en el preprocesador de C ++. Este nombre de macro se puede usar con la directiva `#ifdef` preprocesador condicional o `#if` con el operador `defined()` para indicar si un código fuente o un archivo de inclusión se está compilando como C ++ o C.

```
#ifdef __cplusplus
printf("C++\n");
#else
printf("C\n");
#endif
```

O podrías usar

```
#if defined(__cplusplus)
printf("C++\n");
#else
printf("C\n");
#endif
```

Para especificar el nombre correcto de la función de una función de un archivo fuente C

compilado con el compilador C que se está utilizando en un archivo fuente C ++, puede verificar la constante definida `__cplusplus` para provocar la `extern "C" { /* ... */ };` para ser usado para declarar externos de C cuando el archivo de encabezado se incluye en un archivo fuente de C ++. Sin embargo, cuando se compila con un compilador de C, el `extern "C" { */ ... */ };` no se utiliza Esta compilación condicional es necesaria porque `extern "C" { /* ... */ };` es válido en C ++ pero no en C.

```
#ifdef __cplusplus
// if we are being compiled with a C++ compiler then declare the
// following functions as C functions to prevent name mangling.
extern "C" {
#endif

// exported C function list.
int foo (void);

#ifdef __cplusplus
// if this is a C++ compiler, we need to close off the extern declaration.
};
#endif
```

Macros similares a funciones

Las macros similares a funciones son similares a las funciones en `inline` , son útiles en algunos casos, como el registro de depuración temporal:

```
#ifdef DEBUG
# define LOGFILENAME "/tmp/logfile.log"

# define LOG(str) do {                              \
  FILE *fp = fopen(LOGFILENAME, "a");              \
  if (fp) {                                         \
    fprintf(fp, "%s:%d %s\n", __FILE__, __LINE__, \
                 /* don't print null pointer */   \
                 str ?str :"<null>");             \
    fclose(fp);                                     \
  }                                                 \
  else {                                            \
    perror("Opening '" LOGFILENAME "' failed");   \
  }                                                 \
} while (0)
#else
  /* Make it a NOOP if DEBUG is not defined. */
# define LOG(LINE) (void)0
#endif


#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
        LOG("There are command line arguments");
    else
        LOG("No command line arguments");
    return 0;
```

```
}
```

Aquí, en ambos casos (con `DEBUG` o no), la llamada se comporta de la misma manera que una función con tipo de retorno `void` . Esto asegura que los condicionales `if/else` se interpreten como se espera.

En el caso de `DEBUG` , esto se implementa mediante una construcción `do { ... } while(0)` . En el otro caso, `(void)0` es una declaración sin efecto secundario que simplemente se ignora.

Una alternativa para este último sería

```
#define LOG(LINE) do { /* empty */ } while (0)
```

de modo que sea en todos los casos sintácticamente equivalente al primero.

Si usa GCC, también puede implementar una macro similar a una función que devuelva resultados utilizando una extensión GNU no estándar: expresiones de declaración . Por ejemplo:

```
#include <stdio.h>

#define POW(X, Y) \
({ \
        int i, r = 1; \
        for (i = 0; i < Y; ++i) \
                r *= X; \
        r; \ // returned value is result of last operation
})

int main(void)
{
        int result;

        result = POW(2, 3);
        printf("Result: %d\n", result);
}
```

Argumentos variables macro

C99

Macros con argumentos variadicos:

Supongamos que desea crear una macro de impresión para depurar su código, tomemos esta macro como ejemplo:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Algunos ejemplos de uso:

La función `somefunc()` devuelve -1 si falla y 0 si es `somefunc()` , y se llama desde muchos lugares diferentes dentro del código:

```
int retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}

/* some other code */

 retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

¿Qué sucede si cambia la implementación de `somefunc()` y ahora devuelve valores diferentes que coinciden con diferentes tipos de error posibles? Aún desea utilizar la macro de depuración e imprimir el valor de error.

```
debug_printf(retVal);      /* this would obviously fail */
debug_printf("%d",retVal); /* this would also fail */
```

Para resolver este problema se introdujo la macro `__VA_ARGS__` . Esta macro permite múltiples parámetros de X-macro:

Ejemplo:

```
 #define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
                               printf("\nError occurred in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Uso:

```
int retVal = somefunc();

debug_print("retVal of somefunc() is-> %d", retVal);
```

Esta macro le permite pasar varios parámetros e imprimirlos, pero ahora le prohíbe enviar parámetros.

```
debug_print("Hey");
```

Esto provocaría un error de sintaxis ya que la macro espera al menos un argumento más y el preprocesador no ignorará la falta de coma en la macro `debug_print()` . También `debug_print("Hey",);` generaría un error de sintaxis ya que no puede mantener vacío el argumento pasado a la macro.

Para resolver esto, se introdujo la macro `##__VA_ARGS__` , esta macro indica que si no existen argumentos variables, el preprocesador elimina la coma del código.

Ejemplo:

```
#define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
                              printf("\nError occured in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Uso:

```
debug_print("Ret val of somefunc()?");
debug_print("%d",somefunc());
```

Lea Preprocesador y Macros en línea: https://riptutorial.com/es/c/topic/447/preprocesador-y-macros

Capítulo 53: Punteros

Introducción

Un puntero es un tipo de variable que puede almacenar la dirección de otro objeto o una función.

Sintaxis

- <Tipo de datos> * <Nombre de variable>;
- int * ptrToInt;
- void * ptrToVoid; / * C89 + * /
- struct someStruct * ptrToStruct;
- int ** ptrToPtrToInt;
- int arr [longitud]; int * ptrToFirstElem = arr; / * Para <C99 'length' debe ser una constante de tiempo de compilación, para> = C11 podría ser una. * /
- int * arrayOfPtrsToInt [longitud]; / * Para <C99 'length' debe ser una constante de tiempo de compilación, para> = C11 podría ser una. * /

Observaciones

La posición del asterisco no afecta el significado de la definición:

```
/* The * operator binds to right and therefore these are all equivalent. */
int *i;
int * i;
int* i;
```

Sin embargo, al definir varios punteros a la vez, cada uno requiere su propio asterisco:

```
int *i, *j; /* i and j are both pointers */
int* i, j;  /* i is a pointer, but j is an int not a pointer variable */
```

También es posible una matriz de punteros, donde se da un asterisco antes del nombre de la variable de matriz:

```
int *foo[2]; /* foo is a array of pointers, can be accessed as *foo[0] and *foo[1] */
```

Examples

Errores comunes

El uso incorrecto de los punteros es a menudo una fuente de errores que pueden incluir errores de seguridad o fallas en los programas, la mayoría de las veces debido a fallas de segmentación.

No revisar fallas en la asignación

No se garantiza que la asignación de memoria tenga éxito y, en cambio, puede devolver un puntero `NULL` . El uso del valor devuelto, sin verificar si la asignación es exitosa, invoca un comportamiento indefinido . Por lo general, esto conduce a un choque, pero no hay garantía de que ocurra un choque, por lo que confiar también puede ocasionar problemas.

Por ejemplo, de manera insegura:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Camino seguro:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Usando números literales en lugar de sizeof al solicitar memoria

Para una configuración de compilador / máquina dada, los tipos tienen un tamaño conocido; sin embargo, no hay ninguna norma que defina que el tamaño de un tipo dado (que no sea `char`) sea el mismo para todas las configuraciones de compilador / máquina. Si el código usa 4 en lugar de `sizeof(int)` para la asignación de memoria, puede funcionar en la máquina original, pero el código no es necesariamente portátil a otras máquinas o compiladores. Los tamaños fijos para los tipos deben reemplazarse por `sizeof(that_type)` o `sizeof(*var_ptr_to_that_type)` .

Asignación no portátil:

```
int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Asignación portátil:

```
int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

O, mejor aún:

```
int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

Pérdidas de memoria

Si no se desasigna la memoria utilizando `free` cables `free` acumula una memoria no reutilizable que el programa ya no utiliza. Esto se llama una pérdida de memoria . La memoria pierde recursos de memoria de desperdicio y puede conducir a fallas en la asignación.

Errores logicos

Todas las asignaciones deben seguir el mismo patrón:

1. Asignación utilizando `malloc` (o `calloc`)
2. Uso para almacenar datos
3. Desasignación utilizando `free`

Si no se adhiere a este patrón, como el uso de la memoria después de una llamada para `free` (puntero colgante) o antes de una llamada a `malloc` (puntero salvaje), llamar dos veces `free` ("doble libre"), etc., generalmente causa un fallo de segmentación y los resultados en una caída del programa.

Estos errores pueden ser transitorios y difíciles de depurar; por ejemplo, el sistema operativo generalmente no recupera la memoria liberada, por lo que los punteros colgantes pueden persistir por un tiempo y parecer que funcionan.

En los sistemas donde funciona, Valgrind es una herramienta invaluable para identificar qué memoria se ha filtrado y dónde se asignó originalmente.

Creando punteros para apilar variables.

Crear un puntero no prolonga la vida de la variable a la que se apunta. Por ejemplo:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Aquí, `x` tiene *una duración de almacenamiento automático* (comúnmente conocida como asignación de *pila*). Debido a que está asignado en la pila, su vida útil es solo mientras `myFunction` esté ejecutando; después de que `myFunction` haya salido, la variable `x` se destruye. Esta función obtiene la dirección de `x` (usando `&x`) y la devuelve a la persona que llama, dejando a la persona que llama con un puntero a una variable que no existe. Intentar acceder a esta variable invocará un comportamiento indefinido .

La mayoría de los compiladores en realidad no borran un marco de pila después de que la función sale, por lo tanto, al eliminar la referencia al puntero devuelto a menudo se obtienen los datos esperados. Sin embargo, cuando se llama a otra función, la memoria a la que se apunta puede sobrescribirse, y parece que los datos a los que se apunta están dañados.

Para resolver esto, `malloc` un `malloc` el almacenamiento de la variable que se devolverá y devuelva un puntero al almacenamiento recién creado, o requiera que se pase un puntero válido

a la función en lugar de devolver uno, por ejemplo:

```
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int *x)
{
    /* NB: calling this function with an invalid or null pointer
       causes undefined behaviour. */

    *x = 10;
}

int main(void)
{
    {
        /* Use solution1() */

        int *foo = solution1();
        if (foo == NULL)
        {
            /* Something went wrong */
            return 1;
        }

        printf("The value set by solution1() is %i\n", *foo);
        /* Will output: "The value set by solution1() is 10" */

        free(foo);    /* Tidy up */
    }

    {
        /* Use solution2() */

        int bar;
        solution2(&bar);

        printf("The value set by solution2() is %i\n", bar);
        /* Will output: "The value set by solution2() is 10" */
    }

    return 0;
}
```

Incremento / decremento y desreferenciación

Si escribe `*p++` para incrementar lo que apunta `p` , está equivocado.

El incremento / decremento posterior se ejecuta antes de la desreferenciación. Por lo tanto, esta expresión incrementará el puntero `p` sí mismo y devolverá lo que fue apuntado por `p` antes de incrementarlo sin cambiarlo.

Debería escribir `(*p)++` para incrementar lo que apunta `p` .

Esta regla también se aplica al decremento posterior: `*p--` disminuirá el puntero `p` sí, no lo que se señala con `p` .

Desreferenciación de un puntero

```
int a = 1;
int *a_pointer = &a;
```

Para `a_pointer` referencia a `a_pointer` y cambiar el valor de a, usamos la siguiente operación

```
*a_pointer = 2;
```

Esto se puede verificar usando las siguientes declaraciones impresas.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

Sin embargo, uno podría confundirse con la desreferencia de un puntero `NULL` o no válido. Esta

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

Generalmente es un comportamiento indefinido . `p1` no puede ser desreferenciado porque apunta a una dirección `0xbad` que puede no ser una dirección válida. ¿Quién sabe qué hay ahí? Puede ser la memoria del sistema operativo, o la memoria de otro programa. El único código de tiempo como este que se usa es el desarrollo integrado, que almacena información particular en direcciones codificadas. `p2` no se puede anular debido a que es `NULL` , lo cual no es válido.

Desreferenciación de un puntero a una estructura

Digamos que tenemos la siguiente estructura:

```
struct MY_STRUCT
{
    int my_int;
    float my_float;
};
```

Podemos definir `MY_STRUCT` para omitir la palabra clave `struct` por lo que no tenemos que escribir `struct MY_STRUCT` cada vez que lo usamos. Esto, sin embargo, es opcional.

```
typedef struct MY_STRUCT MY_STRUCT;
```

Si entonces tenemos un puntero a una instancia de esta estructura

```
MY_STRUCT *instance;
```

Si esta declaración aparece en el ámbito del archivo, la `instance` se inicializará con un puntero nulo cuando se inicie el programa. Si esta declaración aparece dentro de una función, su valor no está definido. La variable debe inicializarse para que apunte a una variable `MY_STRUCT` válida, o al espacio asignado dinámicamente, antes de que se pueda eliminar la referencia. Por ejemplo:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

Cuando el puntero es válido, podemos anularlo para acceder a sus miembros usando una de dos notaciones diferentes:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

Si bien ambos métodos funcionan, es mejor utilizar el operador de flecha `->` lugar de la combinación de paréntesis, el operador de referencia `*` y el punto `.` Operador porque es más fácil de leer y entender, especialmente con usos anidados.

Otra diferencia importante se muestra a continuación:

```
MY_STRUCT copy = *instance;
copy.my_int    = 2;
```

En este caso, la `copy` contiene una copia del contenido de la `instance` . Cambiar `my_int` de `copy` no lo cambiará en la `instance` .

```
MY_STRUCT *ref = instance;
ref->my_int    = 2;
```

En este caso, `ref` es una referencia a la `instance` . Cambiar `my_int` usando la referencia lo cambiará en la `instance` .

Es una práctica común utilizar punteros a estructuras como parámetros en funciones, en lugar de las estructuras en sí. Usar las estructuras como parámetros de función podría hacer que la pila se desborde si la estructura es grande. El uso de un puntero a una estructura solo utiliza suficiente espacio de pila para el puntero, pero puede causar efectos secundarios si la función cambia la estructura que se pasa a la función.

Punteros de funcion

Los punteros también se pueden utilizar para señalar funciones.

Tomemos una función básica:

```
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
```

Ahora, definamos un puntero del tipo de esa función:

```
int (*my_pointer)(int, int);
```

Para crear una, solo usa esta plantilla:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

Entonces debemos asignar este puntero a la función:

```
my_pointer = &my_function;
```

Este puntero ahora se puede utilizar para llamar a la función:

```
/* Calling the pointed function */
int result = (*my_pointer)(4, 2);

...

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}
```

Aunque esta sintaxis parece más natural y coherente con los tipos básicos, los punteros de función de atribución y eliminación de referencias no requieren el uso de los operadores `&` y `*` . Así que el siguiente fragmento de código es igualmente válido:

```
/* Attribution without the & operator */
my_pointer = my_function;

/* Dereferencing without the * operator */
int result = my_pointer(4, 2);
```

Para aumentar la legibilidad de los punteros de función, se pueden usar typedefs.

```
typedef void (*Callback)(int a);
```

```
void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Otro truco de legibilidad es que el estándar C permite simplificar el puntero de una función en argumentos como el anterior (pero no en la declaración de variables) a algo que parece un prototipo de función; por lo tanto, lo siguiente se puede usar de manera equivalente para las definiciones y declaraciones de funciones:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

Ver también

Punteros a funciones

Inicializando los punteros

La inicialización del puntero es una buena manera de evitar los punteros salvajes. La inicialización es simple y no es diferente de la inicialización de una variable.

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

         /* NULL is a macro defined in stddef.h, stdio.h, stdlib.h, and string.h */

    ...
}
```

En la mayoría de los sistemas operativos, el uso involuntario de un puntero que se ha inicializado a `NULL` a menudo hace que el programa se bloquee de inmediato, lo que facilita la identificación de la causa del problema. El uso de un puntero no inicializado a menudo puede causar errores difíciles de diagnosticar.

Precaución:

El resultado de la anulación de la referencia de un puntero `NULL` no está definido, por lo *que no necesariamente causará un bloqueo* incluso si ese es el comportamiento natural del sistema operativo en el que se está ejecutando el programa. Las optimizaciones del compilador pueden

enmascarar el bloqueo, provocar que se produzca el bloqueo antes o después del punto en el código fuente en el que se produjo la desreferencia del puntero nulo, o causar que partes del código que contiene la desreferencia del puntero nulo se eliminen inesperadamente del programa. Las versiones de depuración no suelen mostrar estos comportamientos, pero esto no está garantizado por el estándar de idioma. También se permite otro comportamiento inesperado y / o indeseable.

Debido a que `NULL` nunca apunta a una variable, a la memoria asignada oa una función, es seguro usarlo como valor de guarda.

Precaución:

Normalmente `NULL` se define como `(void *)0` . Pero esto no implica que la dirección de memoria asignada sea `0x0` . Para obtener más información, consulte C-faq para punteros NULL

Tenga en cuenta que también puede inicializar los punteros para que contengan valores distintos de NULL.

```
int i1;

int main()
{
   int *p1 = &i1;
   const char *p2 = "A constant string to point to";
   float *p3 = malloc(10 * sizeof(float));
}
```

Dirección de Operador (&)

Para cualquier objeto (es decir, variable, matriz, unión, estructura, puntero o función), se puede utilizar el operador de dirección única para acceder a la dirección de ese objeto.

Suponer que

```
int i = 1;
int *p = NULL;
```

Entonces, una declaración `p = &i;` , copia la dirección de la variable `i` al puntero `p` .

Se expresa como `p` ***apunta a*** `i` .

`printf("%d\n", *p);` Imprime 1, que es el valor de `i` .

Aritmética de puntero

Por favor vea aquí: Aritmética de punteros

Void * punteros como argumentos y valores de retorno a las funciones estándar

K&R

`void*` es un tipo de captura de todos los punteros a los tipos de objetos. Un ejemplo de esto en uso es con la función `malloc` , que se declara como

```
void* malloc(size_t);
```

El tipo de retorno de puntero a vacío significa que es posible asignar el valor de retorno de `malloc` a un puntero a cualquier otro tipo de objeto:

```
int* vector = malloc(10 * sizeof *vector);
```

En general, se considera una buena práctica *no* lanzar explícitamente los valores dentro y fuera de los punteros de vacío. En el caso específico de `malloc()` esto se debe a que, con una conversión explícita, el compilador puede asumir, pero no advertir, un tipo de retorno incorrecto para `malloc()` , si olvida incluir `stdlib.h` . También se trata de utilizar el comportamiento correcto de los punteros de vacío para ajustarse mejor al principio DRY (no se repita); compare lo anterior con lo siguiente, en donde el siguiente código contiene varios lugares adicionales innecesarios donde un error tipográfico podría causar problemas:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

Del mismo modo, funciones como

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

sus argumentos se especifican como `void *` porque la dirección de cualquier objeto, independientemente del tipo, se puede pasar. Aquí también, una llamada no debe usar una conversión

```
unsigned char buffer[sizeof(int)];
int b = 67;
memcpy(buffer, &b, sizeof buffer);
```

Punteros const

Punteros individuales

- Puntero a un `int`

  El puntero puede apuntar a diferentes enteros y los `int` pueden cambiarse a través del puntero. Esta muestra de código asigna b para apuntar a `int b` luego cambia el valor de `b` a `100` .

  ```
  int b;
  int* p;
  p = &b;    /* OK */
  *p = 100;  /* OK */
  ```

- Puntero a una `const int`

  El puntero puede apuntar a diferentes enteros, pero el valor del `int` no se puede cambiar a través del puntero.

  ```
  int b;
  const int* p;
  p = &b;    /* OK */
  *p = 100;  /* Compiler Error */
  ```

- puntero de `const` a `int`

  El puntero solo puede apuntar a un `int` pero el valor del `int` puede cambiarse a través del puntero.

  ```
  int a, b;
  int* const p = &b; /* OK as initialisation, no assignment */
  *p = 100;  /* OK */
  p = &a;    /* Compiler Error */
  ```

- `const` puntero a `const int`

  El puntero solo puede apuntar a un `int` y el `int` no se puede cambiar a través del puntero.

  ```
  int a, b;
  const int* const p = &b; /* OK as initialisation, no assignment */
  p = &a;   /* Compiler Error */
  *p = 100; /* Compiler Error */
  ```

Puntero a puntero

- Puntero a un puntero a un `int`

  Este código asigna la dirección de `p1` al puntero doble `p` (que luego apunta a `int* p1` (que apunta a `int`)).

  Luego cambia `p1` para apuntar a `int a` . Luego cambia el valor de a para ser 100.

  ```
  void f1(void)
  {
    int a, b;
    int *p1;
    int **p;
    p1 = &b; /* OK */
    p = &p1; /* OK */
    *p = &a; /* OK */
    **p = 100; /* OK */
  }
  ```

- Puntero a puntero a una `const int`

```
 void f2(void)
{
  int b;
  const int *p1;
  const int **p;
  p = &p1; /* OK */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- Puntero a `const` puntero a un `int`

```
void f3(void)
{
  int b;
  int *p1;
  int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* OK */
}
```

- `const` puntero a puntero a `int`

```
void f4(void)
{
  int b;
  int *p1;
  int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p' */
  *p = &b; /* OK */
  **p = 100; /* OK */
}
```

- Puntero a `const` puntero a `const int`

```
void f5(void)
{
  int b;
  const int *p1;
  const int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- `const` puntero a puntero a `const int`

```
void f6(void)
{
  int b;
  const int *p1;
  const int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p' */
  *p = &b; /* OK */
```

```
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- `const` point to `const` point to `int`

```
void f7(void)
{
  int b;
  int *p1;
  int * const * const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p'  */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* OK */
}
```

Mismo Asterisco, Diferentes Significados

Premisa

Lo más confuso que rodea la sintaxis del puntero en C y C ++ es que en realidad hay dos significados diferentes que se aplican cuando el símbolo del puntero, el asterisco (`*`), se usa con una variable.

Ejemplo

En primer lugar, utiliza `*` para **declarar** una variable de puntero.

```
int i = 5;
/* 'p' is a pointer to an integer, initialized as NULL */
int *p = NULL;
/* '&i' evaluates into address of 'i', which then assigned to 'p' */
p = &i;
/* 'p' is now holding the address of 'i' */
```

Cuando no está declarando (o multiplicando), `*` se usa para **eliminar la referencia de** una variable de puntero:

```
*p = 123;
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

Cuando quieres que una variable de puntero existente contenga la dirección de otra variable, **no** usas `*` , pero hazlo así:

```
p = &another_variable;
```

Una confusión común entre los novatos en la programación C surge cuando declaran e inicializan una variable de puntero al mismo tiempo.

```
int *p = &i;
```

Dado que `int i = 5;` e `int i; i = 5;` dar el mismo resultado, algunos de ellos podrían pensar `int *p = &i;` e `int *p; *p = &i;` Da el mismo resultado también. El hecho es, no, `int *p; *p = &i;` intentará deferir un puntero **no inicializado** que dará como resultado UB. Nunca use `*` cuando no esté declarando ni desreferir un puntero.

Conclusión

El asterisco (`*`) tiene dos significados distintos dentro de C en relación con los punteros, dependiendo de dónde se use. Cuando se utiliza dentro de una **declaración de variable** , el valor en el lado derecho del lado igual debe ser un ***valor de puntero*** a una ***dirección*** en la memoria. Cuando se usa con una **variable** ya **declarada** , el asterisco ***eliminará*** la ***referencia*** del valor del puntero, lo seguirá hasta el lugar señalado en la memoria y permitirá que el valor almacenado allí sea asignado o recuperado.

Para llevar

Es importante tener en cuenta sus P y Q, por así decirlo, al tratar con punteros. Tenga en cuenta cuándo está usando el asterisco y qué significa cuando lo usa allí. Pasar por alto este pequeño detalle podría resultar en un comportamiento defectuoso y / o indefinido con el que realmente no quieres tener que lidiar.

Puntero a puntero

En C, un puntero puede referirse a otro puntero.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &pA;
  int*** pppA = &ppA;

  printf("%d", ***pppA); /* prints 42 */

  return EXIT_SUCCESS;
}
```

Pero, referencia y referencia directamente no está permitida.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &&A; /* Compilation error here! */
```

```
int*** pppA = &&&A;  /* Compilation error here! */

...
```

Introducción

Un puntero se declara de forma muy similar a cualquier otra variable, excepto que se coloca un asterisco (`*`) entre el tipo y el nombre de la variable para denotar que es un puntero.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid
object yet */
```

Para declarar dos variables de puntero del mismo tipo, en la misma declaración, use el símbolo de asterisco antes de cada identificador. Por ejemplo,

```
int *iptr1, *iptr2;
int *iptr3,  iptr4;  /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int
*/
```

El operador de dirección o referencia denotado por un signo (`&`) da la dirección de una variable dada que se puede colocar en un puntero del tipo apropiado.

```
int value = 1;
pointer = &value;
```

El operador de direccionamiento indirecto o de desreferencia indicado por un asterisco (`*`) obtiene el contenido de un objeto apuntado por un puntero.

```
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

Si el puntero apunta a una estructura o tipo de unión, puede anular la referencia y acceder a sus miembros directamente mediante el operador `->` :

```
SomeStruct *s = &someObject;
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

En C, un puntero es un tipo de valor distinto que se puede reasignar y, de lo contrario, se trata como una variable por derecho propio. Por ejemplo, el siguiente ejemplo imprime el valor del puntero (variable).

```
printf("Value of the pointer itself: %p\n", (void *)pointer);
/* Value of the pointer itself: 0x7ffcd41b06e4 */
/* This address will be different each time the program is executed */
```

Debido a que un puntero es una variable mutable, es posible que no apunte a un objeto válido, ya sea que se establezca en nulo

```
pointer = 0;      /* or alternatively */
pointer = NULL;
```

o simplemente al contener un patrón de bits arbitrario que no es una dirección válida. La última es una situación muy mala, ya que no se puede probar antes de que se elimine la referencia al puntero, solo hay una prueba para el caso de que el puntero sea nulo:

```
if (!pointer) exit(EXIT_FAILURE);
```

Un puntero solo puede ser referenciado si apunta a un objeto *válido* , de lo contrario el comportamiento es indefinido. Muchas implementaciones modernas pueden ayudarlo al generar algún tipo de error, como un error de segmentación y terminar la ejecución, pero otras pueden dejar su programa en un estado no válido.

El valor devuelto por el operador de referencia es un alias mutable a la variable original, por lo que se puede cambiar, modificando la variable original.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

Los punteros también son re-asignables. Esto significa que un puntero que apunta a un objeto se puede usar más tarde para apuntar a otro objeto del mismo tipo.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Como cualquier otra variable, los punteros tienen un tipo específico. No puede asignar la dirección de un `short int` a un puntero a un `long int` , por ejemplo. Este tipo de comportamiento se conoce como punning de tipo y está prohibido en C, aunque hay algunas excepciones.

Aunque el puntero debe ser de un tipo específico, la memoria asignada para cada tipo de puntero es igual a la memoria utilizada por el entorno para almacenar direcciones, en lugar del tamaño del tipo al que se apunta.

```
#include <stdio.h>

int main(void) {
    printf("Size of int pointer: %zu\n", sizeof (int*));      /* size 4 bytes */
    printf("Size of int variable: %zu\n", sizeof (int));      /* size 4 bytes */
    printf("Size of char pointer: %zu\n", sizeof (char*));    /* size 4 bytes */
    printf("Size of char variable: %zu\n", sizeof (char));    /* size 1 bytes */
    printf("Size of short pointer: %zu\n", sizeof (short*));  /* size 4 bytes */
    printf("Size of short variable: %zu\n", sizeof (short));  /* size 2 bytes */
    return 0;
}
```

(NB: si está utilizando Microsoft Visual Studio, que no es compatible con los estándares C99 o C11, debe usar `%Iu` [1] en lugar de `%zu` en el ejemplo anterior).

Tenga en cuenta que los resultados anteriores pueden variar de un entorno a otro en números, pero todos los entornos mostrarían los mismos tamaños para diferentes tipos de punteros.

Extracto basado en información de Cardiff University C Punteros Introducción

Punteros y matrices

Los punteros y las matrices están íntimamente conectados en C. Las matrices en C siempre se guardan en ubicaciones contiguas en la memoria. La aritmética de punteros siempre se escala según el tamaño del elemento apuntado. Entonces, si tenemos una matriz de tres dobles y un puntero a la base, `*ptr` refiere al primer doble, `*(ptr + 1)` al segundo, `*(ptr + 2)` al tercero. Una notación más conveniente es usar la notación de matriz `[]` .

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;

/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

Así que esencialmente ptr y el nombre de la matriz son intercambiables. Esta regla también significa que una matriz se desintegra en un puntero cuando se pasa a una subrutina.

```
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
   return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2])
}
```

Un puntero puede apuntar a cualquier elemento en una matriz, o al elemento más allá del último elemento. Sin embargo, es un error establecer un puntero a cualquier otro valor, incluido el elemento antes de la matriz. (La razón es que en las arquitecturas segmentadas la dirección antes del primer elemento puede cruzar un límite de segmento, el compilador garantiza que no suceda para el último elemento más uno).

Nota al pie 1: la información de formato de Microsoft se puede encontrar a través de `printf()` y la sintaxis de especificación de formato .

Comportamiento polimórfico con punteros vacíos.

La función de biblioteca estándar `qsort()` es un buen ejemplo de cómo se pueden usar los punteros de vacío para que una sola función funcione en una gran variedad de tipos diferentes.

```
void qsort (
    void *base,                          /* Array to be sorted */
    size_t num,                          /* Number of elements in array */
```

```
    size_t size,                                /* Size in bytes of each element */
    int (*compar)(const void *, const void *)); /* Comparison function for two elements */
```

La matriz que se va a ordenar se pasa como un puntero de vacío, por lo que se puede utilizar una matriz de cualquier tipo de elemento. Los siguientes dos argumentos le dicen a `qsort()` cuántos elementos debería esperar en la matriz, y qué tan grande, en bytes, es cada elemento.

El último argumento es un puntero de función a una función de comparación que a su vez toma dos punteros nulos. Al hacer que la persona que llama proporcione esta función, `qsort()` puede ordenar efectivamente los elementos de cualquier tipo.

Aquí hay un ejemplo de tal función de comparación, para comparar flotadores. Tenga en cuenta que cualquier función de comparación pasada a `qsort()` debe tener este tipo de firma. La forma en que se hace polimórfico es mediante la conversión de los argumentos de puntero nulo a los indicadores del tipo de elemento que deseamos comparar.

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

Como sabemos que qsort utilizará esta función para comparar flotantes, devolvemos los argumentos de los punteros nulos a los punteros flotantes antes de eliminarlas.

Ahora, el uso de la función polimórfica qsort en una matriz "array" con longitud "len" es muy simple:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Lea Punteros en línea: https://riptutorial.com/es/c/topic/1108/punteros

Capítulo 54: Punteros a funciones

Introducción

Los punteros de función son punteros que apuntan a funciones en lugar de tipos de datos. Se pueden usar para permitir la variabilidad en la función que se va a llamar, en tiempo de ejecución.

Sintaxis

- returnType (* nombre) (parámetros)
- typedef returnType (* nombre) (parámetros)
- typedef returnType Name (parámetros);
  Nombre nombre;
- typedef returnType Name (parámetros);
  typedef Nombre * NamePtr;

Examples

Asignar un puntero de función

```
#include <stdio.h>

/* increment: take number, increment it by one, and return it */
int increment(int i)
{
    printf("increment %d by 1\n", i);
    return i + 1;
}

/* decrement: take number, decrement it by one, and return it */
int decrement(int i)
{
    printf("decrement %d by 1\n", i);
    return i - 1;
}

int main(void)
{
    int num = 0;          /* declare number to increment */
    int (*fp)(int);       /* declare a function pointer */

    fp = &increment;      /* set function pointer to increment function */
    num = (*fp)(num);     /* increment num */
    num = (*fp)(num);     /* increment num a second time */

    fp = &decrement;      /* set function pointer to decrement function */
    num = (*fp)(num);     /* decrement num */
    printf("num is now: %d\n", num);
```

```
    return 0;
}
```

Devolviendo punteros a funciones desde una función

```
#include <stdio.h>

enum Op
{
  ADD = '+',
  SUB = '-',
};


/* add: add a and b, return result */
int add(int a, int b)
{
    return a + b;
}

/* sub: subtract b from a, return result */
int sub(int a, int b)
{
    return a - b;
}

/* getmath: return the appropriate math function */
int (*getmath(enum Op op))(int,int)
{
    switch (op)
    {
        case ADD:
            return &add;
        case SUB:
            return &sub;
        default:
            return NULL;
    }
}

int main(void)
{
    int a, b, c;
    int (*fp)(int,int);

    fp = getmath(ADD);

    a = 1, b = 2;
    c = (*fp)(a, b);
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Mejores prácticas

Usando typedef

Puede ser útil usar un `typedef` lugar de declarar el puntero a la función cada vez que esté a mano.

La sintaxis para declarar un `typedef` para un puntero de función es:

```
typedef returnType (*name)(parameters);
```

Ejemplo:

Dice que tenemos una función, una `sort` , que espera un puntero de función a una función de `compare` tal que:

> compare: una función de comparación para dos elementos que se debe suministrar a una función de clasificación.
>
> se espera que "compare" devuelva 0 si los dos elementos se consideran iguales, un valor positivo si el primer elemento pasado es "más grande" en cierto sentido que el último elemento y, de lo contrario, la función devuelve un valor negativo (lo que significa que el primer elemento es "menor" que este último).

Sin un `typedef` , `typedef` un puntero a una función como un argumento a una función de la siguiente manera:

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {
    /* inside of this block, the function is named "compare" */
}
```

Con un `typedef` , escribiríamos:

```
typedef int (*compare_func)(const void *, const void *);
```

y luego podríamos cambiar la firma de función de `sort` a:

```
void sort(compare_func func) {
    /* In this block the function is named "func" */
}
```

Ambas definiciones de `sort` aceptarían cualquier función de la forma.

```
int compare(const void *arg1, const void *arg2) {
    /* Note that the variable names do not have to be "elem1" and "elem2" */
}
```

Los punteros de función son el único lugar donde debe incluir la propiedad de puntero del tipo, por ejemplo, no intente definir tipos como `typedef struct something_struct *something_type` . Esto se aplica incluso para una estructura con los miembros que no se supone que se accede directamente por las personas que llaman API, por ejemplo, el stdio.h `FILE` tipo (que ya que ahora se dará cuenta de que no es un puntero).

Tomando punteros de contexto.

Un puntero de función casi siempre debe tomar un void * proporcionado por el usuario como un puntero de contexto.

Ejemplo

```
/* function minimiser, details unimportant */
double findminimum( double (*fptr)(double x, double y, void *ctx), void *ctx)
{
    ...
    /* repeatedly make calls like this */
    temp = (*fptr)(testx, testy, ctx);
}

/* the function we are minimising, sums two cubics */
double *cubics(double x, double y, void *ctx)
{
    double *coeffsx = ctx;
    double *coeffsy = coeffx + 4;

    return coeffsx[0] * x * x * x + coeffsx[1] * x * x + coeffsx[2] * x + coeffsx[3] +
           coeffsy[0] * y * y * y + coeffsy[1] * y * y + coeffsy[2] * y + coeffsy[3];

}

void caller()
{
    /* context, the coefficients of the cubics */
    double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    double min;

    min = findminimum(cubics, coeffs);
}
```

El uso del puntero de contexto significa que los parámetros adicionales no necesitan ser codificados en la función apuntada, o requieren el uso de elementos globales.

La función de biblioteca `qsort()` no sigue esta regla, y a menudo se puede escapar sin contexto para funciones de comparación triviales. Pero para algo más complicado, el puntero de contexto se vuelve esencial.

Ver también

Punteros funciones

Introducción

Al igual que `char` e `int` , una función es una característica fundamental de C. Como tal, puede declarar un puntero a uno: lo que significa que puede pasar *qué función llamar* a otra función para

ayudarlo a hacer su trabajo. Por ejemplo, si tenía una función `graph()` que mostraba un gráfico, podría pasar *qué función graficar* en `graph()` .

```
// A couple of external definitions to make the example clearer
extern unsigned int screenWidth;
extern void plotXY(double x, double y);

// The graph() function.
// Pass in the bounds: the minimum and maximum X and Y that should be plotted.
// Also pass in the actual function to plot.
void graph(double minX, double minY,
           double maxX, double maxY,
           ???? *fn) {             // See below for syntax

    double stepX = (maxX - minX) / screenWidth;
    for (double x=minX; x<maxX; x+=stepX) {

        double y = fn(x);          // Get y for this x by calling passed-in fn()

        if (minY<=y && y<maxY) {
            plotXY(x, y);          // Plot calculated point
        } // if
    } for
} // graph(minX, minY, maxX, maxY, fn)
```

Uso

Por lo tanto, el código anterior representará gráficamente cualquier función que le haya asignado, siempre y cuando esa función cumpla con ciertos criterios: a saber, que pase un `double` y obtenga un `double` . Hay muchas funciones como esa - `sin()` , `cos()` , `tan()` , `exp()` etc. - ¡pero hay muchas que no lo son, como la `graph()` sí misma!

Sintaxis

Entonces, ¿cómo especifica qué funciones puede pasar a `graph()` y cuáles no? La forma convencional es mediante el uso de una sintaxis que puede no ser fácil de leer o entender:

```
double (*fn)(double); // fn is a pointer-to-function that takes a double and returns one
```

El problema anterior es que hay dos cosas que se deben definir al mismo tiempo: la estructura de la función y el hecho de que es un puntero. Entonces, ¡divide las dos definiciones! Pero al usar `typedef` , se puede lograr una mejor sintaxis (más fácil de leer y entender).

Mnemónicos para escribir punteros a funciones.

Todas las funciones de C están en punteros de actualidad a un punto en la memoria del programa donde existe algún código. El uso principal de un puntero de función es proporcionar una "devolución de llamada" a otras funciones (o para simular clases y objetos).

La sintaxis de una función, tal como se define más adelante en esta página es:

returnType (* nombre) (parámetros)

Un mnemónico para escribir una definición de puntero de función es el siguiente procedimiento:

1. Comience escribiendo una declaración de función normal: `returnType name(parameters)`
2. Ajuste el nombre de la función con la sintaxis del puntero: `returnType (*name)(parameters)`

Lo esencial

Al igual que puede tener un puntero a un **int** , **char** , **float** , **array / string** , **struct** , etc., puede tener un puntero a una función.

La declaración del puntero toma el *valor de retorno de la función* , el *nombre de la función* y el *tipo de argumentos / parámetros que recibe* .

Digamos que tienes la siguiente función declarada e inicializada:

```
int addInt(int n, int m){
    return n+m;
}
```

Puede declarar e inicializar un puntero a esta función:

```
int (*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional
```

Si tienes una función de vacío, podría verse así:

```
void Print(void){
    printf("look ma' - no hands, only pointers!\n");
}
```

Entonces declarar el puntero a él sería:

```
void (*functionPtrPrint)(void) = Print;
```

Acceder a la función en sí requeriría desreferenciar el puntero:

```
sum = (*functionPtrAdd)(2, 3); //will assign 5 to sum
(*functionPtrPrint)(); //will print the text in Print function
```

Como se ve en los ejemplos más avanzados de este documento, declarar un puntero a una función podría ensuciarse si la función pasa más que unos pocos parámetros. Si tiene algunos punteros a funciones que tienen una "estructura" idéntica (el mismo tipo de valor de retorno y el mismo tipo de parámetros) es mejor usar el comando **typedef** para ahorrar algo de escritura y para aclarar el código:

```
typedef int (*ptrInt)(int, int);
```

```
int Add(int i, int j){
    return i+j;
}

int Multiply(int i, int j){
    return i*j;
}

int main()
{
    ptrInt ptr1 = Add;
    ptrInt ptr2 = Multiply;

    printf("%d\n", (*ptr1)(2,3)); //will print 5
    printf("%d\n", (*ptr2)(2,3)); //will print 6
    return 0;
}
```

También puede crear una **matriz de punteros de función** . Si todos los punteros son de la misma "estructura":

```
int (*array[2]) (int x, int y); // can hold 2 function pointers
array[0] = Add;
array[1] = Multiply;
```

Puedes aprender más aquí y aquí .

También es posible definir una matriz de punteros de función de diferentes tipos, aunque eso requeriría la conversión cuando quiera que quiera acceder a la función específica. Puedes aprender más aquí .

Lea Punteros a funciones en línea: https://riptutorial.com/es/c/topic/250/punteros-a-funciones

Capítulo 55: Puntos de secuencia

Observaciones

Norma Internacional ISO / IEC 9899: 201x Lenguajes de programación - C

> El acceso a un objeto volátil, la modificación de un objeto, la modificación de un archivo o la llamada a una función que realiza cualquiera de estas operaciones son todos *efectos secundarios* , que son cambios en el estado del entorno de ejecución.
>
> La presencia de un *punto de secuencia* entre la evaluación de las expresiones A y B implica que cada cálculo de valor y efecto secundario asociado con A se secuencia antes de cada cálculo de valor y efecto secundario asociado con B.

Aquí está la lista completa de puntos de secuencia del Anexo C del borrador de publicación en línea 2011 del estándar en lenguaje C:

> **Puntos de secuencia**
>
> 1 Los siguientes son los puntos de secuencia descritos en 5.1.2.3:
>
> - Entre las evaluaciones del designador de función y los argumentos reales en una llamada de función y la llamada real. (6.5.2.2).
> - Entre las evaluaciones del primer y segundo operandos de los siguientes operadores: lógico AND `&&` (6.5.13); OR lógico `||` (6.5.14); coma `,` (6.5.17).
> - ¿Entre las evaluaciones del primer operando del condicional `? :` operador y cualquiera de los segundos y terceros operandos se evalúa (6.5.15).
> - El fin de un declarador completo: declaradores (6.7.6);
> - Entre la evaluación de una expresión completa y la siguiente expresión completa a evaluar. Las siguientes son expresiones completas: un inicializador que no forma parte de un literal compuesto (6.7.9); la expresión en una declaración de expresión (6.8.3); la expresión de control de una declaración de selección (`if` o `switch`) (6.8.4); la expresión de control de una sentencia `while` o `do` (6.8.5); cada una de las expresiones (opcionales) de una declaración `for` (6.8.5.3); la expresión (opcional) en una declaración de `return` (6.8.6.4).
> - Inmediatamente antes de que una función de biblioteca regrese (7.1.4).
> - Después de las acciones asociadas con cada especificador de conversión de función de entrada / salida con formato (7.21.6, 7.29.2).
> - Inmediatamente antes e inmediatamente después de cada llamada a una función de comparación, y también entre cualquier llamada a una función de comparación y cualquier movimiento de los objetos pasados como argumentos a esa llamada (7.22.5).

Examples

Expresiones secuenciadas

Se *secuencian las* siguientes expresiones:

```
a && b
a || b
a , b
a ? b : c
for ( a ; b ; c ) { ... }
```

En todos los casos, la expresión `a` se evalúa completamente y *todos los efectos secundarios se aplican* antes de evaluar `b` o `c` . En el cuarto caso, solo uno de `b` o `c` será evaluado. En el último caso, `b` se evalúa completamente y todos los efectos secundarios se aplican antes de evaluar `c` .

En todos los casos, la evaluación de la expresión `a` se *secuencia antes de* las evaluaciones de `b` o `c` (alternativamente, las evaluaciones de `b` y `c` se *secuencian después de* la evaluación de `a`).

Así, expresiones como

```
x++ && x++
x++ ? x++ : y++
(x = f()) && x != 0
for ( x = 0; x < 10; x++ ) { ... }
y = (x++, x++);
```

Tener un comportamiento bien definido.

Expresiones sin secuencia

C11

Las siguientes expresiones no tienen *secuencia* :

```
a + b;
a - b;
a * b;
a / b;
a % b;
a & b;
a | b;
```

En los ejemplos anteriores, la expresión `a` puede evaluarse antes o después de la expresión `b` , `b` puede evaluarse antes de `a` , o incluso pueden entremezclarse si corresponden a varias instrucciones.

Una regla similar es válida para las llamadas a funciones:

```
f(a, b);
```

Aquí no sólo `a` y `b` son unsequenced (es decir, la `,` operador en una llamada de función *no* produce un punto de secuencia), sino también `f` , la expresión que determina la función que se va

a llamar.

Los efectos secundarios pueden aplicarse inmediatamente después de la evaluación o diferirse hasta un punto posterior.

Expresiones como

```
x++ & x++;
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */
x++ * x++;
a[i] = i++;
```

o

```
x++ & x;
f(x++, x);
x++ * x;
a[i++] = i;
```

producirá *un comportamiento indefinido* porque

- una modificación de un objeto y cualquier otro acceso a él debe ser secuenciado
- el orden de evaluación y el orden en que se aplican los *efectos secundarios* [1] no se especifican.

---

1 Cualquier cambio en el estado del entorno de ejecución.

Expresiones en secuencia indeterminada

Las llamadas de función como `f(a)` siempre implican un punto de secuencia entre la evaluación de los argumentos y el designador (aquí `f` y `a`) y la llamada real. Si dos de estas llamadas no se secuencian, las dos llamadas de función se secuencian de forma indeterminada, es decir, una se ejecuta antes que la otra, y el orden no se especifica.

```
unsigned counter = 0;

unsingned account(void) {
   return counter++;
}

int main(void) {
   printf("the order is %u %u\n", account(), account());
}
```

Esta modificación doble de `counter` implícita durante la evaluación de los argumentos de `printf` es válida, simplemente no sabemos cuál de las llamadas viene primero. Como el orden no está especificado, puede variar y no se puede depender de él. Así que la impresión podría ser:

> el orden es 0 1

o

el orden es 1 0

La declaración análoga a la anterior sin llamada de función intermedia

```
printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

tiene un comportamiento indefinido porque no hay un punto de secuencia entre las dos modificaciones del `counter` .

Lea Puntos de secuencia en línea: https://riptutorial.com/es/c/topic/1275/puntos-de-secuencia

Capítulo 56: Restricciones

Observaciones

Las restricciones son un término usado en todas las especificaciones C existentes (recientemente ISO-IEC 9899-2011). Son una de las tres partes del lenguaje descritas en la cláusula 6 de la norma (junto con la sintaxis y la semántica).

ISO-IEC 9899-2011 define una restricción como:

> Restricción, ya sea sintáctica o semántica, mediante la cual se debe interpretar la exposición de elementos del lenguaje.

(Tenga en cuenta también que, en términos del estándar C, una "restricción de tiempo de ejecución" no es un tipo de restricción y tiene reglas muy diferentes).

En otras palabras, una restricción describe una regla del lenguaje que haría que un programa por lo demás sintácticamente válido fuera ilegal. En este sentido, las restricciones son algo así como un comportamiento indefinido, cualquier programa que no las siga no está definido en términos del lenguaje C.

Las restricciones, por otro lado, tienen una diferencia muy significativa con respecto a los comportamientos indefinidos. Es decir, se requiere una implementación para proporcionar un mensaje de diagnóstico durante la fase de traducción (parte de la compilación) si se infringe una restricción, este mensaje puede ser una advertencia o puede detener la compilación.

Examples

Duplicar nombres de variables en el mismo ámbito.

Un ejemplo de una restricción expresada en el estándar C es tener dos variables del mismo nombre declaradas en un alcance [1)] , por ejemplo:

```
void foo(int bar)
{
    int var;
    double var;
}
```

Este código infringe la restricción y debe producir un mensaje de diagnóstico en el momento de la compilación. Esto es muy útil en comparación con un comportamiento indefinido, ya que el desarrollador será informado del problema antes de que se ejecute el programa, haciendo potencialmente cualquier cosa.

Las restricciones, por lo tanto, tienden a ser errores que son fácilmente detectables en el momento de la compilación, como por ejemplo, los problemas que resultan en un comportamiento indefinido pero que serían difíciles o imposibles de detectar en el momento de la compilación no

son restricciones.

---

1) redacción exacta:

C99

Si un identificador no tiene enlace, no habrá más de una declaración del identificador (en un declarador o especificador de tipo) con el mismo alcance y en el mismo espacio de nombre, excepto para las etiquetas como se especifica en 6.7.2.3.

Operadores aritméticos únicos

Los operadores unarios `+` y `-` solo se pueden usar en tipos aritméticos, por lo tanto, si por ejemplo, uno intenta usarlos en una estructura, el programa producirá un diagnóstico, por ejemplo:

```
struct foo
{
    bool bar;
};

void baz(void)
{
   struct foo testStruct;
   -testStruct; /* This breaks the constraint so must produce a diagnostic */
}
```

Lea Restricciones en línea: https://riptutorial.com/es/c/topic/7397/restricciones

Capítulo 57: Saltar declaraciones

Sintaxis

- devuelve val; / * Vuelve de la función actual. val puede ser un valor de cualquier tipo que se convierte al tipo de retorno de la función. * /
- regreso; / * Regresa de la función void actual. * /
- descanso; / * Salta incondicionalmente más allá del final ("rompe") de una instrucción de iteración (bucle) o de la instrucción de conmutación más interna. * /
- continuar; / * Salta incondicionalmente al principio de una instrucción de iteración (bucle). * /
- goto LBL; / * Salta para etiquetar LBL. * /
- LBL: *declaración* / * cualquier declaración en la misma función. * /

Observaciones

Estos son los saltos que se integran en C mediante palabras clave.

C también tiene otra construcción de *salto* , *salto de longitud* , que se especifica con un tipo de datos, `jmp_buf` y llamadas a la biblioteca de C, `setjmp` y `longjmp` .

Ver también

Iteraciones / bucles de repetición: for, while, do-while

Examples

Usando goto para saltar fuera de los bucles anidados

Saltar fuera de los bucles anidados normalmente requeriría el uso de una variable booleana con una comprobación de esta variable en los bucles. Suponiendo que estamos iterando sobre `i` y `j` , podría verse así

```
size_t i,j;
for (i = 0; i < myValue && !breakout_condition; ++i) {
    for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
        ... /* Do something, maybe modifying breakout_condition */
            /* When breakout_condition == true the loops end */
     }
}
```

Pero el lenguaje C ofrece la cláusula `goto` , que puede ser útil en este caso. Al usarlo con una etiqueta declarada después de los bucles, podemos romper fácilmente los bucles.

```
size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
```

```
        ...
        if(breakout_condition)
          goto final;
    }
}
final:
```

Sin embargo, a menudo, cuando surge esta necesidad, una `return` podría ser mejor utilizada en su lugar. Este constructo también se considera "no estructurado" en la teoría de la programación estructural.

Otra situación en la que `goto` podría ser útil es para saltar a un manejador de errores:

```
ptr = malloc(N *  x);
if(!ptr)
  goto out_of_memory;

/* normal processing */
free(ptr);
return SUCCESS;

out_of_memory:
 free(ptr); /* harmless, and necessary if we have further errors */
 return FAILURE;
```

El uso de `goto` mantiene el flujo de errores separado del flujo de control del programa normal. Sin embargo, también se considera "no estructurado" en el sentido técnico.

Usando el retorno

Devolviendo un valor

Un caso comúnmente usado: regresar de `main()`

```
#include <stdlib.h> /* for EXIT_xxx macros */

int main(int argc, char ** argv)
{
  if (2 < argc)
  {
    return EXIT_FAILURE; /* The code expects one argument:
                            leave immediately skipping the rest of the function's code */
  }

  /* Do stuff. */

  return EXIT_SUCCESS;
}
```

Notas adicionales:

1. Para una función que tenga un tipo de retorno como `void` (sin incluir los tipos `void *` o relacionados), la declaración de `return` no debe tener ninguna expresión asociada; es decir,

la única declaración de devolución permitida sería la `return;` .

2. Para una función que tiene un tipo de retorno no `void` , la declaración de `return` no aparecerá sin una expresión.

3. Para `main()` (y solo para `main()`), no se requiere una declaración de `return` *explícita* (en C99 o posterior). Si la ejecución alcanza la terminación `}` , se devuelve un valor implícito de `0` . Algunas personas piensan que omitir este `return` es una mala práctica; otros sugieren activamente omitirlo.

Devolviendo nada

Volviendo de una función de `void`

```
void log(const char * message_to_log)
{
  if (NULL == message_to_log)
  {
    return; /* Nothing to log, go home NOW, skip the logging. */
  }

  fprintf(stderr, "%s:%d %s\n", __FILE__, _LINE__, message_to_log);

  return; /* Optional, as this function does not return a value. */
}
```

Usando break y continue

Inmediatamente `continue` leyendo en una entrada no válida o `break` en la solicitud del usuario o al final del archivo:

```
#include <stdlib.h> /* for EXIT_xxx macros */
#include <stdio.h>  /* for printf() and getchar() */
#include <ctype.h> /* for isdigit() */

void flush_input_stream(FILE * fp);


int main(void)
{
  int sum = 0;
  printf("Enter digits to be summed up or 0 to exit:\n");

  do
  {
    int c = getchar();
    if (EOF == c)
    {
      printf("Read 'end-of-file', exiting!\n");

      break;
    }

    if ('\n' != c)
```

```
    {
      flush_input_stream(stdin);
    }

    if (!isdigit(c))
    {
      printf("%c is not a digit! Start over!\n", c);

      continue;
    }

    if ('0' == c)
    {
      printf("Exit requested.\n");

      break;
    }

    sum += c - '0';

    printf("The current sum is %d.\n", sum);
  } while (1);

  return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
  size_t i = 0;
  int c;
  while ((c = fgetc(fp)) != '\n' && c != EOF) /* Pull all until and including the next new-
line. */
  {
    ++i;
  }

  if (0 != i)
  {
    fprintf(stderr, "Flushed %zu characters from input.\n", i);
  }
}
```

Lea Saltar declaraciones en línea: https://riptutorial.com/es/c/topic/5568/saltar-declaraciones

Capítulo 58: Secuencia de caracteres de múltiples caracteres

Observaciones

No todos los preprocesadores admiten el procesamiento de secuencias trigráficas. Algunos compiladores dan una opción extra o un interruptor para procesarlos. Otros utilizan un programa separado para convertir trigraphs.

El compilador GCC no los reconoce a menos que usted lo solicite explícitamente (use `-trigraphs` para habilitarlos; use `-Wtrigraphs` , parte de `-Wall` , para obtener advertencias sobre los trigraphs).

Como la mayoría de las plataformas en uso hoy en día son compatibles con la gama completa de caracteres únicos utilizados en C, se prefieren los dígrafos en lugar de los trigrafos, pero generalmente se desaconseja el uso de cualquier secuencia de caracteres de múltiples caracteres.

Además, tenga cuidado con el uso accidental del trígrafo (por ejemplo, `puts("What happened??!!");`

Examples

Trigrafos

Los símbolos `[ ] { } ^ \ | ~ #` se utilizan con frecuencia en los programas de C, pero a fines de la década de 1980, hubo conjuntos de códigos en uso (variantes ISO 646, por ejemplo, en países escandinavos) donde las posiciones de caracteres ASCII para estos se usaron para caracteres de variantes de idiomas nacionales (por ejemplo, `£` para `#` en el Reino Unido; `Æ Å æ å ø Ø` para `{ } { } | \` en Dinamarca; no había `~` en EBCDIC). Esto significaba que era difícil escribir código C en las máquinas que usaban estos conjuntos.

Para resolver este problema, el estándar C sugirió el uso de combinaciones de tres caracteres para producir un solo carácter llamado trigraph. Un trigraph es una secuencia de tres caracteres, los dos primeros de los cuales son signos de interrogación.

El siguiente es un ejemplo simple que usa secuencias trigráficas en lugar de `#` , `{` y `}` :

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!\n");
??>
```

Esto lo cambiará el preprocesador de C reemplazando los trigrafos con sus equivalentes de un solo carácter como si el código hubiera sido escrito:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
}
```

Trígrafo	Equivalente
?? =	#
?? /	\
??	^
??	El
??)	]
??		
?? <	{
??>	}
?? -	~

Tenga en cuenta que los trigraphs son problemáticos porque, por ejemplo, `??/` es una barra invertida y puede afectar el significado de las líneas de continuación en los comentarios, y deben ser reconocidos dentro de cadenas y literales de caracteres (por ejemplo, `'??/??/'` es una sola personaje, una barra invertida).

Digraphs

C99

En 1994 se proporcionaron alternativas más legibles a cinco de los trigrafos. Estos utilizan solo dos caracteres y son conocidos como digraphs. A diferencia de los trigraphs, los digraphs son tokens. Si aparece un dígrafo en otro token (por ejemplo, literales de cadenas o constantes de caracteres), no se tratará como un dígrafo, sino que permanecerá como está.

A continuación se muestra la diferencia antes y después de procesar la secuencia de dígrafos.

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!\n"); /* Note that the string contains a digraph */
%>
```

Que será tratado igual que:

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!\n"); /* Note the unchanged digraph within the string. */
}
```

Dígrafo	Equivalente
<:	El
:>	]
<%	{
%>	}
%:	#

Lea Secuencia de caracteres de múltiples caracteres en línea:
https://riptutorial.com/es/c/topic/7111/secuencia-de-caracteres-de-multiples-caracteres

Capítulo 59: Selección genérica

Sintaxis

- _Generic (asignación-expresión, generic-assoc-list)

Parámetros

Parámetro	Detalles
lista-assoc-genérica	asociación genérica **O** asociación-genérica-asociación, asociación-genérica
asociación genérica	nombre-tipo: expresión-asignación **O por** defecto: expresión-asignación

Observaciones

1. Todos los calificadores de tipo se eliminarán durante la evaluación de la expresión primaria `_Generic` .
2. `_Generic` expresión primaria `_Generic` se evalúa en la fase de traducción 7 . Así que las fases como la concatenación de cuerdas se han terminado antes de su evaluación.

Examples

Compruebe si una variable es de un determinado tipo calificado

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x),  \
        const int *: "a const int",     \
        int *:       "a non-const int", \
        default:     "of other type")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i is %s\n", is_const_int(i));
    printf("j is %s\n", is_const_int(j));
    printf("k is %s\n", is_const_int(k));
}
```

Salida:

```
i is a const int
j is a non-const int
k is of other type
```

Sin embargo, si la macro genérica de tipo se implementa así:

```
#define is_const_int(x) _Generic((x), \
        const int: "a const int",     \
        int:       "a non-const int", \
        default:   "of other type")
```

La salida es:

```
i is a non-const int
j is a non-const int
k is of other type
```

Esto se debe a que todos los calificadores de tipo se eliminan para la evaluación de la expresión de control de una expresión primaria `_Generic` .

Tipo genérico de impresión macro

```
#include <stdio.h>

void print_int(int x) { printf("int: %d\n", x); }
void print_dbl(double x) { printf("double: %g\n", x); }
void print_default() { puts("unknown argument"); }

#define print(X) _Generic((X), \
        int: print_int, \
        double: print_dbl, \
        default: print_default)(X)

int main(void) {
    print(42);
    print(3.14);
    print("hello, world");
}
```

Salida:

```
int: 42
double: 3.14
unknown argument
```

Tenga en cuenta que si el tipo no es `int` ni `double` , se generará una advertencia. Para eliminar la advertencia, puede agregar ese tipo a la macro de `print(X)` .

Selección genérica basada en múltiples argumentos.

Si se desea una selección en múltiples argumentos para un tipo de expresión genérica, y todos

los tipos en cuestión son tipos aritméticos, una manera fácil de evitar las expresiones `_Generic` anidadas es usar la adición de los parámetros en la expresión de control:

```
int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y),                \
                           int:      max_int,      \
                           unsigned: max_unsigned, \
                           default:  max_double)   \
                    ((X), (Y))
```

Aquí, la expresión de control `(X)+(Y)` solo se inspecciona según su tipo y no se evalúa. Las conversiones habituales para los operandos aritméticos se realizan para determinar el tipo seleccionado.

Para una situación más compleja, se puede realizar una selección basada en más de un argumento para el operador, al anidarlos juntos.

Este ejemplo selecciona entre cuatro funciones implementadas externamente, que toman combinaciones de dos int y / o argumentos de cadena, y devuelven su suma.

```
int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char*  a, int b );
int AddStrStr(const char*  a, const char*  b);

#define AddStr(y)                            \
   _Generic((y),         int: AddStrInt,     \
                        char*: AddStrStr,    \
                  const char*: AddStrStr )

#define AddInt(y)                            \
   _Generic((y),          int: AddIntInt,    \
                        char*: AddIntStr,    \
                  const char*: AddIntStr )

#define Add(x, y)                            \
   _Generic((x) ,        int: AddInt(y) ,    \
                       char*: AddStr(y) ,    \
                 const char*: AddStr(y))     \
                         ((x), (y))

int main( void )
{
    int result = 0;
    result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

    const int a = -123;
    char b[] = "4321";
    result = Add( a , b );

    int c = 1;
```

```
    const char d[] = "0";
    result = Add( d , ++c );
}
```

Aunque parece que el argumento `y` se evalúa más de una vez, no es [1] . Ambos argumentos se evalúan solo una vez, al final de la macro Agregar: `( x , y )` , al igual que en una llamada de función ordinaria.

---

[1] (Citado de: ISO: IEC 9899: 201X 6.5.1.1 Selección genérica 3)
La expresión de control de una selección genérica no se evalúa.

Lea Selección genérica en línea: https://riptutorial.com/es/c/topic/571/seleccion-generica

Capítulo 60: Tipos de datos

Observaciones

- Aunque se requiere que `char` sea de 1 byte, **no se** requiere que 1 byte sea de 8 bits (a menudo también se denomina *octeto*), aunque la mayoría de las plataformas de computadoras modernas lo definen como 8 bits. El número de bits de la implementación por `char` lo proporciona la macro `CHAR_BIT` , definida en `<limits.h>` . POSIX requiere 1 byte para ser de 8 bits.
- Los tipos de enteros de ancho fijo deben usarse de forma dispersa, los tipos incorporados de C están diseñados para ser naturales en cada arquitectura, los tipos de ancho fijo solo deben usarse si necesita explícitamente un entero de tamaño específico (por ejemplo, para redes).

Examples

Tipos enteros y constantes

Los enteros con signo pueden ser de estos tipos (`int` después de `short` o `long` es opcional):

```
signed char c = 127; /* required to be 1 byte, see remarks for further information. */
signed short int si = 32767; /* required to be at least 16 bits. */
signed int i = 32767; /* required to be at least 16 bits */
signed long int li = 2147483647; /* required to be at least 32 bits. */
```

C99

```
signed long long int li = 2147483647; /* required to be at least 64 bits */
```

Cada uno de estos tipos de enteros con signo tiene una versión sin firmar.

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

Para todos los tipos, excepto `char` se asume la versión `signed` si se omite la parte `signed` o `unsigned` . El tipo `char` constituye un tercer tipo de carácter, diferente del `signed char` y `unsigned char` y la firma (o no) depende de la plataforma.

Se pueden escribir diferentes tipos de constantes enteras (llamadas *literales* en la jerga C) en diferentes bases y diferentes anchos, según su prefijo o sufijo.

```
/* the following variables are initialized to the same value: */
int d = 42;   /* decimal constant (base10) */
int o = 052;  /* octal constant (base8) */
int x = 0xaf; /* hexadecimal constants (base16) */
int X = 0XAf; /* (letters 'a' through 'f' (case insensitive) represent 10 through 15) */
```

Las constantes decimales siempre están `signed` . Las constantes hexadecimales comienzan con `0x` o `0X` y las constantes octales comienzan con `0` . Los dos últimos están `signed` o `unsigned` según si el valor se ajusta al tipo firmado o no.

```
/* suffixes to describe width and signedness : */
long int i = 0x32; /* no suffix represent int, or long int */
unsigned int ui = 65535u; /* u or U represent unsigned int, or long int */
long int li = 65536l; /* l or L represent long int */
```

Sin un sufijo, la constante tiene el primer tipo que se ajusta a su valor, es decir, una constante decimal que es más grande que `INT_MAX` es de tipo `long` si es posible, o `long long` contrario.

El archivo de encabezado `<limits.h>` describe los límites de los enteros de la siguiente manera. Sus valores definidos por la implementación serán iguales o mayores en magnitud (valor absoluto) a los que se muestran a continuación, con el mismo signo.

Macro	Tipo	Valor
`CHAR_BIT`	Objeto más pequeño que no es un campo de bits (byte)	8
`SCHAR_MIN`	`signed char`	-127 / - (2^{7} - 1)
`SCHAR_MAX`	`signed char`	+127 / 2^{7} - 1
`UCHAR_MAX`	`unsigned char`	255/2^{8} - 1
`CHAR_MIN`	`char`	vea abajo
`CHAR_MAX`	`char`	vea abajo
`SHRT_MIN`	`short int`	-32767 / - (2^{15} - 1)
`SHRT_MAX`	`short int`	+32767 / 2^{15} - 1
`USHRT_MAX`	`unsigned short int`	65535/2^{16} - 1
`INT_MIN`	`int`	-32767 / - (2^{15} - 1)
`INT_MAX`	`int`	+32767 / 2^{15} - 1
`UINT_MAX`	`unsigned int`	65535/2^{16} - 1
`LONG_MIN`	`long int`	-2147483647 / - (2^{31} - 1)
`LONG_MAX`	`long int`	+2147483647 / 2^{31} - 1
`ULONG_MAX`	`unsigned long int`	4294967295/2^{32} - 1

C99

Macro	Tipo	Valor
`LLONG_MIN`	`long long int`	-9223372036854775807 / - (2^{63} - 1)
`LLONG_MAX`	`long long int`	+9223372036854775807 / 2^{63} - 1
`ULLONG_MAX`	`unsigned long long int`	18446744073709551615/2^{64} - 1

Si el valor de un objeto de tipo `char` -se extiende cuando se usa en una expresión, el valor de `CHAR_MIN` será el mismo que el de `SCHAR_MIN` y el valor de `CHAR_MAX` será el mismo que el de `SCHAR_MAX` . Si el valor de un objeto de tipo `char` no se extiende al signo cuando se usa en una expresión, el valor de `CHAR_MIN` será 0 y el valor de `CHAR_MAX` será el mismo que el de `UCHAR_MAX` .

C99

El estándar C99 agregó un nuevo encabezado, `<stdint.h>` , que contiene definiciones para enteros de ancho fijo. Consulte el ejemplo de entero de ancho fijo para obtener una explicación más detallada.

Literales de cuerda

Un literal de cadena en C es una secuencia de caracteres, terminada por un cero literal.

```
char* str = "hello, world"; /* string literal */

/* string literals can be used to initialize arrays */
char a1[] = "abc"; /* a1 is char[4] holding {'a','b','c','\0'} */
char a2[4] = "abc"; /* same as a1 */
char a3[3] = "abc"; /* a1 is char[3] holding {'a','b','c'}, missing the '\0' */
```

Los literales de cadena **no** son **modificables** (y, de hecho, se pueden colocar en la memoria de solo lectura, como .rodata). Intentar alterar sus valores resulta en un comportamiento indefinido.

```
char* s = "foobar";
s[0] = 'F'; /* undefined behaviour */

/* it's good practice to denote string literals as such, by using `const` */
char const* s1 = "foobar";
s1[0] = 'F'; /* compiler error! */
```

Varios literales de cadena se concatenan en tiempo de compilación, lo que significa que puede escribir construcciones como éstas.

C99

```
/* only two narrow or two wide string literals may be concatenated */
char* s = "Hello, " "World";
```

C99

```
/* since C99, more than two can be concatenated */
```

```
/* concatenation is implementation defined */
char* s1 = "Hello" ", " "World";

/* common usages are concatenations of format strings */
char* fmt = "%" PRId16; /* PRId16 macro since C99 */
```

Los literales de cadena, al igual que las constantes de caracteres, admiten diferentes conjuntos de caracteres.

```
/* normal string literal, of type char[] */
char* s1 = "abc";

/* wide character string literal, of type wchar_t[] */
wchar_t* s2 = L"abc";
```

C11

```
/* UTF-8 string literal, of type char[] */
char* s3 = u8"abc";

/* 16-bit wide string literal, of type char16_t[] */
char16_t* s4 = u"abc";

/* 32-bit wide string literal, of type char32_t[] */
char32_t* s5 = U"abc";
```

Tipos de enteros de ancho fijo (desde C99)

C99

El encabezado `<stdint.h>` proporciona varias definiciones de tipo entero de ancho fijo. Estos tipos son *opcionales* y solo se proporcionan si la plataforma tiene un tipo de entero del ancho correspondiente, y si el tipo con signo correspondiente tiene una representación complementaria de dos valores negativos.

Consulte la sección de comentarios para obtener sugerencias de uso de tipos de ancho fijo.

```
/* commonly used types include */
uint32_t u32 = 32; /* exactly 32-bits wide */

uint8_t u8 = 255;  /* exactly 8-bits wide */

int64_t i64 = -65  /* exactly 64 bit in two's complement representation */
```

Constantes de punto flotante

El lenguaje C tiene tres tipos obligatorios de punto flotante real, `float` , `double` y `long double` .

```
float f = 0.314f;        /* suffix f or F denotes type float */
double d = 0.314;        /* no suffix denotes double */
long double ld = 0.314l; /* suffix l or L denotes long double */

/* the different parts of a floating point definition are optional */
```

```
double x = 1.; /* valid, fractional part is optional */
double y = .1; /* valid, whole-number part is optional */

/* they can also defined in scientific notation */
double sd = 1.2e3; /* decimal fraction 1.2 is scaled by 10^3, that is 1200.0 */
```

El encabezado `<float.h>` define varios límites para las operaciones de punto flotante.

Aritmética de punto flotante es la implementación definida. Sin embargo, la mayoría de las plataformas modernas (brazo, x86, x86_64, MIPS) usan operaciones de punto flotante IEEE 754 .

C también tiene tres tipos de puntos flotantes complejos opcionales que se derivan de lo anterior.

Interpretando declaraciones

Una peculiaridad sintáctica distintiva de C es que las declaraciones reflejan el uso del objeto declarado como sería en una expresión normal.

El siguiente conjunto de operadores con precedencia y asociatividad idénticos se reutilizan en los declaradores, a saber:

- el operador de "desreferencia" unaria `*` que denota un puntero;
- el operador binario `[]` "suscripción de matriz" que denota una matriz;
- el operador (1 + n) -ary `()` "llamada de función" que denota una función;
- los `()` paréntesis de agrupación que anulan la precedencia y la asociatividad del resto de los operadores listados.

Los tres operadores anteriores tienen la siguiente prioridad y asociatividad:

Operador	Precedencia relativa	Asociatividad
`[]` (suscripción de matriz)	1	De izquierda a derecha
`()` (llamada de función)	1	De izquierda a derecha
`*` (desreferencia)	2	De derecha a izquierda

Al interpretar declaraciones, uno tiene que comenzar desde el identificador hacia afuera y aplicar los operadores adyacentes en el orden correcto según la tabla anterior. Cada solicitud de un operador puede ser sustituida por las siguientes palabras en inglés:

Expresión	Interpretación
`thing[X]`	una matriz de tamaño `X` de ...
`thing(t1, t2, t3)`	una función que toma `t1` , `t2` , `t3` y regresa ...
`*thing`	un puntero a ...

De ello se deduce que el comienzo de la interpretación en inglés siempre comenzará con el identificador y terminará con el tipo que se encuentra en el lado izquierdo de la declaración.

Ejemplos

```
char *names[20];
```

`[]` tiene prioridad sobre `*` , por lo que la interpretación es: `names` es una matriz del tamaño 20 de un puntero a `char` .

```
char (*place)[10];
```

En caso de usar paréntesis para anular la precedencia, el `*` se aplica primero: `place` es un puntero a una matriz de tamaño 10 de `char` .

```
int fn(long, short);
```

No hay ninguna prioridad de qué preocuparse aquí: `fn` es una función que tarda `long` , `short` y devuelve `int` .

```
int *fn(void);
```

Primero se aplica `()` : `fn` es una función que `void` y devuelve un puntero a `int` .

```
int (*fp)(void);
```

Anular la prioridad de `()` : `fp` es un puntero a una función que `void` y devuelve `int` .

```
int arr[5][8];
```

Las matrices multidimensionales no son una excepción a la regla; los operadores `[]` se aplican en orden de izquierda a derecha según la asociatividad en la tabla: `arr` es una matriz de tamaño 5 de una matriz de tamaño 8 de `int` .

```
int **ptr;
```

Los dos operadores de referencia tienen la misma prioridad, por lo que la asociatividad surte efecto. Los operadores se aplican en el orden de derecha a izquierda: `ptr` es un puntero a un puntero a un `int` .

Declaraciones multiples

La coma se puede usar como separador (* no * actuando como el operador de coma) para delimitar múltiples declaraciones dentro de una sola declaración. La siguiente declaración contiene cinco declaraciones:

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

Los objetos declarados en el ejemplo anterior son:

- `fn` : una función que `void` y devuelve `int` ;
- `ptr` : un puntero a un `int` ;
- `fp` : un puntero a una función que toma `int` y devuelve `int` ;
- `arr` : una matriz de tamaño 10 de una matriz de tamaño 20 de `int` ;
- `num` : `int` .

Interpretación alternativa

Debido a que las declaraciones reflejan el uso, una declaración también puede interpretarse en términos de los operadores que podrían aplicarse sobre el objeto y el tipo resultante final de esa expresión. El tipo que se encuentra en el lado izquierdo es el resultado final que se obtiene después de aplicar todos los operadores.

```
/*
 * Subscripting "arr" and dereferencing it yields a "char" result.
 * Particularly: *arr[5] is of type "char".
 */
char *arr[20];

/*
 * Calling "fn" yields an "int" result.
 * Particularly: fn('b') is of type "int".
 */
int fn(char);

/*
 * Dereferencing "fp" and then calling it yields an "int" result.
 * Particularly: (*fp)() is of type "int".
 */
int (*fp)(void);

/*
 * Subscripting "strings" twice and dereferencing it yields a "char" result.
 * Particularly: *strings[5][15] is of type "char"
 */
char *strings[10][20];
```

Lea Tipos de datos en línea: https://riptutorial.com/es/c/topic/309/tipos-de-datos

Capítulo 61: Typedef

Introducción

El mecanismo `typedef` permite la creación de alias para otros tipos. No crea nuevos tipos. Las personas a menudo usan `typedef` para mejorar la portabilidad del código, para dar alias a los tipos de estructura o unión, o para crear alias para los tipos de función (o puntero de función).

En el estándar C, `typedef` se clasifica como una 'clase de almacenamiento' por conveniencia; se produce de forma sintáctica donde pueden aparecer clases de almacenamiento, como `static` o `extern` .

Sintaxis

- typedef existing_name alias_name;

Observaciones

Desventajas de Typedef

`typedef` podría conducir a la contaminación del espacio de nombres en grandes programas de C.

Desventajas de las estructuras Typedef

Además, las estructuras `typedef` sin un nombre de etiqueta son una causa importante de la imposición innecesaria de relaciones de ordenación entre archivos de encabezado.

Considerar:

```
#ifndef FOO_H
#define FOO_H 1

#define FOO_DEF (0xDEADBABE)

struct bar; /* forward declaration, defined in bar.h*/

struct foo {
    struct bar *bar;
};

#endif
```

Con tal definición, sin usar `typedefs` , es posible que una unidad de compilación incluya `foo.h` para obtener la definición `FOO_DEF` . Si no intenta eliminar la referencia al miembro de la `bar` de la estructura `foo` entonces no habrá necesidad de incluir el archivo `bar.h`

Typedef vs #define

`#define` es una directiva de preprocesador C que también se utiliza para definir los alias para varios tipos de datos similares a `typedef` pero con las siguientes diferencias:

- `typedef` se limita a dar nombres simbólicos a tipos solo cuando como `#define` se puede usar para definir alias para valores también.
- `typedef` interpretación `typedef` es realizada por el compilador mientras que las instrucciones `#define` son procesadas por el preprocesador.
- Tenga en cuenta que `#define cptr char *` seguido de `cptr a, b;` no hace lo mismo que `typedef char *cptr;` seguido de `cptr a, b;` . Con `#define` , `b` es una variable de `char` simple, pero también es un puntero con el `typedef` .

Examples

Typedef para Estructuras y Uniones

Puedes dar nombres de alias a una `struct` :

```
typedef struct Person {
    char name[32];
    int age;
} Person;

Person person;
```

En comparación con la forma tradicional de declarar estructuras, los programadores no necesitarían tener `struct` cada vez que declaran una instancia de esa estructura.

Tenga en cuenta que el nombre de `Person` (a diferencia de `struct Person`) no se define hasta el último punto y coma. Por lo tanto, para las listas enlazadas y las estructuras de árbol que deben contener un puntero al mismo tipo de estructura, debe utilizar:

```
typedef struct Person {
    char name[32];
    int age;
    struct Person *next;
} Person;
```

o:

```
typedef struct Person Person;

struct Person {
    char name[32];
    int age;
    Person *next;
};
```

El uso de un `typedef` para un tipo de `union` es muy similar.

```
typedef union Float Float;

union Float
{
    float f;
    char  b[sizeof(float)];
};
```

Se puede utilizar una estructura similar a esta para analizar los bytes que forman un valor `float` .

Usos Simples de Typedef

Para dar nombres cortos a un tipo de datos

En lugar de:

```
long long int foo;
struct mystructure object;
```

uno puede usar

```
/* write once */
typedef long long ll;
typedef struct mystructure mystruct;

/* use whenever needed */
ll foo;
mystruct object;
```

Esto reduce la cantidad de escritura necesaria si el tipo se usa muchas veces en el programa.

Mejora de la portabilidad

Los atributos de los tipos de datos varían entre las diferentes arquitecturas. Por ejemplo, un `int` puede ser un tipo de 2 bytes en una implementación y un tipo de 4 bytes en otra. Supongamos que un programa necesita usar un tipo de 4 bytes para ejecutarse correctamente.

En una implementación, deje que el tamaño de `int` sea de 2 bytes y el de `long` sea de 4 bytes. En otra, deje que el tamaño de `int` sea de 4 bytes y el de `long` sea de 8 bytes. Si el programa está escrito usando la segunda implementación,

```
/* program expecting a 4 byte integer */
int foo; /* need to hold 4 bytes to work */
/* some code involving many more ints */
```

Para que el programa se ejecute en la primera implementación, todas las declaraciones `int` tendrán que cambiarse a `long` .

```
/* program now needs long */
long foo; /*need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

Para evitar esto, se puede usar typedef

```
/* program expecting a 4 byte integer */
typedef int myint; /* need to declare once - only one line to modify if needed */
myint foo; /* need to hold 4 bytes to work */
/* some code involving many more myints */
```

Entonces, solo la instrucción typedef tendrá que cambiarse cada vez, en lugar de examinar todo el programa.

C99

El encabezado <stdint.h> y el encabezado <inttypes.h> relacionado definen nombres de tipo estándar (usando typedef) para enteros de varios tamaños, y estos nombres son a menudo la mejor opción en el código moderno que necesita enteros de tamaño fijo. Por ejemplo, uint8_t es un tipo entero de 8 bits sin signo; int64_t es un tipo entero de 64 bits con signo. El tipo uintptr_t es un tipo entero sin signo lo suficientemente grande como para contener cualquier puntero a objeto. Estos tipos son teóricamente opcionales, pero es raro que no estén disponibles. Existen variantes como uint_least16_t (el tipo de entero sin signo más pequeño con al menos 16 bits) e int_fast32_t (el tipo de entero con signo más rápido con al menos 32 bits). Además, intmax_t y uintmax_t son los tipos de enteros más grandes admitidos por la implementación. Estos tipos son obligatorios.

Especificar un uso o mejorar la legibilidad.

Si un conjunto de datos tiene un propósito particular, uno puede usar typedef para darle un nombre significativo. Además, si la propiedad de los datos cambia de modo que el tipo base deba cambiar, solo se deberá cambiar la declaración typedef , en lugar de examinar todo el programa.

Typedef para punteros de función

Podemos usar typedef para simplificar el uso de punteros de función. Imagina que tenemos algunas funciones, todas con la misma firma, que usan su argumento para imprimir algo de diferentes maneras:

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i)
        printf("%d\n", i);
}

void print_n(int n)
{
    printf("%d\n, n);
}
```

Ahora podemos usar typedef para crear un tipo de puntero de función llamado impresora:

```
typedef void (*printer_t)(int);
```

Esto crea un tipo, llamado `printer_t` para un puntero a una función que toma un solo argumento `int` y no devuelve nada, lo que coincide con la firma de las funciones que tenemos arriba. Para usarlo creamos una variable del tipo creado y le asignamos un puntero a una de las funciones en cuestión:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Luego, para llamar a la función apuntada por la variable de puntero de función:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);        // So does this
```

Por lo tanto, `typedef` permite una sintaxis más simple cuando se trata de punteros de función. Esto se hace más evidente cuando los punteros de función se usan en situaciones más complejas, como argumentos a funciones.

Si está utilizando una función que toma un puntero de función como parámetro sin un tipo de puntero de función definido, la definición de la función sería

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

Sin embargo, con el `typedef` es:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

Del mismo modo, las funciones pueden devolver punteros a funciones y, de nuevo, el uso de `typedef` puede hacer que la sintaxis sea más simple al hacerlo.

Un ejemplo clásico es la función de `signal` de `<signal.h>` . La declaración para ello (de la norma C) es:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Esa es una función que toma dos argumentos: un `int` y un puntero a una función que toma un `int` como un argumento y no devuelve nada, y que devuelve un puntero para que funcione como su segundo argumento.

Si definimos un tipo `SigCatcher` como un alias para el puntero al tipo de función:

```
typedef void (*SigCatcher)(int);
```

entonces podríamos declarar `signal()` usando:

```
SigCatcher signal(int sig, SigCatcher func);
```

En general, esto es más fácil de entender (aunque el estándar C no eligió definir un tipo para realizar el trabajo). La función de `signal` toma dos argumentos, un `int` y un `SigCatcher` , y devuelve un `SigCatcher` , donde un `SigCatcher` es un puntero a una función que toma un argumento `int` y no devuelve nada.

Aunque el uso de los nombres `typedef` para los tipos de puntero a función facilita la vida, también puede generar confusión para otras personas que mantendrán su código más adelante, así que úselo con precaución y la documentación adecuada. Véase también punteros a funciones .

Lea Typedef en línea: https://riptutorial.com/es/c/topic/2681/typedef

Capítulo 62: Uniones

Examples

Diferencia entre estructura y unión.

Esto ilustra que los miembros de la unión comparten memoria y que los miembros de la estructura no comparten memoria.

```
#include <stdio.h>
#include <string.h>

union My_Union
{
  int variable_1;
  int variable_2;
};

struct My_Struct
{
  int variable_1;
  int variable_2;
};

int main (void)
{
  union My_Union u;
  struct My_Struct s;
  u.variable_1 = 1;
  u.variable_2 = 2;
  s.variable_1 = 1;
  s.variable_2 = 2;
  printf ("u.variable_1: %i\n", u.variable_1);
  printf ("u.variable_2: %i\n", u.variable_2);
  printf ("s.variable_1: %i\n", s.variable_1);
  printf ("s.variable_2: %i\n", s.variable_2);
  printf ("sizeof (union My_Union): %i\n", sizeof (union My_Union));
  printf ("sizeof (struct My_Struct): %i\n", sizeof (struct My_Struct));
  return 0;
}
```

Uso de uniones para reinterpretar valores.

Algunas implementaciones de C permiten que el código escriba en un miembro de un tipo de unión y luego lea de otro para realizar una especie de reinterpretación de la conversión (analizando el nuevo tipo como la representación de bits del antiguo).

Es importante tener en cuenta, sin embargo, esto no está permitido por el estándar C actual o el pasado y dará lugar a un comportamiento indefinido, sin embargo, es una extensión muy común ofrecida por los compiladores (así que verifique los documentos de su compilador si planea hacerlo) .

Un ejemplo real de esta técnica es el algoritmo de "Raíz Cuadrada Inversa Rápida", que se basa en los detalles de implementación de los números de punto flotante IEEE 754 para realizar una raíz cuadrada inversa más rápidamente que mediante operaciones de punto flotante. Este algoritmo se puede realizar a través del lanzamiento de punteros. (que es muy peligroso y rompe la regla estricta de aliasing) o a través de una unión (que sigue siendo un comportamiento indefinido pero funciona en muchos compiladores):

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* Float must be 32 bits IEEE 754 for this to work */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input;     /* Assign to the float member */
    i = x.intMember;           /* Read back from the integer member */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i;           /* Assign to the integer member */
    f = x.floatMember;         /* Read back from the float member */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

Esta técnica fue ampliamente utilizada en gráficos y juegos de computadora en el pasado debido a su mayor velocidad en comparación con el uso de operaciones de punto flotante, y es en gran medida un compromiso, perdiendo cierta precisión y siendo muy poco portátil a cambio de la velocidad.

Escribir a un miembro del sindicato y leer de otro.

Los miembros de un sindicato comparten el mismo espacio en la memoria. Esto significa que al escribir en un miembro se sobrescriben los datos en todos los demás miembros y que la lectura de un miembro da como resultado la misma información que la lectura de todos los demás miembros. Sin embargo, dado que los miembros de la unión pueden tener diferentes tipos y tamaños, los datos que se leen pueden interpretarse de manera diferente, consulte http://www.riptutorial.com/c/example/9399/using-unions-to-reinterpret-values

El ejemplo simple a continuación demuestra una unión con dos miembros, ambos del mismo tipo. Muestra que escribir en el miembro `m_1` hace que el valor escrito se lea del miembro `m_2` y que escribir en el miembro `m_2` resultado que el valor escrito se lea en el miembro `m_1` .

```
#include <stdio.h>

union my_union /* Define union */
{
    int m_1;
    int m_2;
};
```

```
int main (void)
{
    union my_union u;             /* Declare union */
    u.m_1 = 1;                    /* Write to m_1 */
    printf("u.m_2: %i\n", u.m_2); /* Read from m_2 */
    u.m_2 = 2;                    /* Write to m_2 */
    printf("u.m_1: %i\n", u.m_1); /* Read from m_1 */
    return 0;
}
```

Resultado

```
u.m_2: 1
u.m_1: 2
```

Lea Uniones en línea: https://riptutorial.com/es/c/topic/7645/uniones

Capítulo 63: Valgrind

Sintaxis

- valgrind *nombre-programa- argumentos-opcionales* < *entrada de prueba*

Observaciones

Valgrind es una herramienta de depuración que se puede utilizar para diagnosticar errores relacionados con la administración de memoria en los programas de C. Valgrind se puede usar para detectar errores, como el uso de punteros no válidos, como escribir o leer más allá del espacio asignado, o hacer una llamada no válida a `free()` . También se puede utilizar para mejorar las aplicaciones a través de funciones que realizan perfiles de memoria.

Para más información vea http://valgrind.org .

Examples

Corriendo valgrind

```
valgrind ./my-program arg1 arg2 < test-input
```

Esto ejecutará su programa y producirá un informe de las asignaciones y desasignaciones que realizó. También le advertirá acerca de los errores comunes, como el uso de memoria no inicializada, la desreferenciación de los punteros a lugares extraños, la eliminación del final de los bloques asignados con malloc o la ausencia de bloques libres.

Añadiendo banderas

También puede activar más pruebas, como:

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

Consulte valgrind --help para obtener más información sobre las (muchas) opciones, o consulte la documentación en http://valgrind.org/ para obtener información detallada sobre el significado de la salida.

Bytes perdidos - Olvidando liberar

Aquí hay un programa que llama malloc pero no es gratis:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
```

```
{
    char *s;

    s = malloc(26); // the culprint

    return 0;
}
```

Sin argumentos adicionales, valgrind no buscará este error.

Pero si `--leak-check=yes` o `--tool=memcheck` , se quejará y mostrará las líneas responsables de esas pérdidas de memoria si el programa se compiló en modo de depuración:

```
$ valgrind -q --leak-check=yes ./missing_free
==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4776==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==4776==    by 0x80483F8: main (missing_free.c:9)
==4776==
```

Si el programa no está compilado en modo de depuración (por ejemplo, con el indicador `-g` en GCC), todavía nos mostrará dónde ocurrió la fuga en términos de la función relevante, pero no las líneas.

Esto nos permite retroceder y ver qué bloque se asignó en esa línea e intentar rastrear hacia adelante para ver por qué no se liberó.

Errores más comunes encontrados al usar Valgrind

Valgrind le proporciona las *líneas en las que se produjo el error* al final de cada línea en el formato `(file.c:line_no)` . Los errores en valgrind se resumen de la siguiente manera:

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Los errores más comunes incluyen:

1. **Errores ilegales de lectura / escritura**

```
==8451== Invalid read of size 2
==8451==    at 0x4E7381D: getenv (getenv.c:84)
==8451==    by 0x4EB1559: __libc_message (libc_fatal.c:80)
==8451==    by 0x4F5256B: __fortify_fail (fortify_fail.c:37)
==8451==    by 0x4F5250F: __stack_chk_fail (stack_chk_fail.c:28)
==8451==    by 0x40059C: main (valg.c:10)
==8451==  Address 0x700000007 is not stack'd, malloc'd or (recently) free'd
```

Esto sucede cuando el código comienza a acceder a la memoria que no pertenece al programa. El tamaño de la memoria a la que se accede también le da una indicación de qué variable se usó.

2. **Uso de variables no inicializadas**

```
==8795== 1 errors in context 5 of 8:
```

```
==8795== Conditional jump or move depends on uninitialised value(s)
==8795==    at 0x4E881AF: vfprintf (vfprintf.c:1631)
==8795==    by 0x4E8F898: printf (printf.c:33)
==8795==    by 0x400548: main (valg.c:7)
```

De acuerdo con el error, en la línea 7 de `main` de `valg.c` , la llamada a `printf()` pasó una variable sin inicializar a `printf` .

3. **Liberación ilegal de la memoria.**

```
==8954== Invalid free() / delete / delete[] / realloc()
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x4005A8: main (valg.c:10)
==8954==  Address 0x5203040 is 0 bytes inside a block of size 240 free'd
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40059C: main (valg.c:9)
==8954==  Block was alloc'd at
==8954==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40058C: main (valg.c:7)
```

Según valgrind, el código liberó la memoria ilegalmente (una segunda vez) en la *línea 10* de `valg.c` , mientras que ya estaba liberado en la *línea 9* , y al bloque mismo se le asignó memoria en la *línea 7* .

Lea Valgrind en línea: https://riptutorial.com/es/c/topic/2674/valgrind

Creditos

S. No	Capítulos	Contributors
1	Empezando con el lenguaje C	4444, Abhineet, Alejandro Caro, alk, Ankush, ArturFH, Bahm, bevenson, bfd, Blacksilver, blatinox, bta, chqrlie, Community, Dair, Dan Fairaizl, Daniel Jour, Daniel Margosian, David G., David Grayson, Donald Duck, Dov Benyomin Sohacheski, Ed Cottrell, employee of the month, EOF, EsmaeelE, Frosty The DopeMan, Iskar Jarak, Jens Gustedt, John Slegers, JonasCz, Jonathan Leffler, Juan T, juleslasne, Kusalananda, Leandros, LiHRaM, Lundin, Malick, Mark Yisri, MC93, MoultoB, msohng, Myst, Narox Nox, Neal, Nemanja Boric, Nicolas Verlet, OiciTrap, P.P., PSN, Rakitić, RamenChef, Roland Illig, Ryan Hilbert, Shoe , Shog9, skrtbhtngr, sohnryang, stackptr, syb0rg, techydesigner, tlhIngan, Toby, vasili111, Vin, Vraj Pandya
2	- clasificación de caracteres y conversión	Alejandro Caro, Jonathan Leffler, Roland Illig, Toby
3	Acolchado y embalaje de estructuras	EsmaeelE, Jarrod Dixon, Jedi, Jesferman, Jonathan Leffler, Liju Thomas, MayeulC, tilz0R
4	Afirmación	2501, AShelly, Blagovest Buyukliev, bta, eush77, greatwolf, J Wu, Jens Gustedt, Jonathan Leffler, Jossi, jxh, Leandros, Malcolm McLean, Ryan Haining, stackptr, syb0rg, Tim Post, Toby
5	Aliasing y tipo efectivo.	2501, 4386427, Jens Gustedt
6	Ámbito identificador	embedded_guy, Firas Moalla, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler
7	Archivos y flujos de E / S	alk, bevenson, EWoodward, haccks, iRove, Jean Vitor, Jens Gustedt, Jonathan Leffler, Jossi, Leandros, Malcolm McLean, Pedro Henrique A. Oliveira, RamenChef, reshad, Snaipe, stackptr, syb0rg, tkk, Toby, tversteeg, William Pursell
8	Argumentos de línea de comando	4386427, A B, alk, drov, dvhh, Jonathan Leffler, Malcolm McLean, Shog9, syb0rg, Toby, Woodrow Barlow, Yotam Salmon
9	Argumentos	2501, Blacksilver, eush77, Jean-Baptiste Yunès, Jonathan

	variables	Leffler, Leandros, mirabilos, syb0rg, Toby
10	Arrays	2501, alk, AnArrayOfFunctions, AShelly, cdrini, cSmout, Dariusz , Elazar, Eli Sadoff, Firas Moalla, Guy, Iskar Jarak, Jasmin Solanki, Jens Gustedt, John Bollinger, Jonathan Leffler, L.V.Rao, Leandros, Liju Thomas, lordjohncena, Magisch, mhk, OznOg, Ray, Ryan Haining, Ryan Hilbert, stackptr, Toby, Waqas Bukhary
11	Atomística	Jens Gustedt
12	Booleano	alk, Bob__, Braden Best, Chrono Kitsune, dhein, Insane, Jens Gustedt, Magisch, Mateusz Piotrowski, Peter, Toby
13	Calificadores de tipo	alk, Blagovest Buyukliev, Jens Gustedt, Jesferman, madD7, tversteeg
14	Campos de bits	alk, EvilTeach, Fantastic Mr Fox, haccks, Ishay Peled, Jens Gustedt, John Odom, Jonathan Leffler, Lundin, madD7, Paul Hutchinson, RamenChef, Rishikesh Raje, Toby, vkgade
15	Clases de almacenamiento	alk, Blagovest Buyukliev, Chrono Kitsune, greatwolf, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, L.V.Rao, madD7, Neui, Nitinkumar Ambekar, P.P., Toby, tversteeg, vuko_zrno
16	Comentarios	Ankush, Chandrahas Aroori, Jonathan Leffler, Toby
17	Compilacion	alk, Amani Kilumanga, bevenson, Blacksilver, Firas Moalla, haccks, Ishay Peled, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, Jossi, jxh, MC93, MikeCAT, nathanielng, P.P., Qrchack, R. Joiny, syb0rg, Toby, tofro, Turtle, Vraj Pandya, Алексей Неудачин
18	Comportamiento definido por la implementación	Jens Gustedt, John Bollinger, P.P.
19	Comportamiento indefinido	2501, Abhineet, Aleksi Torhamo, alk, Antti Haapala, Armali, Ben Steffan, blatinox, bta, BurnsBA, caf, Christoph, Cody Gray, Community, cshu, DaBler, Daniel Jour, DarkDust, FedeWar, Firas Moalla, Giorgi Moniava, gsamaras, haccks, hmijail, honk, Jacob H, Jean-Baptiste Yunès, Jens Gustedt, John, John Bollinger, Jonathan Leffler, Kamiccolo, Leandros, Lundin, Magisch, Mark Yisri, Martin, MikeCAT, Nemanja Boric, P.P., Peter, Roland Illig, TimF, Toby, tversteeg, user45891, Vasfed, void
20	Comunicación entre	CLDSEED, EsmaeelE, Jonathan Leffler, Toby

	procesos (IPC)	
21	Conversiones implícitas y explícitas	alk, Firas Moalla, Jens Gustedt, Jeremy Thien, kdopen, Lundin, Toby
22	Crear e incluir archivos de encabezado	4444, Jonathan Leffler, patrick96, Sirsireesh Kodali
23	Declaración vs Definición	Ashish Ahuja, foxtrot9, Kerrek SB, Toby
24	Declaraciones	alk, AnArrayOfFunctions, Blacksilver, Firas Moalla, J Wu, Jens Gustedt, Jonathan Leffler, Jonathon Reinhart
25	Declaraciones de selección	alk, bevenson, Blagovest Buyukliev, Faisal Mudhir, GoodDeeds, gsamaras, jxh, L.V.Rao, lordjohncena, MikeCAT, NeoR, noamgot, OznOg, P.P., Toby, tofro
26	Efectos secundarios	EsmaeelE, Jonathan Leffler, L.V.Rao, madD7, RamenChef, Sirsireesh Kodali, Toby
27	En línea	Alex, EsmaeelE, Jens Gustedt, Jonathan Leffler, Toby
28	Entrada / salida formateada	alk, fluter, Jonathan Leffler, Jossi, lardenn, MikeCAT, polarysekt , StardustGogeta
29	Enumeraciones	Alejandro Caro, alk, jasoninnn, Jens Gustedt, Jonathan Leffler, OznOg, Toby
30	Errores comunes	abacles, Accepted Answer, alk, bevenson, Bjorn A., Chrono Kitsune, clearlight, Community, Dmitry Grigoryev, Dreamer, Dunno, FedeWar, Fred Barclay, Gavin Higham, Giorgi Moniava, hlovdal, Ishay Peled, Jeremy, John Hascall, Jonathan Leffler, Ken Y-N, Leandros, Lord Farquaad, MikeCAT, P.P., Roland Illig , rxantos, Sourav Ghosh, stackptr, Tamarous, techEmbedded, Toby, Waqas Bukhary
31	Estructuras	alk, Chrono Kitsune, Damien, Elazar, EsmaeelE, Faisal Mudhir, Firas Moalla, gmug, jasoninnn, Jens Gustedt, Jonathan Leffler, Jossi, kamoroso94, Madhusoodan P, OznOg, Paul Kramme, PhotometricStereo, RamenChef, Toby, Vality
32	Generación de números aleatorios	dylanweber, ganchito55, haccks, hexwab, Jonathan Leffler, Leandros, Malcolm McLean, MikeCAT, Toby
33	Gestión de la memoria	4386427, alk, Anderson Giacomolli, Andrey Markeev, Ankush, Antti Haapala, Cullub, Daksh Gupta, dhein, dkrmr, doppelheathen, dvhh, elslooo, EOF, EsmaeelE, Firas Moalla, fluter, gdc, greatwolf, honk, Jens Gustedt, Jonathan Leffler,

		juleslasne, Luiz Berti, madD7, Malcolm McLean, Mark Yisri, Matthieu, Neui, P.P., Paul Campbell, Paul V, reflective_mind, Seth, Srikar, stackptr, syb0rg, Tamarous, tbodt, the sudhakar, Toby, tofro, Vivek S, vuko_zrno, Wyzard
34	Hilos (nativos)	alk, Jens Gustedt, P.P.
35	Inicialización	Jonathan Leffler, Liju Thomas, P.P.
36	Instrumentos de cuerda	4386427, alk, Amani Kilumanga, Andrey Markeev, bevenson, catalogue_number, Chris Sprague, Chrono Kitsune, Cody Gray, Damien, Daniel, depperm, dylanweber, FedeWar, Firas Moalla, haccks, Ishay Peled, jasoninnn, Jean-Baptiste Yunès, Jens Gustedt, John Bollinger, Jonathan Leffler, Leandros, Malcolm McLean, mantal, MikeCAT, P.P., Purag, Roland Illig, stackptr, still_learning, syb0rg, Toby, vasili111, Waqas Bukhary, Wolf, Wyzard, Алексей Неудачин
37	Iteraciones / bucles de repetición: for, while, do-while	alk, GoodDeeds, Jens Gustedt, jxh, L.V.Rao, Malcolm McLean, Nagaraj, RamenChef, reshad, Toby
38	Lenguajes comunes de programación C y prácticas de desarrollador.	Chandrahas Aroori, Jonathan Leffler, Nityesh Agarwal, Shubham Agrawal
39	Listas enlazadas	4386427, alk, Andrea Biondo, bevenson, iRove, Jonathan Leffler, Jossi, Leandros, Mateusz Piotrowski, Ryan, Toby
40	Literales compuestos	alk, haccks, Jens Gustedt, Kerrek SB
41	Literales para números, caracteres y cadenas.	Jens Gustedt, Jonathan Leffler, Klas Lindbäck, Neui, Paul92, Toby
42	Los operadores	202_accepted, 3442, alk, Amani Kilumanga, Andrea Corbelli, Bakhtiar Hasan, BenG, blatinox, cpplearner, Damien, Dariusz, EsmaeelE, Faisal Mudhir, Fantastic Mr Fox, Firas Moalla, gsamaras, hrs, Iwillnotexist Idonotexist, Jens Gustedt, Jonathan Leffler, kdopen, Ken Y-N, L.V.Rao, Leandros, LostAvatar, Magisch, MikeCAT, noamgot, P.P., Paul92, Peter, stackptr, Toby, Will, Wolf, Yu Hao
43	Macros x	Cimbali, Jens Gustedt, John Bollinger, Leandros, MD XF, mpromonet, poolie, RamenChef, technosaurus, templatetypedef , Toby
44	Manejo de errores	Jens Gustedt, stackptr

45	Manejo de señales	3442, alk, Dariusz, Jens Gustedt, Leandros, mirabilos
46	Marcos de prueba	Community, EsmaeelE, Jonathan Leffler, lordjohncena, Toby, user2314737, vuko_zrno
47	Matemáticas estándar	Alejandro Caro, alk, Blagovest Buyukliev, immerhart, Jonathan Leffler, manav m-n, Toby
48	Montaje en línea	bevenson, EsmaeelE, Jonathan Leffler
49	Multihilo	Parham Alvani, Toby
50	Parámetros de función	2501, Alejandro Caro, alk, Chrono Kitsune, ganesh kumar, George Stocker, Jens Gustedt, Jonathan Leffler, Leandros, MikeCAT, Minar Ashiq Tishan, P.P., RamenChef, Richard Chambers, someoneigna, syb0rg, Toby
51	Pasar arrays 2D a funciones	deamentiaemundi, Malcolm McLean, Shrinivas Patgar, Toby
52	Preprocesador y Macros	Alex Garcia, alk, bevenson, bwoebi, Dariusz, DrPrItay, Erlend Graff, EsmaeelE, EvilTeach, fastlearner, Firas Moalla, gman, hashdefine, hlovdal, javac, Jens Gustedt, Jonathan Leffler, Justin, Leandros, luser droog, Madhusoodan P, Maniero, mnoronha, Nitinkumar Ambekar, P.P., Paul J. Lucas, Peter, Richard Chambers, Robert Baldyga, stackptr, Toby, v7d8dpo4
53	Punteros	0xEDD1E, alk, Altece, Amani Kilumanga, Andrey Markeev, Ankush, Antti Haapala, Ashish Ahuja, Bjorn A., bruno, bta, chqrlie, Courtney Pattison, Dair, Daniel Porteous, David G., dhein, dkrmr, Don't You Worry Child, e.jahandar, elslooo, EOF, erebos, Faisal Mudhir, Fantastic Mr Fox, FedeWar, Firas Moalla, fluter, foxtrot9, Gavin Higham, gdc, Giorgi Moniava, gsamaras, haccks, haltode, Harry Johnston, Hemant Kumar, honk, Jens Gustedt, Jonathan Leffler, Jonnathan Soares, Josh de Kock, jpX, L.V.Rao, LaneL, Leandros, Luiz Berti, Malcolm McLean, Matthieu, Michael Fitzpatrick, MikeCAT, Neui, Nitinkumar Ambekar, OiciTrap, P.P., Pbd, Peter, RamenChef, raymai97, Rohan, Sergey, Shahbaz, signal, slugonamission, solomonope, someoneigna, Spidey, Srikar, stackptr, syb0rg, tbodt, the sudhakar, thndrwrks, Toby, Vality, vijay kant sharma, Vivek S, Wyzard, xhienne, Алексей Неудачин
54	Punteros a funciones	Alejandro Caro, alk, David Refaeli, Filip Allberg, hlovdal, John Burger, Leandros, Malcolm McLean, P.P., Srikar, stackptr, Toby
55	Puntos de secuencia	2501, Armali, bta, Community, haccks, Jens Gustedt, John Bode, Toby

56	Restricciones	Armali, Toby, Vality
57	Saltar declaraciones	alk, Jens Gustedt, Jonathan Leffler, lordjohncena, Malcolm McLean, Sourav Ghosh, syb0rg, Toby
58	Secuencia de caracteres de múltiples caracteres	Jonathan Leffler, PassionInfinite, Toby
59	Selección genérica	2501, Jens Gustedt, Sun Qingyao
60	Tipos de datos	2501, alk, Blagovest Buyukliev, Firas Moalla, Jens Gustedt, Keith Thompson, Ken Y-N, Leandros, P.P., Peter, WMios
61	Typedef	Buser, Chandrahas Aroori, GoodDeeds, Jonathan Leffler, mame98, PhotometricStereo, Stephen Leppik, Toby
62	Uniones	Jossi, RamenChef, Toby, Vality
63	Valgrind	abacles, alk, Ankush, Chandrahas Aroori, Devansh Tandon, drov, Firas Moalla, J F, Jonathan Leffler, vasili111