Notas de curso
Temas Selectos de Lenguajes de Programación
El lenguaje de programación C++

Rodrigo Alexander Castro Campos
UAM Azcapotzalco, División de CBI
https://racc.mx
https://omegaup.com/profile/rcc

Última revisión: 24 de noviembre de 2022

1. Introducción al lenguaje C++ para programadores de C

El lenguaje de programación C++ es un lenguaje eficiente de alto nivel diseñado por Bjarne Stroustrup en 1979. El lenguaje C++ fue estandarizado por primera vez ante la Organización Internacional de Normalización (ISO) en 1989 y su versión más reciente fue publicada en 2020. Se usa ampliamente en la implementación de sistemas operativos, videojuegos y sistemas embedidos.

Los compiladores más famosos de C++ son GCC, Visual Studio y Clang. En Windows lo más común es usar Visual Studio o MinGW que es una variante de GCC, mientras que en MacOS se suele usar Clang y en Linux se suele usar GCC. Como Windows no trae preinstalado ningún compilador, se sugiere descargar la versión más reciente de Visual Studio o de MinGW, por ejemplo WinLibs. En Linux lo normal es tener GCC preinstalado. En MacOS, Apple recomienda instalar Xcode que usa Clang.

El lenguaje C++ es casi un superconjunto del lenguaje C. Sin entrar en demasiados detalles, esto significa que prácticamente todo código válido en C también compila en C++. Esta amplia compatibilidad permite usar en C++ las funciones de `<stdio.h>` y del resto de la biblioteca de C.

Código	Salida

`#include <stdio.h>  // Válido en C++`
 	
`int main( ) {`
`  printf("Hola mundo");`
`  return 0;`
`}`
	`Hola mundo`

Adicionalmente, un archivo de la biblioteca de C puede incluirse con el prefijo `c` y sin extensión de archivo. Por ejemplo, `<stdio.h>` puede incluirse como `<cstdio>` y las utilidades declaradas en el mismo pueden usarse con (y posiblemente también sin) el prefijo `std::` (por ejemplo, `printf` o `std::printf`) donde `std` es el nombre del espacio de nombres estándar y `::` es el operador de resolución de ámbito.

Código	Salida

`#include <cstdio>  // Válido en C++`
 	
`int main( ) {`
`  std::printf("Hola mundo");`
`  return 0;`
`}`
	`Hola mundo`

Sí existen diferencias entre incluir el archivo original de C o la versión de C++ además del prefijo del espacio de nombres. Por ejemplo, la versión de C++ tiene permitido definir funciones adicionales a las existentes en C. En general, los archivos exclusivos de la biblioteca estándar de C++ no lleva extensión.

A pesar de su flexibilidad, las funciones `scanf` y `printf` tienen inconvenientes serios como el tener que elegir el especificador de formato `%` correcto para cada tipo de dato, o que olvidar el `&` en `scanf` al leer una variable no provoca un error de compilación, pero sí uno de ejecución.

Código	Diagnóstico

`int n;`
`scanf("%d", n);  // error en ejecución`
`printf("%f", n); // formato incorrecto`
	`Error en ejecución`

La entrada y salida al estilo C++ se realiza con las utilidades declaradas en el archivo de biblioteca `<iostream>`. Las variables `std::cin` y `std::cout` de `<iostream>` representan la entrada y salida de datos, respectivamente. El origen y el destino de los datos son los mismos que con `scanf` y `printf`, por lo que es fácil entender su semántica. Estas variables emplean una fea notación con los operadores `>>` y `<<` para realizar la lectura y la escritura respectivamente. Una forma fácil de aprender el sentido de los operadores es: si los datos van de la entrada (`std::cin`) a la variable, entonces el operador es `>>`, mientras que si los datos van de la variable a la salida (`std::cout`), entonces el operador es `<<`. Por ejemplo:

Código	Entrada	Salida

`#include <iostream>`
 	
`int main( ) {`
`  int a, b;`
`  std::cin >> a;  // leer a`
`  std::cin >> b;  // leer b`
`  std::cout << a + b;  // imprimir`
`  return 0;`
`}`
	`1 2`	`3`

Podemos leer e imprimir más de un valor en la misma línea.

Código	Entrada	Salida

`#include <iostream>`
 	
`int main( ) {`
`  int a, b;`
`  std::cin >> a >> b;`
`  std::cout << a + b << " " << a - b;`
`  return 0;`
`}`
	`8 3`	`11 5`

Una gran ventaja de usar `std::cin` y `std::cout` es que funcionan de manera uniforme con los tipos primitivos y con muchos otros tipos de la biblioteca. Por omisión, el operador `>>` siempre se salta espacios.

Código	Entrada	Salida

`int n; char c; float f;`
`std::cin >> n >> c >> f;`
`std::cout << f << " " << c << " " << n;`
	`1 @ 3.14`	`3.14 @ 1`

2. La función `main` en C++

Se puede considerar (aunque no es del todo cierto) que la ejecución de un programa en C++ comienza en la función `main`. El estándar acepta dos definiciones distintas de esta función:

Código	Código

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

La primera de ellas denota la versión de `main` que no toma parámetros proveídos por parte del sistema operativo, mientras que la segunda versión sí los recibe (normalmente al ejecutar el programa mediante la terminal o línea de comandos). Esta segunda versión es similar a la declaración de `main` del lenguaje de programación Java que tiene como parámetro un arreglo de cadenas. Por otra parte, en C es común ver la primera versión de `main` declarada con `void` como parámetro. Sin embargo, esto está despreciado.

La función `main` debe regresar un entero, el cual se interpreta como el estado en el que el programa terminó. Por convención, el valor 0 se interpreta como que el programa terminó normalmente. De forma similar, en `<stdlib.h>` se declaran dos constantes `EXIT_SUCCESS` y `EXIT_FAILURE` que representan una terminación normal o anormal, respectivamente.

Código	Código equivalente

`int main( ) {`
`  return 0;`
`}`
	`#include <stdlib.h>`
	
	`int main( ) {`
	`  return EXIT_SUCCESS;`
	`}`

Una regla importante del lenguaje es que toda función que prometa regresar un valor debe hacerlo. Sin embargo, `main` es la única excepción en el sentido de que si la ejecución de `main` llega a la llave de cierre, entonces se regresa 0 automáticamente.

Código	Código equivalente

`int main( ) {`
`  return 0;`
`}`
	`int main( ) {`
	`}`

El programa completo termina cuando `main` termina. Además, en la biblioteca `<stdlib.h>` está definida la función `exit` que toma un entero y termina el programa desde cualquier parte del mismo y actúa como si `main` hubiera devuelto dicho entero.

3. Tipos fundamentales en C++

En C++, las variables que consumen menos memoria son de tipo `char`. Históricamente, `char` ha sido del mismo tamaño que la unidad mínima direccionalmente de la memoria, siendo este último concepto también conocido como byte. La cantidad de bits de un `char` se define en `<limits.h>` bajo el nombre de la constante `CHAR_BIT`. El lenguaje garantiza que `CHAR_BIT` es mayor o igual a 7, aunque actualmente la gran mayoría de los procesadores tienen un `CHAR_BIT` igual a 8 y es por esto que la definición actualmente aceptada de byte es igual a un octeto de bits (si bien esto no fue cierto siempre; hace décadas era relativamente común encontrar procesadores con `CHAR_BIT` igual a 7 o con `CHAR_BIT` igual a 16). El estándar POSIX (que define la parte común de los sistemas basados en UNIX) exige que `CHAR_BIT` sea 8.

El operador `sizeof` regresa el número de `char` que ocupa una variable o tipo. Este operador opera en un contexto no evaluado. Por ejemplo, el siguiente programa únicamente imprimirá 1.

Código

```
#include <iostream>

char f( ) {
   std::cout << "hola\n";
   return '@';
}
int main( ) {
   std::cout << sizeof(f( ));   // imprime sizeof(char), no imprime hola
}
```

En C++ el tipo `char` actúa como un tipo entero. Desafortunadamente, el estándar sólo garantiza que `char` puede almacenar los enteros del 0 al 127 (es decir, 2^7 valores distintos como si `char` fuera de 7 bits). Como la mayoría de las implementaciones usan 8 bits para `char`, algunas usan el bit extra para representar el rango de 0 a 255 y otras para representar el rango de -128 a 127. Es decir, el signo de `char` está definido por la implementación. Los tipos `signed char` y `unsigned char` pueden usarse para los casos específicos en los que se necesite o no se necesite signo.

Las literales de caracter se delimitan entre comillas sencillas (por ejemplo `'@'`) y la implementación usa automáticamente su valor entero según la codificación por omisión, normalmente la ASCII para los caracteres en ese rango. Las literales de cadena son secuencias de caracteres de sólo lectura que pueden contener cero o más caracteres y se delimitan por comillas dobles (por ejemplo `"gato"`). Algunos caracteres especiales se denotan mediante secuencias de escape. Los más importantes son:

Secuencia de escape	**Caracter**
`\0`	Nulo
`\'`	Comilla sencilla
`\"`	Comilla doble
`\\`	Diagonal invertida
`\a`	Campana
`\n`	Salto de línea
`\r`	Retorno de carro
`\t`	Tabulador
`\O{`*nn...*`}`	Caracter con un valor entero expresado mediante dígitos octales
`\X{`*nn...*`}`	Caracter con un valor entero expresado mediante dígitos hexadecimales

Los tipos `char8_t`, `char16_t` y `char32_t` son tipos enteros sin signo pensados en ser usados para representar los caracteres del conjunto Unicode, el cual es mucho más grande que el conjunto ASCII y que actualmente cuenta con aproximadamente 150,000 caracteres. Los tipos `char8_t`, `char16_t` y `char32_t` corresponden con las tres principales codificaciones Unicode llamadas UTF-8, UTF-16 y UTF-32, respectivamente. Mientras que la literal de caracter `"a"` está codificada de forma dependiente de la implementación usando `char`, la literal de caracter `u8"a"` está codificada con UTF-8 usando `char8_t`, la literal de caracter `u'a'` con UTF-16 usando `char16_t` y la literal de caracter `U'a'` con UTF-32 usando `char32_t`. El tipo `wchar_t` es un tipo entero más grande que `char` que se intentó usar para almacenar cualquier símbolo del Conjunto Universal de Caracteres (UCS por sus siglas en inglés) que actualmente es compatible con Unicode. Cuando apenas habían 50,000 símbolos definidos en el UCS, Windows definió `wchar_t` con 16 bits e incluso definió una versión alternativa de `main` llamada `wmain` que recibe cadenas de `wchar_t`. Desafortunadamente, el UCS creció rápidamente y 16 bits se volvieron insuficientes. Linux define `wchar_t` con 32 bits. La literal de caracter `L"a"` es una literal de `wchar_t`. Este tipo debe evitarse en la medida de lo posible y se deben preferir literales en UTF-8.

Tipo de dato	**Número usual de bits**	**Rango usual de valores**	**Prefijo de literal**
`char`	8	[-128, 127] o [0, 255]	
`signed char`	8	[-128, 127]	
`unsigned char`	8	[0, 255]	
`char8_t`	16	[0, 255]	u8
`char16_t`	16	[0, 65535]	u
`char32_t`	32	[0, 4294967295]	U
`wchar_t`	16 o 32	[0, 65535] o [0, 4294967295]	L

El tipo `bool` permite declarar variables binarias que almacenan los valores verdadero o falso. Las palabras reservadas `true` o `false` permiten expresar dichos valores y éstos operan aritméticamente como 1 y 0, respectivamente. Como las variables `bool` deben ser direccionables, `bool` ocupa al menos lo mismo que un `char` y es posible incluso que ocupe más, ya que el estándar sólo garantiza que `sizeof(bool)` es mayor o igual que 1. Afortunadamente, `sizeof(bool)` casi siempre es exactamente 1.

Código	Salida
`bool a = true, b = false;` `std::cout << a << " " << b;`	`1 0`

El tipo `void` denota la ausencia de valor y es ilegal intentar declarar variables de este tipo. Por la misma razón, `sizeof(void)` es ilegal, aunque GCC lo permite y lo evalúa como 1.

El tipo `int` es el principal tipo entero del lenguaje. Este tipo tiene signo y se garantiza que `sizeof(int)` es al menos 2, aunque en la mayoría de las computadoras actuales es 4. Su variante sin signo es el tipo `unsigned int` que puede escribirse simplemente como `unsigned`. El modificador `short` permite declarar enteros potencialmente más chicos que `int`. Este tipo tiene signo, se puede escribir simplemente como `short` y también existe su variante `unsigned short`. Se garantiza que `sizeof(short)` es al menos 2 y normalmente éste es el caso. El modificador `long` permite declarar enteros potencialmente más grandes que `int`. Este tipo tiene signo, se puede escribir simplemente como `long` y también existe su variante `unsigned long`. Se garantiza que `sizeof(long)` es al menos 4 y normalmente es 4 en Windows y 8 en Linux. Finalmente, el modificador `long long` permite declarar enteros potencialmente más grandes que `long`. Este tipo tiene signo, se puede escribir simplemente como `long long` y también existe su variante `unsigned long long`. Se garantiza que `sizeof(long long)` es al menos 8 y normalmente éste es el caso. Los tipos enteros más grandes que `int` tienen sufijos para sus literales. Por ejemplo, `2L` es de tipo `long` y `2LL` es de tipo `long long`. Los sufijos `U` indican versiones sin signo. Por ejemplo, `2U` es de tipo `unsigned`.

Tipo de dato	Número usual de bits	Rango usual de valores	Sufijo de literal
`short`	16	[-32768, 32767]	
`unsigned short`	16	[0, 65535]	
`int`	32	[-2147483648, 2147483647]	
`unsigned int`	32	[0, 4294967295]	`U`
`long`	32 o 64	[-2147483648, 2147483647] o [-9223372036854775808, 9223372036854775807]	`L`
`unsigned long`	32 o 64	[0, 4294967295] o [0, 18446744073709551615]	`LU`
`long long`	64	[-9223372036854775808, 9223372036854775807]	`LL`
`unsigned long long`	64	[0, 18446744073709551615]	`LLU`

Las literales enteras se pueden escribir con comillas simples como separadores de dígitos (por ejemplo `1'234'567`) sin que éstas afecten su significado. Aunque por omisión una literal entera sin sufijo es de tipo `int`, si el valor de la literal no cabe en el rango de `int` entonces se promoverá al tipo más chico con signo que pueda representar tal valor. Por ejemplo, la literal `1'000'000'000'000'000` podrá ser de tipo `int`, `long` o `long long` dependiendo de la implementación. A su vez, las literales que comienzan con `0` se interpretan en base octal (por ejemplo, `010` es igual a `8`), las literales que comienzan con `0x` se interpretan en base hexadecimal (por ejemplo, `0xF` es igual a `15`) y las literales que comienzan con `0b` se interpretan en base binaria (por ejemplo, `0b101` es igual a `5`). Las literales octales deben evitarse.

El archivo de biblioteca `<stddef.h>` define `ptrdiff_t` como un tipo entero con signo capaz de representar la diferencia entre dos apuntadores que estén apuntando al mismo arreglo, y `size_t` como un tipo entero sin signo capaz de representar el número de bytes de cualquier variable que pueda ser declarada, incluyendo arreglos. El sufijo `Z` sirve para declarar literales de tipo `size_t`.

El archivo de biblioteca `<stdint.h>` define las familias de tipos enteros que se listan a continuación. Los tipos `int`n`_t` y `uint`n`_t` son tipos enteros (con y sin signo, respectivamente) de exactamente n bits, donde n puede ser 8, 16, 32 o 64. Los tipos `int_fast`n`_t` y `uint_fast`n`_t` son los tipos enteros más rápidos con al menos n bits. Los tipos `int_least`n`_t` y `uint_least`n`_t` son los tipos enteros más pequeños con al menos n bits. Los tipos `intptr_t` y `uintptr_t` son tipos enteros que tienen una cantidad suficiente de bits para representar una dirección de memoria. Los tipos `intmax_t` y `uintmax_t` son los tipos enteros con mayor rango. Estos últimos tipos normalmente son de 64 bits aún cuando el compilador GCC proporciona tipos enteros no estándar `__int128_t` y `__uint128_t`. En código que deba ser totalmente portable, se

debe preferir usar los tipos de ancho fijo `int`n`_t` y `uint`n`_t`. Desafortunadamente, `std::cout` imprimirá valores de los tipos `int8_t` y `uint8_t` como caracteres, lo cual rara vez es deseable.

Los tipos de punto flotante permiten expresar valores con parte fraccionaria y siempre tienen signo. Una literal en punto flotante se distingue de una literal entera porque la primera tiene punto decimal, o bien, un exponente al estilo de notación científica usando una `e` como introducción al exponente. Por omisión, la base del exponente de las literales escritas en notación científica es 10. Las literales `3.14`, `1.0`, `2e4` y `3.12e-1.2` son todas de punto flotante. Una literal en punto flotante puede comenzar con `0x` y entonces la parte entera debe escribirse en hexadecimal y se usa una `p` como introducción a un exponente escrito en decimal y con base 2. Por ejemplo, `0xFp3` es equivalente a 15×2^3.

El tipo `double` es el principal tipo en punto flotante del lenguaje. Si la implementación soporta el estándar IEEE 754 para aritmética en punto flotante, entonces este tipo debe corresponder con el tipo *binary64* de tal estándar. En el mismo sentido, el tipo `float` debe corresponder con el tipo *binary32* y el tipo `long double` debe corresponder con el tipo *binary128* o con una extensión del tipo *binary64*. Normalmente el tipo `long double` se implementa con 80 bits y no con 128 bits, dado que el formato de 80 bits fue introducido primero en las arquitecturas Intel y se volvió ampliamente popular.

Tipo de dato	Número de bits	Rango de valores	Sufijo de literal
`float`	32	[-3.40282e+038, 3.40282e+038]	`F`
`double`	64	[-1.79769e+308, 1.79769e+308]	
`long double`	80 o 128	[-1.18973e+4932, 1.18973e+4932]	`L`

Finalmente, el tipo `nullptr_t` declarado en `<stddef.h>` es el tipo de la literal de apuntador `nullptr`. Esta literal denota un apuntador nulo, el cual se asume apunta a una dirección inválida de memoria.

4. Arreglos y apuntadores

Un arreglo es una secuencia de variables del mismo tipo. Un arreglo se declara especificando el tipo de los elementos del arreglo, el nombre del arreglo y su tamaño encontre corchetes. El tamaño debe ser un valor entero no negativo y debe conocerse en tiempo de compilación.

Código

```
int a[5];               // arreglo de 5 int
int n;
std::cin >> n;
int b[n];               // error: tamaño desconocido en tiempo de compilación
```

Una vez decladaro el arreglo, sus elementos se numeran o indizan implícitamente a partir de 0 y se pueden nombrar individualmente especificando su índice, también entre corchetes.

Código	Salida

`int a[3];`
`a[0] = 3;`
`a[1] = 7;`
`a[2] = 5;`
`std::cout << a[1];`
	`7`

Los elementos de un arreglo aparecen consecutivos en memoria, uno tras otro. Eso significa que el `sizeof` de un arreglo es igual al `sizeof` del tipo de los elementos multiplicado por el tamaño de arreglo.

Código	Salida

`int a[3];`
`std::cout << sizeof(int) << "\n";`
`std::cout << sizeof(a) << "\n";`
	`4`
	`12`

Para acceder a un elemento de un arreglo, el indice puede tomarse del valor de una variable entera. Se debe evitar intentar acceder a un elemento inexistente en un arreglo usando un índice fuera de rango.

Código	Salida

`int a[3], i;`
`i = 0;`
`a[i] = 5;  // a[0] = 5;`
`i = 1;`
`a[i] = 7;  // a[1] = 7;`
`i = 2;`
`a[i] = 9;  // a[2] = 9;`
`std::cout << a[0] << " " << a[1]`
`  << " " << a[2];`
	5 7 9

Un arreglo puede inicializarse especificando sus valores entre llaves. Si el arreglo es más grande que la cantidad de elementos listados entre llaves, el resto de los elementos recibe el valor 0 del tipo. El tamaño de un arreglo se puede omitir cuando se inicializa y se infiere de la cantidad de elementos de la lista:

Código	Salida

`int a[4] = { 5, 2 };`
`std::cout << a[0] << " " << a[1]`
`  << a[2] << " " << a[3];`
`int b[] = { 1, 2, 3 };  // tamaño 3`
	5 2 0 0

Desafortunadamente, el lenguaje C++ no permite copiar los elementos de un arreglo directamente a otro arreglo mediante una asignación, aunque sea del mismo tamaño:

Código
`int a[3] = { 5, 5, 5 };` `int b[3] = a;  // error: un arreglo simplemente no se puede copiar así`

Una matriz es un arreglo multidimensional. En principio, no hay límite en el número de dimensiones.

Código	Salida

`int a[2][4] = {`
`  { 3, 6, 9, 5 },`
`  { 7, 1, 2, 8 }`
`};`
`std::cout << a[1][3];`
	8

Los elementos de una matriz con un índice i en una de las dimensiones aparecen antes en memoria que los elementos que tienen un índice mayor que i en la misma dimensión. Es posible especificar variables sencillas, arreglos y matrices (todas con elementos del mismo tipo) en la misma declaración.

Código
`int a, b[5], c[2][3];  // entero, arreglo de enteros y matriz de enteros`

En C++, toda variable con `sizeof` mayor o igual a 1 tiene asignada una dirección de memoria, la cual además es única con respecto a la de cualquier otra variable del mismo tipo accesible en ese punto del programa. Cuando una variable ocupa más de un `char` en memoria, su dirección es la dirección de su primer `char`. La dirección de memoria de una variable puede obtenerse mediante el operador `&` prefijo y, si la variable es de tipo `T`, ésta puede almacenarse en una apuntador de tipo `T*`. Una computadora de 32 bits con bytes direccionables sólo se puede direccionar 2^{32} bytes distintos, por lo que `sizeof(T*)` es de 4 bytes. En contraste, en una computadora de 64 bits ocurre que `sizeof(T*)` es de 8 bytes. Lo anterior es independiente del tipo `T`, ya que un apuntador siempre almacena una dirección de memoria.

Código	Salida

`char c;`
`char* pc = &c;`
`std::cout << sizeof(c) << "\n";`
`std::cout << sizeof(pc) << "\n";`
	1
	8

La característica más importante de los apuntadores es su capacidad de desreferencia. El operador prefijo `*` sobre un apuntador permite visitar la variable de la que conocemos su dirección. Por ejemplo:

Código	Salida

`int n = 999;`
`int* p = &n;`
`std::cout << *p << "\n";  // n mediante p`
`*p = 5;  // n mediante p`
`std::cout << n << "\n";`
	`999`
	`5`

Si reasignamos un apuntador a una nueva dirección, ahora se referirá a otra variable:

Código	Salida

`int a = 0, b = 1;`
`int* p = &a;`
`std::cout << *p << "\n";  // a mediante p`
`p = &b;`
`std::cout << *p << "\n";  // b mediante p`
	`0`
	`1`

A pesar de que `T*` es un tipo apuntador, se pretende que `*` sea un modificador sobre el nombre de la variable. Eso hace que la declaración de múltiples apuntadores sea poco intuitiva.

Código

`int* a, b;  // un apuntador a entero y un entero ordinario`
`int* c, *d;  // dos apuntadores a entero`

Cuando un apuntador apunta a algún elemento de un arreglo, podemos usar los operadores `++` y `--` para movernos de un elemento al siguiente o al anterior en el arreglo. El lenguaje conoce el `sizeof(T)` de los elementos para adecuar correctamente la dirección almacenada en el apuntador.

Código	Salida

`int a[2] = { 5, 7 };`
`int* p = &a[0];`
`std::cout << *p << " ";  // a[0]`
`++p;  // avanzamos`
`std::cout << *p << " ";  // a[1]`
`++p;  // avanzamos`
`std::cout << *p << " ";  // a[2]`
`--p;  // retrocedemos`
`std::cout << *p << " ";  // a[1]`
	`5 7 8 7`

También están definidos los operadores `+=` y `-=` para avanzar o retroceder varios elementos en un paso.

Código	Salida

`int a[3] = { 5, 7, 8 };`
`int* p = &a[0];`
`p += 2;  // avanzamos`
`std::cout << *p;  // a[2]`
	`8`

Los operadores `+` y `-` siguen la semántica de los operadores `+=` y `-=` en el sentido de que producen un apuntador que es el resultado de la suma o de la resta, pero sin modificar el apuntador original.

Código	Salida

`int a[3] = { 5, 7, 8 };`
`int* p = &a[0];`
`std::cout << *(p + 2) << "\n";  // a[2]`
`std::cout << *p << "\n";  // a[0]`
	`8`

Dos apuntadores se pueden comparar para saber si apuntan a la misma dirección (`p1 == p2`) o para saber si un apuntador apunta a una dirección que esté a la izquierda de otra (`p1 < p2`). El resto de los operadores relacionales `!=`, `>`, `<=` y `>=` también están definidos.

Código	Salida

`int a[3] = { 9, 9, 9 };`
`int* p1 = &a[0];`
`int* p2 = &a[0];`
`int* p3 = &a[2];`
`std::cout << (p1 == p2) << "\n";`
`std::cout << (p1 == p3) << "\n";`
`std::cout << (p1 < p3) << "\n";`
	`1`
	`0`
	`1`

En ese sentido, se debe recalcar la diferencia entre comparar apuntadores y comparar los valores de las variables que están siendo apuntadas. Por ejemplo:

Código	Salida

`int a = 2, b = 2;`
`int* pa = &a, *pb = &b;`
`// ¿misma dirección apuntada?`
`std::cout << (pa == pb) << "\n";`
`// ¿mismos valores apuntados?`
`std::cout << (*pa == *pb) << "\n";`
	`0`
	`1`

La resta de apuntadores `p2 - p1` es igual a la cantidad de veces que debemos avanzar el apuntador `p1` para que apunte a la misma dirección de `p2`.

Código	Salida

`int a[10];`
`int* p1 = &a[2], *p2 = &a[7];`
`std::cout << p2 - p1 << "\n";`
	`5`

La asignación de un arreglo sobre un apuntador sorprendentemente compila. En realidad el arreglo no se copia, sino que el apuntador recibe la dirección del primer elemento del arreglo.

Código

```
int a[5];
int* p = &a[0];             // ok, p apunta al primer elemento del arreglo
int* q = a;                 // ok pero feo, equivalente a la línea anterior
```

5. Consulta de tipos, inferencias y alias

En C++, el operador `decltype` sirve para consultar el tipo de una expresión o variable. Al igual que `sizeof`, éste opera en un contexto no evaluado. Es posible usar el resultado de `decltype` en declaraciones.

Código

```
#include <iostream>

char f( ) {
   std::cout << "hola\n";
   return '@';
}
int main( ) {
   int n;
   decltype(n) m;               // int m;
   decltype('@') c;             // char c;
   decltype(f( )) d;            // char d; (no imprime hola)
}
```

La palabra `auto` permite declarar una variable con inicializador usando el tipo del inicializador.

Código

```
int n;
auto m = n;                    // int m = n;
auto c = '@';                  // char c = '@';
auto k;                        // error: no hay inicializador
```

Desafortunadamente, `auto` debe inferir el mismo tipo para todas las varias variables de la misma declaración. Esto puede resultar útil en el caso de apuntadores, pero no sirve para inferir arreglos.

Código

```
auto a = 0, b = 3.14;          // error, inferencia de tipos inconsistente
auto c = 0; auto d = 3.14;     // ok, declaraciones distintas
auto e[] = { 1, 2, 3 };        // error
auto p1 = &a, p2 = &a;         // ok, ambas variables son int*
```

Si se incluye `<initializar_list>` entonces una variable inicializada mediante una lista de expresiones (todas del mismo tipo) escritas dentro de llaves recibirá el tipo `std::initializer_list<T>` donde `T` es el tipo de las expresiones de la lista. Desafortunadamente, la lista no actúa como un arreglo.

Código

```
#include <initializer_list>

int main( ) {
   auto v = { 1, 2, 3 };       // std::initializer_list<int> v = { 1, 2, 3 };
   int n = v[0];               // error, v no es un arreglo
}
```

Una variable declarada con el calificador `const` debe estar inicializada y su valor no puede modificarse posteriormente. Es decir, `const` denota que la variable en realidad es de sólo lectura.

Código

```
const int n = 5;
n = 7;                         // error, intento de modificación

int m;
std::cin >> m;
const int k = m;               // ok, m puede modificarse pero k no
```

El calificador `const` forma parte del tipo de la variable y se propaga para evitar que ocurran modificaciones accidentales a las variables `const`. Por ejemplo, un apuntador a una variable de tipo `T` es de tipo `T*`, pero un apuntador a una variable de tipo `const T` es de tipo `const T*`.

Código

```
int n = 5;
auto p = &n;                   // int* p = &n;
*p = 6;                        // ok, p puede modificar n

const int m = 5;
auto q = &m;                   // const int* q = &m;
*q = 6;                        // error, q no puede modificar m
```

Una variable declarada con el especificador `constexpr` debe estar inicializada con un valor conocido en tiempo de compilación y la variable además se vuelve `const`. Las expresiones que el compilador evalúa en tiempo de compilación también son llamadas expresiones constantes.

Código

```
constexpr int n = 5;
n = 7;                          // error, intento de modificación

int m;
std::cin >> m;
constexpr int k = m;            // error, m no es una expresión constante
```

Una variable declarada con el especificador `constinit` debe estar inicializada con una expresión constante, pero la variable no se vuelve `const`. Este especificador no puede usarse en variables locales.

Código

```
constinit int n = 5;

int main( ) {
   n = 7;                       // ok, la variable no es const
}
```

Una variable declarada con el calificador `volatile` es una variable en donde todos los accesos a la misma deben hacerse mediante en accesos a su localidad de memoria. El compilador debe suponer que la variable puede ser modificada por factores no visibles al programa (normalmente dispositivos de hardware mapeados en memoria o depuradores) y que, por lo mismo, no debe intentar aplicar optimizaciones obvias. El calificador `volatile` también forma parte del tipo de la variable.

Código

```
volatile int n = 5;
std::cout << n;                 // no necesariamente equivale a std::cout << 5;

auto p = &n;                    // volatile int* p = &n;
volatile const int m = 7;       // raro pero válido, la variable sólo puede ser
                                // modificada por factores externos al programa
```

La sentencia `typedef T` *identificador* `;` crea un sinónimo del tipo `T` bajo el nombre del identificador dado. En C++, la notación `using` *identificador* `= T;` tiene el mismo efecto.

Código

```
typedef int entero;
using flotante = double;

entero n = 5;                   // int n = 5;
flotante x = 3.14;              // double x = 3.14;
```

6. Conversiones, moldeados e inicialización

Por retrocompatibilidad con C, C++ permite algunas conversiones implícitas que pierden de información. En una conversión de un tipo entero a otro de menor rango, el valor del tipo destino se obtiene descartando los bits más significativos del valor original. En una conversión de un tipo flotante a otro de menor rango o entre un tipo entero a un tipo flotante, se permite redondeo a un valor cercano representable. En una conversión de un tipo flotante a un tipo entero, la parte fraccionaria se pierde.

Código	**Salida**

`int n = 3.14;`
`unsigned short m = 65536;`
`float f = 1'234'567'890;`
`std::cout << n << " " << m << " " << f;`
	`3 0 1.23457e+09`

Los tipos `float`, `double` y `long double` suelen tener una mantisa de 24, 53 y 64 bits, respectivamente. Se debe tener cuidado al almacenar enteros de 32 o 64 bits en estos tipos cuando la mantisa no tiene suficientes bits para representar el valor entero sin pérdida.

Cuando dos valores participan en una operación aritmética, se aplican las siguientes conversiones implícitas (también llamadas *promociones estándar*) en orden de importancia:

1. Si uno de los operandos es `long double`, el cálculo se realiza en `long double`.
2. Si uno de los operandos es `float` o `double`, el cálculo se realiza en `double`. Nótese que el cálculo se realiza en `double` aún si todos los operandos son tipo `float`.
3. Si uno de los operandos es `unsigned long long`, el cálculo se realiza en `unsigned long long`.
4. Si uno de los operandos es `long long` y el otro operando es un entero sin signo, el cálculo se realiza en `long long` si éste puede representar todos los valores de ambos operandos. Si esto último no se cumple, el cálculo se realiza en `unsigned long long`
5. Si uno de los operandos es `unsigned long`, el cálculo se realiza en `unsigned long`.
6. Si uno de los operandos es `long` y el otro operando es un entero sin signo, el cálculo se realiza en `long` si éste puede representar todos los valores de ambos operandos. Si esto último no se cumple, el cálculo se realiza en `unsigned long`.
7. Si uno de los operandos es `int` y el otro operando es un entero sin signo, el cálculo se realiza en `int` si éste puede representar todos los valores de ambos operandos. Si esto último no se cumple, el cálculo se realiza en `unsigned`.
8. En cualquier otro caso, el cálculo se realiza en `int`. Nótese que esto ocurre aún si todos los operandos son del mismo tipo pero son más pequeños que `int`.

A continuación se muestra un ejemplo de código que muestra las consecuencias más importantes de las promociones estándar en cálculos aritméticos usuales.

Código	**Salida**

`// cuidado, posible sobreflujo`
`std::cout << 1000000000 * 10 << "\n";`
 	
`// ok, cálculo en long long`
`std::cout << 1000000000 * 10LL << "\n";`
 	
`// raro pero esperado: cálculo en int`
`std::cout << 'A' + 0 << "\n";`
 	
`// raro e inesperado: cálculo en int`
`std::cout << 'A' + '\0' << "\n";`
	1410065408
	10000000000
	65
	65

El lenguaje C++ proporciona distintas formas de llevar a cabo conversiónes explícitas, también llamada moldeados o *casts* por su traducción al inglés.

- La notación `(T)valor` es la notación de moldeado al estilo C y permite llevar a cabo conversiones entre tipos relacionados como enteros y flotantes, entre tipos no relacionados como enteros y apuntadores o como apuntadores de distintos tipos, así como agregar o quitar `const`.
- La notación `T(valor)` es la notación funcional al estilo C++ y es tan poderosa como la de C, excepto que `T` debe ser un tipo escrito mediante una única palabra. Es decir, `(unsigned short)1` es válido pero `unsigned short(1)` no lo es.
- El operador `static_cast<T>(valor)` es exclusivo de C++, sólo permite llevar a cabo conversiones entre tipos relacionados y no permite quitar `const`.

- El operador `const_cast<T>(valor)` es exclusivo de C++ y sólo permite agregar o quitar `const`.
- El operador `reinterpret_cast<T>(valor)` es exclusivo de C++ y permite llevar a cabo conversiones entre tipos no relacionados y permite agregar pero no quitar `const`.

Código	Salida

`std::cout << (int)1e9 * (long long)10`
`<< "\n"`
`<< int(1e9) * int64_t(10)`
`<< "\n"`
`<< static_cast<int>('@');`
	`10000000000`
	`10000000000`
	`64`

Los operadores `static_cast<T>(valor)`, `const_cast<T>(valor)` y `reinterpret_cast<T>(valor)` fueron diseñados para ser notacionalmente incómodos, de modo de que el programador dude sobre la necesidad de usarlos. Irónicamente, los moldeados al estilo C y los moldeados funcionales son los más fáciles de escribir y también los más poderosos (y por lo tanto, peligrosos).

Desafortunadamente, en C++ hay demasiadas notaciones para inicializar una variable. A continuación se listan las formas de inicializar un `int` con 0 y también un otro valor distinto de 0.

Código	Código

`// inicializar con 0`
`int a = 0;`
`int b = { 0 };`
`int c(0);`
`int d{0};`
 	
`int e = int( );`
`int f = { };`
`int g{ };`
	`// inicializar con 5`
	`int a = 5;`
	`int b = { 5 };`
	`int c(5);`
	`int d{5};`

La diferencia principal se da entre las notaciones que usan llaves y las que no las usan. Las notaciones que usan llaves prohiben conversiones implícitas que pierdan información.

Código	
`int n = 3.14;`	`// ok, conversión implícita con pérdida`
`int m = { 3.14 };`	`// error`
`int k{3.14};`	`// error también`
`int h{int(3.14)};`	`// ok, la conversión explícita ocurre antes`

7. Sentencias de control local de flujo

Esta sección supone que el lector ya está familiarizado con las sentencias de control de C, por lo que éstas se describen rápidamente a manera de resumen. Sin embargo, sí se incluyen las variantes exclusivas de C++ y también se incluyen puntos importantes a considerar de cada una.

La sentencia `if` es una sentencia que ejecuta su bloque cuando su condición es verdadera. Un valor puede usarse en un contexto booleano si su tipo es entero, flotante, booleano, apuntador o si existe una conversión implícita a alguno de estos tipos. En tal caso, un valor se considera verdadero si es distinto al valor 0 del tipo y se considera falso en otro caso.

Código	Salida

`if (true) {`
`  std::cout << "dentro1\n";`
`}`
`if (5) {`
`  std::cout << "dentro2\n";`
`}`
	`dentro1`
	`dentro2`

Las llaves de un bloque pueden omitirse si éste contiene una única instrucción. Sin embargo, no se recomienda omitir las llaves salvo que la instrucción aparezca en la misma línea, ya que es un error común dejarse engañar por la indentación del código.

Código	Salida

`if (true) std::cout << "dentro1\n";`
`if (false)`
`  std::cout << "dentro2\n";`
`  std::cout << "no en el if\n";`
	`dentro1`
	`no en el if`

La sentencia `else` sirve para ejecutar una instrucción o bloque cuando la condición es falsa.

Código	Salida

`if (false) {`
`  std::cout << "en if\n";`
`} else {`
`  std::cout << "en else\n";`
`}`
	`en else`

En C++ es posible declarar variables antes de la condición de un `if`. Esta característica existe porque es muy común que la condición dependa de una variable que deja de tener relevancia después del `if`.

Código

```
if (int v = f( ); v != 0) {
   std::cout << v << "\n";
}
std::cout << v;             // error, v ya no existe en este punto
```

La sentencia `switch` permite controlar la ejecución de un bloque con base en el valor de una variable entera. La palabra reservada `case` permite especificar un valor y el código se ejecutará a partir de ahí si el valor control coincide con el valor del `case`. En caso de que ningún `case` coincida, el `switch` ejecutará el código a partir de la sección `default`, la cual es opcional.

Código	Salida

`switch (1) {`
`case 0:`
`  std::cout << 0;`
`case 1:`
`  std::cout << 1;`
`case 2:`
`  std::cout << 2;`
`default:`
`  std::cout << "X";`
`}`
	`12X`

No es posible indicar varios valores para el mismo `case`. Se deben especificar varios `case` en su lugar.

Código	Salida

`switch (1) {`
`case 0:`
`case 1:`
`case 2:`
`  std::cout << "cero, uno o dos";`
`}`
	`cero, uno o dos`

El comportamiento de ejecución *en cascada* de un `switch` resulta contraintuitivo en primera instancia. La sentencia `break` permite romper la ejecución del bloque de un `switch`.

Código	Salida

`switch (1) {`
`case 0:`
`  std::cout << 0;`
`  break;`
`case 1:`
`  std::cout << 1;`
`  break;`
`case 2:`
`  std::cout << 2;`
`  break;`
`default:`
`  std::cout << "X";`
`}`
	`1`

Como el `switch` está formado por un único bloque, es notacionalmente posible (y semánticamente incorrecto) saltarse la inicialización de una variable local.

Código

`int n;`
`std::cin >> n;`
`switch (n) {`
`case 0:`
`  int v = 5;`
`case 1:  // error: el case 1 se salta la inicialización de x`
`  std::cout << v;`
`}`

En C++ es posible declarar una variable antes de especificar el control del `switch`.

Código	Salida

`switch (int n = 1; n) {`
`case 0:`
`  std::cout << 0;`
`case 1:`
`  std::cout << "uno";`
`}`
	`uno`

Los ciclos `while` y `do while` son los mismos que en C y en C++ no tienen ninguna extensión. El ciclo `while` permite ejecutar un bloque de código mientras la condición sea verdadera. El ciclo `do while` es similar, pero revisa la condición hasta después de ejecutar el bloque al menos una vez.

Código	Salida

`int i = 1;`
`while (i <= 3) {`
`  std::cout << i << "\n";`
`  i += 1;`
`}`
`do {`
`  std::cout << i << "\n";`
`} while (i > 10);`
	`1`
	`2`
	`3`
	
	`4`

Cabe mencionar que es fácil olvidar el punto y coma que debe aparecer después de la condición del ciclo `do while`, ya que ésta es visualmente irrelevante como separador al existir paréntesis alrededor de la condición. A su vez, una variable declarada dentro de un ciclo `do while` no puede ser usada dentro de la condición del ciclo, debido a que el ámbito de la variable termina en el bloque.

Código

```
do {                    // intención: leer enteros hasta encontrar un 0
   int n;               // tendrá que declararse fuera
   std::cin >> n;
} while (n != 0);       // error, n ya no existe si se declara dentro del bloque
```

El lenguaje C++ tiene dos tipos de ciclos `for`, siendo el primero el de C. Las tres partes del control del ciclo `for` de C son la inicialización, la condición y la actualización, apareciendo separadas por puntos y comas y siendo todas opcionales. En particular, un ciclo sin condición es un ciclo infinito.

Código

```
for (int i = 0; i < 5; ++i) {               // uso típico (válido declarar
   std::cout << i << " ";                   // variables en la inicialización)
}
for (int i = 0, j = 4; i < 5; ++i, --j) {   // múltiple acciones en la
   std::cout << i << " " << j << " ";       // inicialización y actualización
}
for (;;) {                                  // ciclo infinito
   std::cout << "hola";
}
```

La segunda variante del ciclo `for` es el `for` de rango, el cual sirve para iterar sobre los valores de una secuencia sin mencionar sus posibles índices. Este ciclo también puede iterar sobre un `std::initializer_list`.

Código

```
for (int v : { 2, 4, 8 }) {     // en cada iteración, v toma el valor
   std::cout << v << " ";       // de un elemento de la secuencia
}
std::cout << "\n";
```

El ciclo `for` de rango permite declarar una variable previamente a la especificación del control del ciclo.

Código

```
for (auto seq = { 2, 4, 8 }; int v : seq) {
   std::cout << v << " ";       // equivalente al ejemplo anterior
}
std::cout << "\n";
```

Dentro de un ciclo, la sentencia `break` termina dicho ciclo y la sentencia `continue` termina la iteración actual, pero no el ciclo. Un `continue` actúa de forma distinta en un `while` o `do while` que en un `for`. En el `while` o `do while`, la ejecución salta a la condición, pero en el ciclo `for` de C la ejecución salta a la actualización y luego a la condición, mientras que en un `for` de rango la ejecución salta al siguiente elemento de la secuencia. En caso de haber varios ciclos anidados, el efecto ocurre sobre el ciclo interno.

Código	Salida

`for (int i = 1; i <= 3; ++i) {`
`  for (int j = 1; j <= 3; ++j) {`
`  if (j == 1) {`
`  continue;`
`  } else if (i == 2 && j == 2) {`
`  break;`
`  }`
`  std::cout << i << " "`
`  << j << "\n";`
`  }`
`}`
	1 2
	1 3
	3 2
	3 3

La sentencia `goto` permite que la ejecución salte directamente hacia la etiqueta designada. Una etiqueta se declara con un identificador seguido de dos puntos. El uso de `goto` constituye una forma aceptable de salir de ciclos anidados, pero debe evitarse su uso para controlar la ejecución del programa de forma arbitraria, especialmente cuando una sentencia estructurada tiene el mismo efecto.

Código	Salida

`for (int i = 1; i <= 3; ++i) {`
`  for (int j = 1; j <= 3; ++j) {`
`  if (i == 2 && j == 2) {`
`  goto fin;`
`  }`
`  std::cout << i << " "`
`  << j << "\n";`
`  }`
`}`
`fin: std::cout << "terminamos";`
	`1 1`
	`1 2`
	`1 3`
	`2 1`
	`terminamos`

De forma no estándar, el compilador GCC acepta guardar direcciones de etiquetas en apuntadores `void*` y saltar a ellas posteriormente. La dirección de una etiqueta se obtiene con el operador prefijo `&&` para enfatizar que esta característica es una extensión del lenguaje. Esto suele ser muy usado en la implementación de intérpretes de alto rendimiento.

Código	Salida

`void* p = &&eti;`
`goto *p;`
`eti: std::cout << "aqui";`
	`aqui`

8. Tiempos de vida, ámbitos y referencias

Por omisión, las variables de un programa tienen almacenamiento o tiempo de vida *automático*. Las variables locales de una función normalmente tienen este tiempo de vida. El tiempo de vida de una variable automática termina junto con el bloque de código en el que está declarada. Si bien los bloques de código comúnmente están asociados a cuerpos de función, cuerpos de sentencias de decisión o de ciclos, también es posible especificar bloques anónimos delimitados entre llaves.

Código

```
int main( ) {
   int a = 1;       // el tiempo de vida terminará con el bloque de la función
   if (true) {
      int b;        // el tiempo de vida terminará con el bloque del if
   }
   {
      int c;        // el tiempo de vida terminará con el bloque anónimo
   }
   std::cout << a;        // ok
   std::cout << b << c;   // error, el tiempo de vida de las variables terminó
}
```

El valor de variables automáticas de tipos fundamentales que no estén inicializadas está indefinido.

Código	Salida

`#include <iostream>`
 	
`int main( ) {`
`  int n;`
`  std::cout << n;`
`}`
	`-5632673`

Las variables declaradas `static` dentro de una función con el especificador tienen tiempo de vida *estático*. Estas variables sólo se inicializan la primera vez que la ejecución pasa por la declaración y retienen su valor al término de la llamada a función, prolongando su tiempo de vida hasta que el programa termina. El valor de variables estáticas de tipos fundamentales no inicializadas es el valor 0 del tipo.

Código	Salida

`#include <iostream>`
 	
`void f( ) {`
`  int a = 5;`
`  static int b = 1;`
`  static int c;`
`  std::cout << a << b << c << "\n";`
`  a += 1, b += 1, c += 1;`
`}`
 	
`int main( ) {`
`  f( ), f( ), f( ), f( );`
`}`
	510
	521
	532
	543

La visibilidad de una variable puede ser distinta a su tiempo de vida. Por ejemplo, una variable puede estar oculta por otra o puede estar fuera del bloque donde es visible. En este contexto, a la porción del código en el que un identificador es visible se le denomina ámbito.

Código

```
void f( ) {
   int a = 1;
   if (true) {
      int a = 2;
      std::cout << a;        // 2 (se usa la variable interna)
      static int c = 5;      // sobrevive a la función
   }
   c = 7;                    // error, c no visible (aunque sigue viva)
}
```

Una variable global es una que está declarada fuera de toda función. Su tiempo de vida comienza con el inicio del programa y termina con el final del mismo. El valor de variables globales de tipos fundamentales que no estén inicializadas es el valor 0 del tipo. El operador de resolución de ámbito `::` permite accesar a una variable que se encuentre oculta por una variable local.

Código	Salida

`#include <iostream>`
 	
`int g1 = 5, g2;`
 	
`int main( ) {`
`  std::cout << g1 << g2 << "\n";`
`  int g2 = 7;`
`  std::cout << g1 << g2 << "\n";`
`  std::cout << g1 << ::g2 << "\n";`
`}`
	50
	57
	50

Un espacio de nombres es un ámbito que puede declararse globalmente o que puede declararse dentro de otro espacio de nombres. Los símbolos declarados dentro de un espacio de nombres pueden ser referidos desde fuera del mismo usando como prefijo el nombre del espacio y el operador de resolución de ámbito.

Código	Salida

`#include <iostream>`
 	
`int a = 2;`
`namespace ns {`
`  int a = 5;`
`  namespace interno {`
`  int a = 7;`
`  }`
`}`
 	
`int main( ) {`
`  std::cout << a << "\n";`
`  std::cout << ::a << "\n";`
`  std::cout << ns::a << "\n";`
`  std::cout << ns::interno::a << "\n";`
`}`
	2
	2
	5
	7

Una referencia es un nombre alternativo para una variable ya existente. La referencia a una variable de tipo `T` se declara con el tipo `T&` seguido de un identificador. Una referencia siempre debe inicializarse.

Código	Salida

`int n = 0;`
`int& m = n;  // m refiere a n`
`m = 5;`
`std::cout << n;`
	5

Es inválido declarar referencias `T&` a temporales de tipo `T`. Sin embargo, es válido declarar referencias de sólo lectura `const T&` tanto a variables como a temporales. En este último caso, el tiempo de vida del temporal se amplía para que coincida con el tiempo de vida de la referencia.

Código

`int& r1 = 5;  // error, referencia a temporal`
`const int& r2 = 5;  // ok, se amplía el tiempo de vida del temporal`
 	
`int v = 0;`
`int& r3 = v;  // ok, referencia ordinaria`
`const int& r4 = v;  // ok, pero no se puede modificar v usando r4`
`int& r5 = const_cast<int&>(r4);  // ok, v no es originalmente const`

Las referencias pueden usarse como parámetros de función para lograr que un identificador haga referencia a una variable que no es local a esa función. Esto permite implementar el paso por referencia.

Código	Salida

`#include <iostream>`
 	
`void f(int& a) {`
`  a += 1;`
`}`
 	
`int main( ) {`
`  int n = 0;`
`  f(n);`
`  std::cout << n;`
`}`
	1

Es posible que una función regrese una referencia. Sin embargo, es responsabilidad del programador asegurarse de que la variable referenciada tenga un tiempo de vida más amplio que el de sus referencias.

Código

```
int& f( ) {
   int n = 5;
   return n;                    // mal, la variable morirá al terminar f
}
int& g( ) {
   static int n = 7;
   return n;                    // ok, la variable seguirá viva
}

int main( ) {
   int& r1 = f( );
   std::cout << r1 << "\n";     // comportamiento indefinido
   int& r2 = g( );
   std::cout << r2 << "\n";     // ok, la variable referida sigue viva
}
```

9. Enumeradores, estructuras y uniones

Un `enum` es una construcción del lenguaje que permite declarar constantes de un tipo entero. Las constantes declaradas reciben valores incrementales consecutivos que por omisión comienzan en cero.

Código	Salida

`enum ejemplo {`
`  A, B, C`
`};`
 	
`std::cout << A << "\n" << B << "\n"`
`  << C << "\n";`
	`0`
	`1`
	`2`

Es posible especificar el valor de las constantes. La asignación manual de valores reinicia la secuencia a partir de cada valor especificado. Esto vuelve posible que aparezcan constantes con valores duplicados.

Código	Salida

`enum ejemplo {`
`  A = 3, B, C, D = 1, E, F`
`};`
 	
`std::cout << A << "\n" << B << "\n"`
`  << C << "\n" << D << "\n"`
`  << E << "\n" << F << "\n";`
	`3`
	`4`
	`5`
	`1`
	`2`
	`3`

Un `enum` también se vuelve un tipo entero definido por el usuario. Existen algunas conversiones implícitas y explícitas entre tipos `enum` y tipos enteros fundamentales.

Código	Salida

`enum dia {`
`  LUN, MAR, MIE, JUE, VIE, SAB, DOM`
`};`
 	
`int v = LUN;`
`dia d = MIE, e = dia(2);`
`std::cout << v << " " << d << " "`
`  << e << "\n";`
	`0 2 2`

Por omisión, el `sizeof` de un `enum` será (en orden de preferencia) igual al de `int`, `unsigned int`, `long`, `unsigned long`, `long long` u `unsigned long long`, dependiendo de cuál es el primero de los tipos que puede representar todas las constantes del `enum`. En la declaración de un `enum` es posible especificar manualmente el tipo entero `T` a usar con la notación `: T` después del nombre del `enum`.

Código	Salida

`enum dia : uint8_t {`
`  LUN, MAR, MIE, JUE, VIE, SAB, DOM`
`};`
 	
`std::cout << sizeof(dia);`
	`1`