

P A R T E 1

Programación básica

- Introducción a C++
- Elementos del lenguaje
- Estructura de un programa
- Entrada y salida estándar
- Sentencias de control
- Tipos estructurados de datos
- Punteros, referencias y gestión de la memoria
- Más sobre funciones

3e2ff75c30d222a35aca2773f3e6e40d
ebruary

3e2ff75c30d222a35aca2773f3e6e40d
ebruary

3e2ff75c30d222a35aca2773f3e6e40d
ebruary

3e2ff75c30d222a35aca2773f3e6e40d
ebruary

CAPÍTULO 1

© F.J. Ceballos/RA-MA

INTRODUCCIÓN A C++

Un *programa* no es nada más que una serie de instrucciones dadas al ordenador en un lenguaje entendido por él, para decirle exactamente lo que queremos que haga. Si el ordenador no entiende alguna instrucción, lo comunicará generalmente mediante mensajes visualizados en la pantalla.

Un programa tiene que escribirse en un lenguaje entendible por el ordenador. Desde el punto de vista físico, un ordenador es una máquina electrónica. Los elementos físicos (memoria, unidad central de proceso, etc.) de que dispone el ordenador para representar las instrucciones y los datos son de tipo binario; esto es, cada elemento puede diferenciar dos estados (dos niveles de voltaje). Cada estado se denomina genéricamente *bit* y se simboliza por *0* ó *1*. Por lo tanto, para representar y manipular información numérica, alfabética y alfanumérica se emplean cadenas de *bits*. Según esto, se denomina *byte* a la cantidad de información empleada por un ordenador para representar un carácter; generalmente un *byte* es una cadena de ocho *bits*. Esto hace pensar que escribir un programa utilizando ceros y unos (lenguaje máquina) llevaría mucho tiempo y con muchas posibilidades de cometer errores. Por este motivo, se desarrollaron los lenguajes de programación.

Para traducir un programa escrito en un determinado lenguaje de programación a lenguaje máquina (código binario), se utiliza un programa llamado *compilador* que ejecutamos mediante el propio ordenador. Este programa tomará como datos nuestro programa escrito en un lenguaje de alto nivel, por ejemplo en C++, y dará como resultado el mismo programa pero escrito en lenguaje máquina, lenguaje que entiende el ordenador.



¿POR QUÉ APRENDER C++?

Una de las ventajas de C++ es su independencia de la plataforma en lo que a código fuente se refiere. Otra característica importante de C++ es que es un lenguaje que soporta diversos estilos de programación (por ejemplo, la programación genérica y la programación orientada a objetos –POO– de la cual empezaremos a hablar en este mismo capítulo). Todos los estilos se basan en una verificación fuerte de tipos y permiten alcanzar un alto nivel de abstracción.

C++ está organizado de tal forma que el aprendizaje del mismo puede hacerse gradualmente obteniendo beneficios prácticos a largo de este camino. Esto es importante, porque podemos ir produciendo proporcionalmente a lo aprendido.

C++ está fundamentado en C lo que garantiza que los millones de líneas de código C existentes puedan beneficiarse de C++ sin necesidad de reescribirlas. Evidentemente, no es necesario aprender C para aprender C++, lo comprobará con este libro. No obstante, si conoce C, podrá comprobar que C++ es más seguro, más expresivo y reduce la necesidad de tener que centrarse en ideas de bajo nivel.

C++ se utiliza ampliamente en docencia e investigación porque es claro, realista y eficiente. También es lo suficientemente flexible como para realizar los proyectos más exigentes. Y también es lo suficientemente comercial como para ser incorporado en el desarrollo empresarial.

Existen varias implementaciones de C++, de distribución gratuita; por ejemplo, GCC. Las siglas GCC significan *GNU Compiler Collection* (colección de compiladores GNU; antes significaban *GNU C Compiler*: compilador C GNU). Como su nombre indica es una colección de compiladores y admite diversos lenguajes: C, C++, Objective C, Fortran, Java, etc. Existen versiones para prácticamente todos los sistemas operativos y pueden conseguirse en gcc.gnu.org.

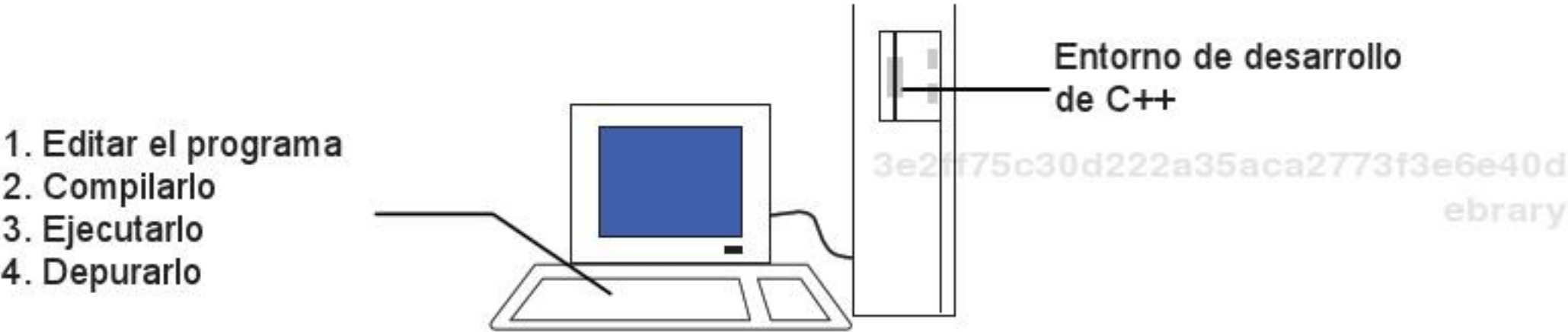
GNU (se trata de un acrónimo recursivo para “Gnu No es Unix”) es un proyecto que comenzó en 1984 para desarrollar un sistema operativo tipo Unix que fuera libre; lo que hoy en día conocemos como Linux es un sistema operativo GNU, aunque sería más preciso llamarlo GNU/Linux. Pues bien, dentro de este proyecto se desarrolló GCC, cuyo compilador C++ se ajusta al estándar ISO/IEC.

La mayor parte del software libre está protegido por la licencia pública GNU denominada GPL (*GNU Public License*).

REALIZACIÓN DE UN PROGRAMA EN C++

En este apartado se van a exponer los pasos a seguir en la realización de un programa, por medio de un ejemplo.

La siguiente figura muestra de forma esquemática lo que un usuario de C++ necesita y debe hacer para desarrollar un programa.

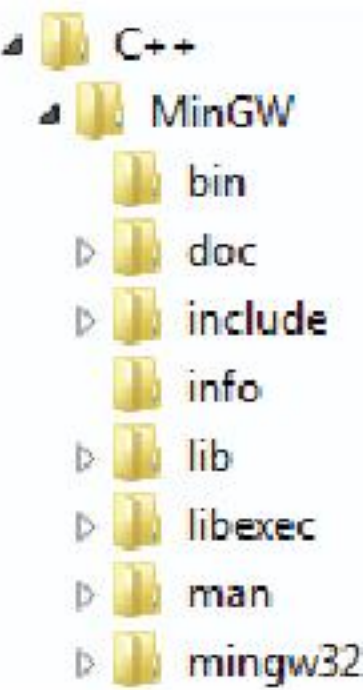


Evidentemente, para poder escribir programas se necesita un entorno de desarrollo C++. En Internet puede encontrar varios con licencia pública GNU que se distribuyen gratuitamente. Por ejemplo, el compilador C++ de GCC se puede obtener en la dirección de Internet:

<http://gcc.gnu.org>

Así mismo, el CD-ROM que acompaña al libro incluye *MinGW*, una versión nativa de Win32 de GCC (para Windows 2000/XP/Vista). Linux también incluye una implementación GCC.

Para instalar la implementación *MinGW* de GCC incluida en el CD en una plataforma Windows, descargue el fichero *MinGW-x.x.x.exe*, o bien utilice la versión suministrada en el CD del libro y ejecútelo. Después, siga los pasos especificados por el programa de instalación. Puede ver más detalles sobre la instalación en los apéndices del libro. Una vez finalizada la instalación, suponiendo que la realizó en la carpeta *C++*, se puede observar el siguiente contenido:



- La carpeta *bin* contiene las herramientas de desarrollo. Esto es, los programas para compilar (*gcc* permite compilar un programa C, *g++* permite compilar un programa C++, etc.), depurar (*gdb*), y otras utilidades.
- La carpeta *include* contiene los ficheros de cabecera de C.
- La carpeta *doc* contiene información de ayuda acerca de la implementación *MinGW*.
- La carpeta *lib* contiene bibliotecas de clases, de funciones y ficheros de soporte requeridos por las herramientas de desarrollo.
- La carpeta *mingw32* contiene otras carpetas *bin* y *lib* con otros ficheros adicionales.

Sólo falta un editor de código fuente C++. Es suficiente con un editor de texto sin formato. No obstante, todo el trabajo de edición, compilación, ejecución y depuración se hará mucho más fácil si se utiliza un entorno de desarrollo con interfaz gráfica de usuario que integre las herramientas mencionadas, en lugar de tener que utilizar la interfaz de línea de órdenes del entorno de desarrollo C++ instalado, como veremos a continuación.

Entornos de desarrollo integrados para C++ hay varios: *Microsoft Visual Studio*, *CodeBlocks*, *Eclipse*, *NetBeans*, etc. Concretamente en el CD se proporciona *CodeBlocks*: un entorno integrado, con licencia pública GNU, y que se ajusta a las necesidades de las aplicaciones que serán expuestas en este libro. Para más detalle véase en los apéndices *Instalación del paquete de desarrollo*.

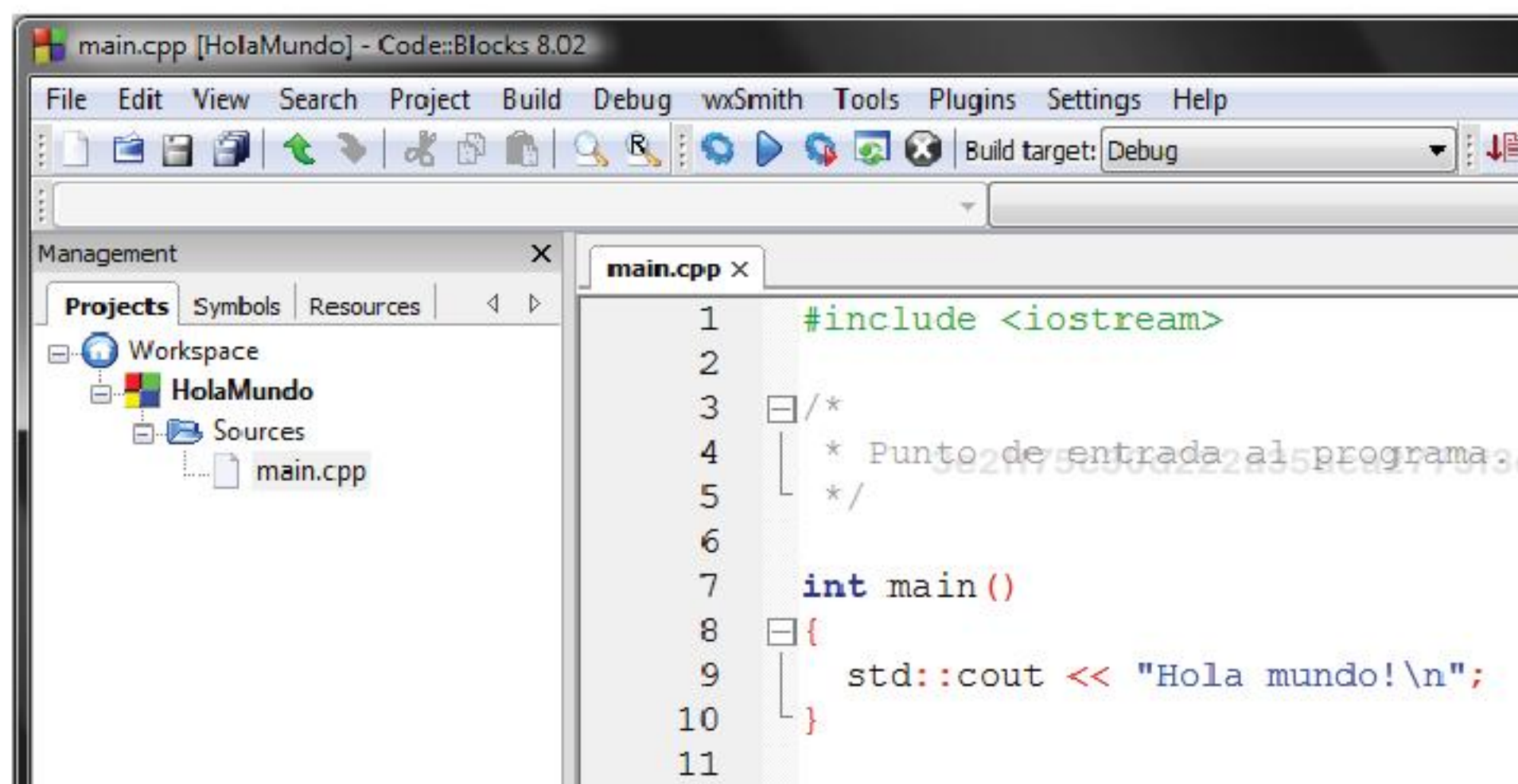
Cómo crear un programa

Empecemos con la creación de un programa sencillo: el clásico ejemplo de mostrar un mensaje de saludo.

Este sencillo programa lo realizaremos utilizando el entorno de desarrollo integrado *CodeBlocks*. No obstante, una vez editado el fichero fuente C++, podría también compilarlo y ejecutarlo desde la línea de órdenes, aspecto que puede ver con detalle en el apéndice *Entornos de desarrollo* del libro.

Empecemos por editar el fichero fuente C++ correspondiente al programa. Primeramente pondremos en marcha el EDI, en nuestro caso *CodeBlocks*. Después, creamos un nuevo proyecto, por ejemplo *HolaMundo*, con un fichero *main.cpp* (el nombre del fichero puede ser cualquiera, pero la extensión debe ser

.cpp) y lo editamos como muestra la figura siguiente (para más detalles, véase el apéndice *Entornos de desarrollo* del libro):



¿Qué hace este programa?

Comentamos brevemente cada línea de este programa. No apurarse si algunos de los términos no quedan muy claros ya que todos ellos se verán con detalle en capítulos posteriores.

La primera línea incluye el fichero de cabecera *iostream* que contiene las declaraciones necesarias para que se puedan ejecutar las sentencias de entrada o salida (E/S) que aparecen en el programa; en nuestro caso para **cout**. Esto significa que, como regla general, antes de invocar a algún elemento de la biblioteca de C++ este tiene que estar declarado. Las palabras reservadas de C++ que empiezan con el símbolo # reciben el nombre de *directrices* del compilador y son procesadas por el *preprocesador* de C++ cuando se invoca al compilador, pero antes de iniciarse la compilación.

Las siguientes líneas encerradas entre */** y **/* son simplemente un comentario. Los comentarios no son tenidos en cuenta por el compilador, pero ayudan a entender un programa cuando se lee.

A continuación se escribe la función principal **main**. Todo programa escrito en C++ tiene una función **main**. Observe que una función se distingue por el modificador () que aparece después de su nombre y que el cuerpo de la misma empieza con el carácter { y finaliza con el carácter }. Las llaves, {}, delimitan el bloque de código que define las acciones que tiene que ejecutar dicha función.

Cuando se ejecuta un programa, C++ espera que haya una función **main**. Esta función define el punto de entrada y de salida normal del programa.

El objeto **std::cout** de la biblioteca C++ escribe en la salida estándar (la pantalla) las expresiones que aparecen a continuación de los operadores << (operador de inserción). En nuestro caso, escribe la cadena de caracteres especificada entre comillas, *Hola mundo!*, y un retorno de carro (CR) más un avance a la línea siguiente (LF) que es lo que indica la constante `\n`. Observe que la sentencia completa finaliza con punto y coma.

Guardar el programa escrito en el disco

El programa editado está ahora en la memoria. Para que este trabajo pueda tener continuidad, el programa escrito se debe grabar en el disco utilizando la orden correspondiente del editor. Muy importante: el nombre del programa fuente debe añadir la extensión *cpp* (*c plus plus: c más más*), o bien *cxx*.

Compilar y ejecutar el programa

El siguiente paso es *compilar* el programa; esto es, traducir el programa fuente a lenguaje máquina para posteriormente enlazarlo con los elementos necesarios de la biblioteca de C++, proceso que generalmente se realiza automáticamente, y obtener así un programa ejecutable. Por ejemplo, en el entorno de desarrollo de C++ que hemos instalado, ejecutaremos la orden *Build* del menú *Build*.

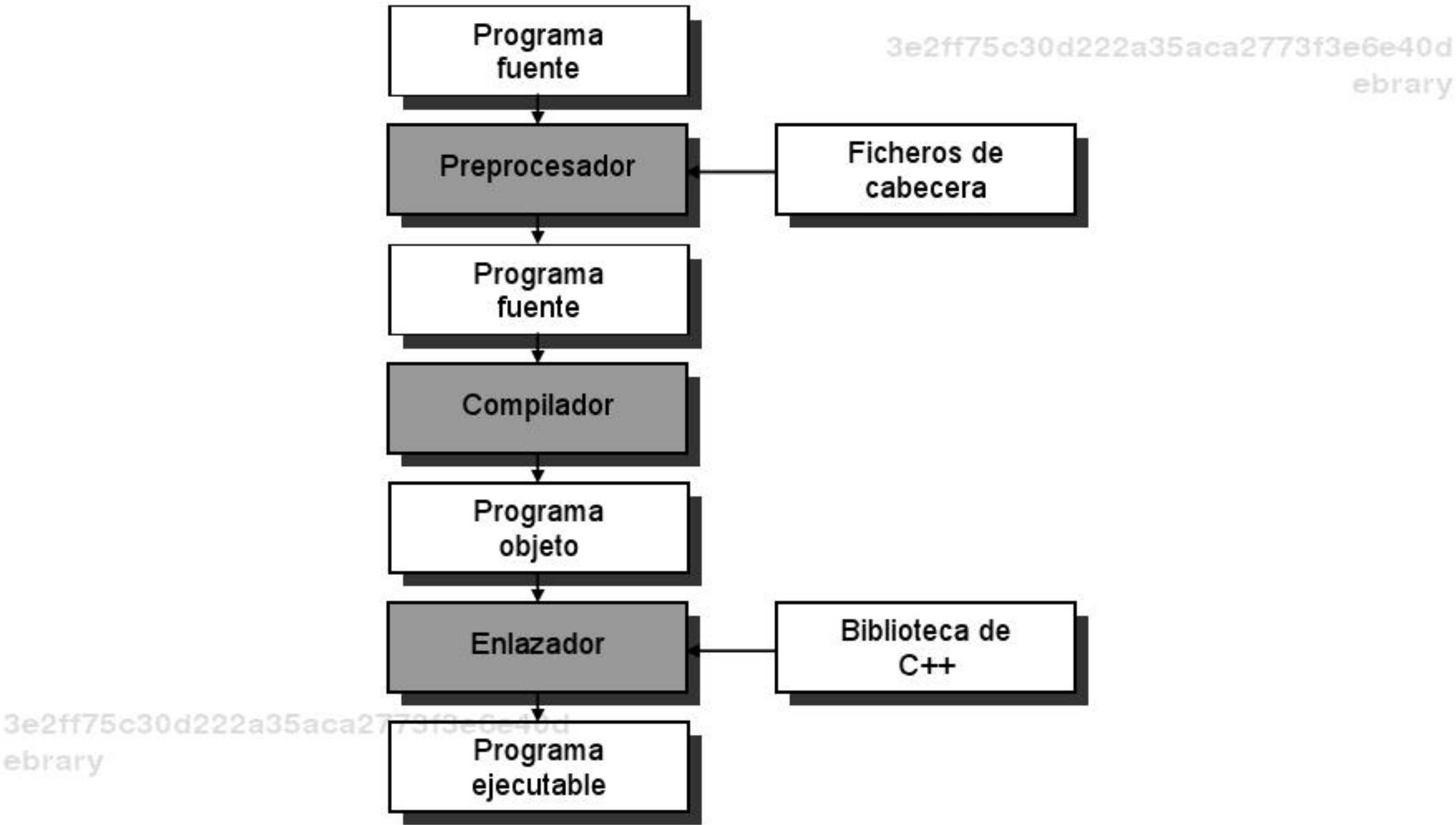
Al compilar un programa se pueden presentar *errores de compilación*, debidos a que el programa escrito no se adapta a la sintaxis y reglas del compilador. Estos errores se irán corrigiendo hasta obtener una compilación sin errores.

Para ejecutar el fichero resultante, en el entorno de desarrollo de C++ que estamos utilizando, ejecutaremos la orden *Run* del menú *Build*.

Biblioteca estándar de C++

C++ carece de instrucciones de E/S, de instrucciones para manejo de cadenas de caracteres, etc., con lo que este trabajo queda para la biblioteca de funciones genéricas y clases provista con el compilador. Una función es un conjunto de instrucciones que realizan una tarea específica. Una clase es un tipo de objetos. Una biblioteca es un fichero separado en el disco (generalmente con extensión *.lib*, típica de Windows, o con extensión *.a*, típica de LINUX) que contiene las clases y funciones genéricas que realizan las tareas más comunes, para que nosotros no tengamos que escribirlas. Como ejemplo, hemos visto anteriormente el objeto **std::cout**. Si este objeto no existiera, sería labor nuestra el escribir el código nece-

sario para visualizar los resultados sobre la pantalla. Toda la biblioteca estándar de C++ está definida en un único espacio de nombres llamado `std` (estándar). Por eso se escribió `std::cout` en vez de `cout`; como explicaremos en un capítulo posterior, los espacios de nombres son un mecanismo para evitar colisiones entre los nombres dados a los elementos que intervienen en un programa. Estos elementos de la biblioteca de C++ utilizados en nuestro programa serán incluidos en el fichero ejecutable por el *enlazador* (*linker*) sólo si no se produjeron errores de compilación. En la figura siguiente se muestran los pasos seguidos, en este caso por el EDI, para obtener un programa ejecutable:



Otra forma de utilizar elementos de la biblioteca estándar de C++ es indicando explícitamente mediante la directriz **using** el espacio de nombres al que pertenecen, en lugar de poner como prefijo para cada uno de ellos dicho espacio. El ejemplo siguiente muestra cómo sería el programa anterior utilizando esta técnica:

```
#include <iostream>
using namespace std;
/*
 * Punto de entrada al programa.
 */
int main()
{
    cout << "¡¡¡Hola mundo!!!\n";
}
```

Cuando se crea un fichero ejecutable, primero se utiliza el compilador C++ para compilar el programa fuente, el cual puede estar formado por uno o más ficheros *.cpp*, dando lugar a uno o más ficheros intermedios conocidos como ficheros objeto (con extensión *.obj*, típica de Windows, o *.o*, típica de LINUX) ya que cada *cpp* se compila por separado. A continuación se utiliza el programa *enlazador* (*linker*) para unir en un único fichero ejecutable el fichero o ficheros objeto y los elementos de la biblioteca estándar de C++ que el programa utilice.

Cuando se hace referencia a un elemento externo (no definido en el programa, sino en una biblioteca externa), como sucede con el objeto `cout`, el enlazador hará una de estas dos cosas: buscará la definición del elemento, primero en los módulos objeto del programa (ficheros *.obj* o *.o*) y después en la biblioteca, y si lo encuentra, la referencia estará resuelta, y si no, añadirá el identificador del mismo a la lista de “referencias no resueltas” para comunicárselo al usuario.

Según lo expuesto, cada vez que se realiza el proceso de *compilación* y *enlace* del programa actual, C++ genera automáticamente sobre el disco un fichero ejecutable. Este fichero puede ser ejecutado directamente desde el sistema operativo sin el soporte de C++, escribiendo el nombre del fichero a continuación del símbolo del sistema (*prompt* del sistema) y pulsando la tecla *Entrar*.

Al ejecutar el programa, pueden ocurrir *errores durante la ejecución*. Por ejemplo, puede darse una división por 0. Estos errores solamente pueden ser detectados cuando se ejecuta el programa y serán notificados con el correspondiente mensaje de error.

Hay *otro tipo de errores* que no dan lugar a mensaje alguno. Por ejemplo, un programa que no termine nunca de ejecutarse, debido a que presenta un lazo donde no se llega a dar la condición de terminación. Para detener la ejecución se tienen que pulsar las teclas *Ctrl+C* en Windows o *Ctrl+D* en Linux.

Depurar un programa

Una vez ejecutado el programa, la solución puede ser incorrecta. Este caso exige un análisis minucioso de cómo se comporta el programa a lo largo de su ejecución; esto es, hay que entrar en la fase de *depuración* del programa.

La forma más sencilla y eficaz para realizar este proceso es utilizar un programa *depurador*. En el apéndice *Entornos de desarrollo* se explica cómo utilizar un depurador desde un EDI.

EJERCICIO

Para practicar con un programa más, escriba el siguiente ejemplo y pruebe los resultados. Este ejemplo visualiza como resultado la suma, la resta, la multiplicación y la división de dos cantidades enteras.

Abra el entorno de desarrollo integrado (EDI), cree un proyecto, por ejemplo *Aritmetica*, y edite el programa ejemplo que se muestra a continuación. Recuerde, el nombre del fichero fuente debe tener extensión *.cpp*, por ejemplo *main.cpp*.

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

/*
 * Operaciones aritméticas
 */

int main()
{
    int dato1, dato2, resultado;

    dato1 = 20;
    dato2 = 10;

    // Suma
    resultado = dato1 + dato2;
    cout << dato1 << " + " << dato2 << " = " << resultado << endl;

    // Resta
    resultado = dato1 - dato2;
    cout << dato1 << " - " << dato2 << " = " << resultado << endl;

    // Producto
    resultado = dato1 * dato2;
    cout << dato1 << " * " << dato2 << " = " << resultado << endl;

    // Cociente
    resultado = dato1 / dato2;
    cout << dato1 << " / " << dato2 << " = " << resultado << endl;
}
```

La constante **endl** (final de línea) hace la misma acción que **\n** y además, vacía el *buffer* de la salida estándar. Una vez editado el programa, guárdelo en el disco con el nombre *Aritmetica.cpp*.

¿Qué hace este programa?

Fijándonos en la función principal, **main**, vemos que se han declarado tres variables enteras (de tipo **int**): *dato1*, *dato2* y *resultado*.

```
int dato1, dato2, resultado;
```

Las líneas que comienzan con `//` son comentarios estilo C++.

El siguiente paso asigna el valor 20 a la variable *dato1* y el valor 10 a la variable *dato2*.

```
dato1 = 20;
dato2 = 10;
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

A continuación se realiza la suma de esos valores y se escriben los datos y el resultado.

```
resultado = dato1 + dato2;
cout << dato1 << " + " << dato2 << " = " << resultado << endl;
```

El objeto **cout** escribe un resultado de la forma:

```
20 + 10 = 30
```

Observe que la expresión resultante está formada por seis elementos: *dato1*, `" + "`, *dato2*, `" = "`, *resultado* y **endl**. Unos elementos son numéricos y otros son constantes de caracteres. Para mostrar cada uno de los seis elementos se ha empleado el operador de inserción `<<`.

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

Un proceso similar se sigue para calcular la diferencia, el producto y el cociente.

Para finalizar, compile, ejecute el programa y observe los resultados.

DECLARACIÓN DE UNA VARIABLE

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo, valor que puede ser modificado a lo largo de la ejecución del bloque donde la variable es accesible, tantas veces como se necesite. La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo: el tipo del valor que va a almacenar. Por ejemplo, el siguiente código declara cuatro variables: *a* de tipo **double**, *b* de tipo **float**, y *c* y *r* de tipo **int**:

3e2ff75c30d222a35aca2773f3e6e40d
ebrary


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

int main()
{
    double a;
    float b;
    int c, r;

    // ...
}
```

Por definición, una variable declarada dentro de un bloque, entendiendo por bloque el código encerrado entre los caracteres '{' y '}', es accesible sólo dentro de ese bloque. Más adelante, cuando tratemos con objetos matizaremos el concepto de accesibilidad.

Según lo expuesto, las variables *a*, *b*, *c* y *r* son accesibles sólo desde la función **main**. En este caso se dice que dichas variables son *locales* al bloque donde han sido declaradas. Una variable local se crea cuando se ejecuta el bloque donde se declara y se destruye cuando finaliza la ejecución de dicho bloque.

Las variables locales no son iniciadas por el compilador C++. Por lo tanto, es aconsejable iniciarlas para evitar resultados inesperados.

```
int main()
{
    double a = 0;
    float b = 0;
    int c = 0, r = 0;

    // ...

    cout << a << ", " << b << ", " << c << ", " << r << endl;
}
```

Cuando elija el identificador para declarar una variable, tenga presente que el compilador C++ trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo las variables *dato1* y *Dato1* son diferentes.

Respecto al tipo de una variable, depende del tipo de valor que vaya a almacenar. Distinguimos varios tipos de valores que podemos clasificar en: tipos enteros, **short**, **int**, **long** y **char**, tipos reales, **float** y **double** y el tipo **bool**.

Cada tipo tiene un rango diferente de valores positivos y negativos, excepto el **bool** que sólo tiene dos valores: **true** y **false**. Por lo tanto, el tipo que se seleccio-

ne para declarar cada variable en un determinado programa dependerá del rango y tipo de los valores que vayan a almacenar: enteros, fraccionarios o booleanos.

El tipo **short** permite declarar datos enteros comprendidos entre -32768 y $+32767$ (16 bits de longitud), el tipo **int** declara datos enteros comprendidos entre -2147483648 y $+2147483647$ (32 bits de longitud) y el tipo **long** dependiendo de la implementación puede ser de 32 bits de longitud, igual que un **int** (es nuestro caso), o de 64 bits (-9223372036854775808 a $+9223372036854775807$). A continuación se muestran algunos ejemplos:

```
short i = 0, j = 758; // short es sinónimo de short int
int k = -125000000;
long l = 125000000; // long es sinónimo de long int
```

3e2ff75c30d222a35aca2773f3e6e40d

ebrary

A los tipos anteriores, C++ añade los correspondientes tipos enteros sin signo: **unsigned short** (0 a $2^{16}-1$), **unsigned int** (0 a $2^{32}-1$) y **unsigned long** (0 a $2^{32}-1$).

El tipo **char** es utilizado para declarar datos enteros en el rango -128 a 127 . Los valores 0 a 127 se corresponden con los caracteres ASCII del mismo código (ver los apéndices). El juego de caracteres ASCII conforma una parte muy pequeña del juego de caracteres Unicode (valores de 0 a 65535), donde cada carácter ocupa 16 bits. Hay lenguajes que utilizan este código con el único propósito de internacionalizar el lenguaje. C++ cubre este objetivo con el tipo **wchar_t**.

El siguiente ejemplo declara la variable *car* de tipo **char** a la que se le asigna el carácter 'a' como valor inicial (observe que hay una diferencia entre 'a' y *a*; *a* entre comillas simples es interpretada por el compilador C++ como un valor, un carácter, y *a* sin comillas sería interpretada como una variable). Las dos declaraciones siguientes son idénticas:

3e2ff75c30d222a35aca2773f3e6e40d

ebrary

```
char car = 'a';
char car = 97; // la 'a' es el decimal 97
```

El tipo **float** (32 bits de longitud) se utiliza para declarar un dato que puede contener una parte decimal. Los datos de tipo **float** almacenan valores con una precisión aproximada de 6 dígitos. Por ejemplo:

```
float a = 3.14159F;
float b = 2.2e-5F; // 2.2e-5 = 2.2 por 10 elevado a -5
float c = 2.0/3.0F; // 0.666667
```

Para especificar que una constante fraccionaria (no entera) es de tipo **float**, hay que añadir al final de su valor la letra 'f' o 'F', de lo contrario será considerada de tipo **double**.

3e2ff75c30d222a35aca2773f3e6e40d

ebrary

El tipo **double** (64 bits de longitud) se utiliza para declarar un dato que puede contener una parte decimal. Los datos de tipo **double** almacenan valores con una precisión aproximada de 15 dígitos. El siguiente ejemplo declara la variable *a*, de tipo real de precisión doble:

```
double a = 3.14159; // una constante es double por omisión
```

ASIGNAR VALORES

La finalidad de un programa es procesar datos *numéricos* y *cadenas de caracteres* para obtener un resultado. Estos datos, generalmente, estarán almacenados en variables y el resultado obtenido también será almacenado en variables. ¿Cómo son almacenados? Pues a través de las funciones proporcionadas por la biblioteca de C++, o bien utilizando una sentencia de asignación de la forma:

variable operador_de_asignación valor

Una sentencia de asignación es asimétrica. Esto quiere decir que se evalúa la expresión que está a la derecha del operador de asignación y el resultado se asigna a la variable especificada a su izquierda. Por ejemplo:

```
resultado = dato1 + dato2; // = es el operador de asignación
```

Pero, según lo expuesto, no sería válido escribir:

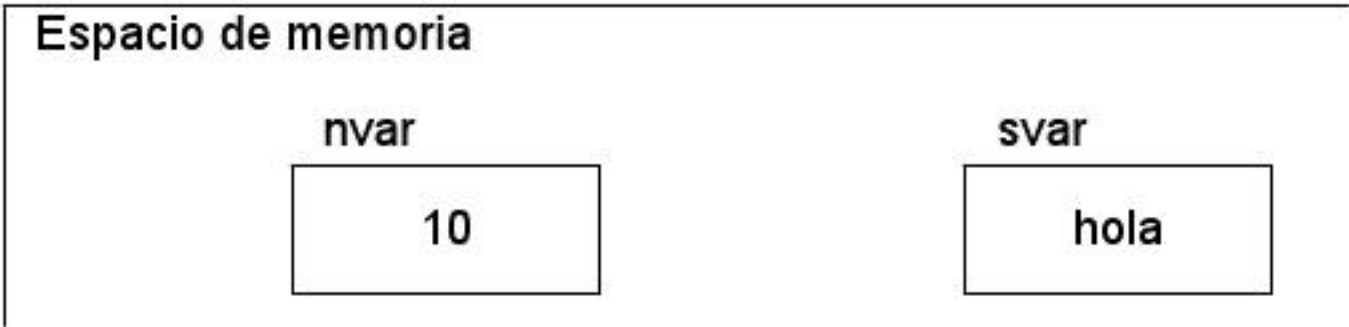
```
dato1 + dato2 = resultado;
```

Mientras que los datos numéricos son almacenados en variables de alguno de los tipos de valores expuestos anteriormente, las cadenas de caracteres son almacenadas en objetos de la clase **string** o en matrices, cuyo estudio se pospone para un capítulo posterior; no obstante veamos un ejemplo. Un objeto de la clase **string** (su manipulación implica incluir el fichero de cabecera `<string>`) se define y se le asigna un valor así:

```
string cadena; // cadena es un objeto string  
cadena = "hola"; // asignar a cadena la constante "hola"  
// Ambas sentencias equivalen a: string cadena = "hola";
```

Cuando se asigna un valor a una variable estamos colocando ese valor en una localización de memoria asociada con esa variable. Por ejemplo:

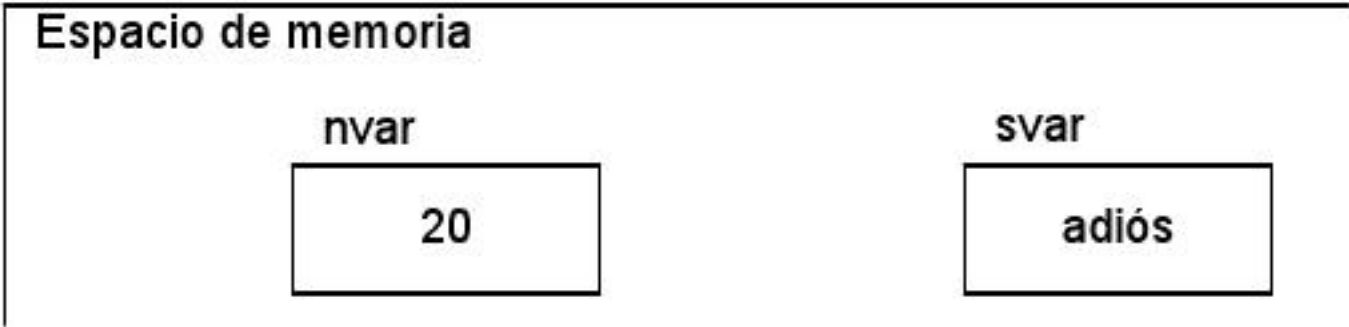
```
int nvar = 10; // variable de un tipo entero (int)  
string svar = "hola"; // objeto de tipo string
```



Lógicamente, cuando la variable tiene asignado un valor y se le asigna uno nuevo, el valor anterior es destruido ya que el valor nuevo pasa a ocupar la misma localización de memoria. En el ejemplo siguiente, se puede observar, con respecto a la situación anterior, que el contenido de *nvar* se modifica con un nuevo valor 20, y que el objeto *svar* también se modifica; ahora el objeto *svar* de tipo **string** contiene “adiós”.

3e2ff75c30d222a35aca2773f3e6e40debrary

```
nvar = 20;
svar = "adiós";
```



El siguiente ejemplo declara tres variables numéricas *a*, *b* y *c*, y un objeto **string** *s*; después asigna valores a esas variables y al objeto.

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

int main()
{
    double a = 0, b = 0;
    int c = 0;
    string s;

    a = 3.14; b = 2.71; c = 2;
    s = "Datos";
}
```

AÑADIR COMENTARIOS

Un comentario es un mensaje a cualquiera que lea el código fuente. Añadiendo comentarios se hace más fácil la comprensión de un programa. La finalidad de los comentarios es explicar el código fuente. Se pueden utilizar comentarios acotados o de una sola línea.

Un comentario acotado empieza con los caracteres `/*` y finaliza con los caracteres `*/`. Estos comentarios pueden ocupar más de una línea, pero no pueden anidarse. Por ejemplo:

```
/*  
 * Asignar datos:  
 *   a, b, c representan datos numéricos.  
 *   s representa una cadena de caracteres.  
 */
```

```
int main()  
{  
    double a = 0, b = 0;  
    int c = 0;  
    string s;  
  
    a = 3.14; b = 2.71; c = 2;  
    s = "Datos";  
}
```

Un comentario de una sola línea comienza con una doble barra (`//`) y se extiende hasta el final de la línea. Por ejemplo, el siguiente programa declara tres variables numéricas *a*, *b* y *c*, y un objeto *s* de tipo **string** (cadena de caracteres); después asigna valores a las variables y al objeto, y finalmente los muestra.

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

```
// Asignar datos:  
//   a, b, c representan datos numéricos.  
//   s representa una cadena de caracteres.  
int main()  
{  
    double a = 0, b = 0;  
    int c = 0;  
    string s;  
  
    a = 3.14; b = 2.71; c = 2;  
    s = "Datos";  
  
    cout << s + ":\n";  
    cout << " a = " << a << '\n';  
    cout << " b = " << b << '\n';  
    cout << " c = " << c << '\n';  
}
```

Ejecución del programa:

Datos:

$a = 3,14$

$b = 2,71$

$c = 2$

La constante de carácter (carácter encerrado entre comillas simples) `\n` especifica un salto al principio de la línea siguiente.

EXPRESIONES ARITMÉTICAS

Una expresión es un conjunto de operandos unidos mediante operadores para especificar una operación determinada. Todas las expresiones cuando se evalúan retornan un valor. Por ejemplo, la siguiente expresión retorna la suma de *dato1* y *dato2*:

`dato1 + dato2`

C++ define cinco operadores aritméticos que son los siguientes:

- +** *Suma*. Los operandos pueden ser enteros o reales.
- *Resta*. Los operandos pueden ser enteros o reales.
- *** *Multiplicación*. Los operandos pueden ser enteros o reales.
- /** *División*. Los operandos pueden ser enteros o reales. Si ambos operandos son enteros, el resultado es entero. En el resto de los casos el resultado es real.
- %** *Módulo* o resto de una división entera. Los operandos tienen que ser enteros.

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

Cuando en una operación aritmética los operandos son de diferentes tipos, ambos son convertidos al tipo del operando de precisión más alta. En una asignación, el resultado obtenido en una operación aritmética es convertido implícita o explícitamente al tipo de la variable que almacena dicho resultado (véase *Conversión entre tipos primitivos* en el capítulo *Elementos del lenguaje C++*).

Así mismo, cuando en una expresión intervienen varios operadores aritméticos, estos se ejecutan de izquierda a derecha y de mayor a menor prioridad. Los operadores `*`, `/` y `%` tienen entre ellos la misma prioridad pero mayor que la de los operadores `+` y `-` que también tienen la misma prioridad entre ellos. Una expresión entre paréntesis siempre se evalúa primero; si hay varios niveles de paréntesis, son evaluados de más internos a más externos. Por ejemplo:

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary


```
/*
 * Operaciones aritméticas
 */
int main()
{
    double a = 10;
    float b = 20;
    int c = 2, r = 0;
    r = static_cast<int>(7.5 * sqrt(a) - b / c);
    cout << r << '\n';
}
```

Ejecución del programa:

3e2ff75c30d222a35aca2773f3e6e40d
ebruary

13

En este ejemplo, primero se realiza la operación $\text{sqrt}(a)$ (invoca a la función `sqrt` de la biblioteca de C++ para calcular la raíz cuadrada de a ; esto requiere incluir el fichero de cabecera `<cmath>`) y después, el resultado de tipo `double` que se obtiene se multiplica por 7,5. A continuación se realiza b / c convirtiendo previamente c al tipo de b ; el resultado que se obtiene es de tipo `float`. Finalmente se hace la resta de los dos resultados anteriores convirtiendo previamente el resultado de tipo `float` a tipo `double`; se obtiene un resultado de tipo `double` que, como puede observar, es convertido explícitamente a tipo `int` (`static_cast<int>`), truncando la parte decimal, para poder almacenarlo en r . Si no se realiza una conversión explícita, el compilador realizará una conversión implícita `double` a `int` y como esta operación acarrea una pérdida de precisión, avisará de ello al programador por si tal hecho hubiera pasado desapercibido.

3e2ff75c30d222a35aca2773f3e6e40d
ebruary

EXPRESIONES CONDICIONALES

En ocasiones interesará dirigir el flujo de ejecución de un programa por un camino u otro en función del valor de una expresión. Para ello, C++ proporciona la sentencia `if`. Para ver cómo se utiliza esta sentencia, vamos a realizar un ejemplo que verifique si un número es par. En caso afirmativo imprimirá un mensaje “Número par” y a continuación el valor del número. En caso negativo sólo imprimirá el valor del número:

```
#include <iostream>
using namespace std;
/*
 * Expresiones condicionales
 */
int main()
{
    int num = 24;
```

3e2ff75c30d222a35aca2773f3e6e40d
ebruary

```

if ( num % 2 == 0 ) // si el resto de la división es igual a 0,
    cout << "Número par\n";

cout << "Valor: " << num << '\n';
}

```

La sentencia **if** del ejemplo anterior se interpreta así: si la condición especificada entre paréntesis, *num % 2 == 0*, es cierta, invocar a **cout** y escribir “Número par”; si es falsa, no hacer lo anterior. En cualquiera de los dos casos, continuar con la siguiente sentencia (*cout << "Valor: " << num << '\n'*). Según esto, el resultado después de ejecutar este programa será:

```

Número par
Valor: 24

```

Si el número hubiera sido 23, el resultado hubiera sido sólo *Valor: 23*. La expresión que hay entre paréntesis a continuación de **if** es una *expresión condicional* y el resultado de su evaluación siempre es un valor booleano **true** (verdadero) o **false** (falso); estas dos constantes están predefinidas en C++. Los operadores de relación o de comparación que podemos utilizar en estas expresiones son los siguientes:

<	¿Primer operando <i>menor que</i> el segundo?
>	¿Primer operando <i>mayor que</i> el segundo?
<=	¿Primer operando <i>menor o igual que</i> el segundo?
>=	¿Primer operando <i>mayor o igual que</i> el segundo?
!=	¿Primer operando <i>distinto que</i> el segundo?
==	¿Primer operando <i>igual que</i> el segundo?

Modifiquemos el programa anterior para que ahora indique si el número es par o impar. Para este caso emplearemos una segunda forma de la sentencia **if** que consiste en añadir a la anterior la cláusula **else** (si no):

```

#include <iostream>
using namespace std;
/*
 * Expresiones condicionales
 */
int main()
{
    int num = 23;

    if ( num % 2 == 0 ) // si el resto de la división es igual a 0,
        cout << "Número par\n";

```



```
else // si el resto de la división no es igual a 0,
    cout << "Número impar\n";

cout << "Valor: " << num << '\n';
}
```

Ejecución del programa:

Número impar
Valor: 23

La sentencia **if** de este otro ejemplo se interpreta así: si la condición especificada entre paréntesis, $num \% 2 == 0$, es cierta, invocar a **cout** y escribir “Número par” y si no, invocar a **cout** y escribir “Número impar”. En cualquiera de los dos casos continuar con la siguiente sentencia del programa.

A continuación se muestra otra versión del programa anterior que produciría exactamente los mismos resultados. No obstante, representa un estilo peor de programación, ya que repite código, que como hemos visto, se puede evitar.

```
#include <iostream>
using namespace std;
/*
 * Expresiones condicionales
 */
int main()
{
    int num = 23;

    if ( num % 2 == 0) // si el resto de la división es igual a 0,
    {
        cout << "Número par" << '\n';
        cout << "Valor: " << num << '\n';
    }
    else // si el resto de la división no es igual a 0,
    {
        cout << "Número impar" << '\n';
        cout << "Valor: " << num << '\n';
    }
}
```

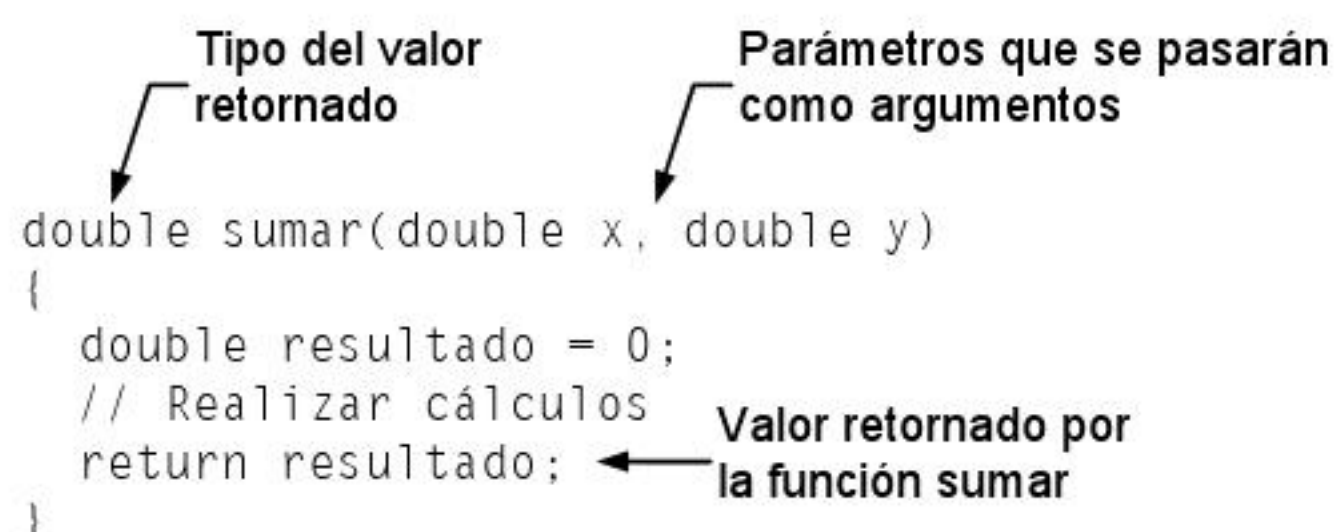
Se puede observar un nuevo detalle, y es que cuando el número de sentencias que se desean ejecutar en función del resultado **true** o **false** de una expresión es superior a una, hay que encerrarlas en un bloque. En el capítulo *Sentencias de control* veremos la sentencia **if** con más detalle.

Observe que los operadores de asignación (**=**) y de igualdad (**==**) son diferentes.

ESCRIBIR NUESTRAS PROPIAS FUNCIONES

De la misma forma que la biblioteca de C++ proporciona funciones predefinidas como `sqrt`, nosotros también podemos añadir a nuestro programa nuestras propias funciones e invocarlas de la misma forma que lo hacemos con las predefinidas.

Por ejemplo, en el programa siguiente la función `main` muestra la suma de dos valores cualesquiera; dicha suma la obtiene invocando a una función `sumar`, añadida por nosotros, que recibe en sus parámetros `x` e `y` los valores a sumar, realiza la suma de ambos y, utilizando la sentencia `return`, devuelve el resultado solicitado por `main`.



```
double sumar(double x, double y)
{
    double resultado = 0;
    // Realizar cálculos
    return resultado;
}
```

Han aparecido algunos conceptos nuevos (argumentos pasados a una función y valor retornado por una función). No se preocupe, sólo se trata de un primer contacto. Más adelante estudiaremos todo esto con mayor profundidad. Para una mejor comprensión de lo dicho, piense en la función llamada *logaritmo* que seguro habrá utilizado más de una vez a lo largo de sus estudios. Esta función devuelve un valor real correspondiente al logaritmo del valor pasado como argumento: $x = \log(y)$. Bueno, pues compárela con la función `sumar` y comprobará que estamos hablando de cosas análogas.

Evidentemente, se pueden escribir funciones que no requieran devolver un valor, lo que se indicará mediante la palabra reservada `void`. Por ejemplo:

```
void mensaje() // esta función no devuelve nada
{
    cout << "un mensaje\n";
}
```

Según lo expuesto y aplicando los conocimientos adquiridos hasta ahora, el programa propuesto puede ser como se muestra a continuación:

```
#include <iostream>

using namespace std;
```



```
/*
 * Función sumar:
 *   parámetros x e y de tipo double
 *   devuelve x + y
 */

double sumar(double x, double y)
{
    double resultado = 0;
    resultado = x + y;
    return resultado;
}

int main()
{
    double a = 10, b = 20, r = 0;
    r = sumar(a, b);
    cout << "Suma = " << r << '\n';
}
```

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

Ejecución del programa:

Suma = 30

Observe cómo es la llamada a la función *sumar*: $r = \text{sumar}(a, b)$. La función es invocada por su nombre, entre paréntesis se especifican los argumentos con los que debe operar, y el resultado que devuelve se almacena en *r*.

Finalmente, si comparamos el esqueleto de la función *sumar* y el de la función **main**, observamos que son muy parecidos: *sumar* devuelve un valor de tipo **double** y **main** un entero (eso es lo que indica **int**) y *sumar* tiene dos parámetros, *x* e *y*, y **main** ninguno, en este caso.

3e2ff75c30d222a35aca2773f3e6e40d
ebrary

EJERCICIOS PROPUESTOS

1. Escriba una aplicación que visualice en el monitor los siguientes mensajes:

```
Bienvenido al mundo de C++.
Podrás dar solución a muchos problemas.
```

2. Decida qué tipos de valores necesita para escribir un programa que calcule la suma y la media de cuatro números de tipo **int**. Escriba un programa como ejemplo.
3. Escriba un programa que incluya una función denominada *calcular* que devuelva como resultado el valor de la expresión:

3e2ff75c30d222a35aca2773f3e6e40d
ebrary