

## CAPÍTULO 2

©F.J.Ceballos/RA-MA

# ELEMENTOS DEL LENGUAJE C++

---

En este capítulo veremos los elementos que aporta C++ (caracteres, secuencias de escape, tipos de datos, operadores, etc.) para escribir un programa. El introducir este capítulo ahora es porque dichos elementos los tenemos que utilizar desde el principio; algunos ya han aparecido en los ejemplos de los capítulos anteriores. Por lo tanto, considere este capítulo como soporte para el resto de los capítulos; esto es, lo que se va a exponer en él lo irá utilizando en menor o mayor medida en los capítulos sucesivos. Por lo tanto, límítense ahora simplemente a realizar un estudio para saber de forma genérica los elementos con los que contamos para desarrollar nuestros programas.

### PRESENTACIÓN DE LA SINTAXIS DE C++

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

Las palabras clave aparecerán en negrita y cuando se utilicen deben escribirse exactamente como aparecen. Por ejemplo:

*char a;*

El texto que aparece en cursiva significa que ahí debe ponerse la información indicada por ese texto. Por ejemplo:

*typedef declaración\_tipo sinónimo[ , sinónimo]...;*

Una información encerrada entre corchetes "[]" es opcional. Los puntos suspensivos "..." indican que pueden aparecer más elementos de la misma forma.

Cuando dos o más opciones aparecen entre llaves "{}" separadas por "|", se elige una, la necesaria dentro de la sentencia. Por ejemplo:

*constante\_entera[{L|U|UL}]*

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

## CARACTERES DE C++

Los caracteres de C++ pueden agruparse en letras, dígitos, espacios en blanco, caracteres especiales, signos de puntuación y secuencias de escape.

### Letras, dígitos y carácter de subrayado

Estos caracteres son utilizados para formar las *constantes*, los *identificadores* y las *palabras clave* de C++. Son los siguientes:

- Letras mayúsculas del alfabeto inglés:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Letras minúsculas del alfabeto inglés:

a b c d e f g h i j k l m n o p q r s t u v w x y z

- Dígitos decimales:

0 1 2 3 4 5 6 7 8 9

- Carácter de subrayado "\_"

El compilador C++ trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo, los identificadores *Pi* y *PI* son diferentes.

### Espacios en blanco

Los caracteres espacio en blanco, tabulador horizontal, tabulador vertical, avance de página y nueva línea son caracteres denominados *espacios en blanco*, porque la labor que desempeñan es la misma que la del espacio en blanco: actuar como separadores entre los elementos de un programa, lo cual permite escribir programas más legibles. Por ejemplo, el siguiente código:

```
int main() { cout << "Hola, qué tal estáis.\n"; }
```

puede escribirse de una forma más legible así:

```
int main()
{
    cout << "Hola, qué tal estáis.\n";
}
```

Los espacios en blanco en exceso son ignorados por el compilador. Según esto, el código siguiente se comporta exactamente igual que el anterior:

```
int main()
{
    cout << "Hola, qué tal estás.\n";
}
```

línea en blanco  
espacios en blanco

La secuencia *Ctrl+Z* en Windows o *Ctrl+D* en Linux es tratada por el compilador como un indicador de fin de fichero (*End Of File*).

## Caracteres especiales y signos de puntuación

Este grupo de caracteres se utiliza de diferentes formas; por ejemplo, para indicar que un identificador es una función o una matriz; para especificar una determinada operación aritmética, lógica o de relación; etc. Son los siguientes:

, . ; : ? ' " ( ) [ ] { } < ! | / \ ~ + # % & ^ \* - = >

## Secuencias de escape

Cualquier carácter de los anteriores puede también ser representado por una *secuencia de escape*. Una secuencia de escape está formada por el carácter \ seguido de una *letra* o de una *combinación de dígitos*. Son utilizadas para acciones como nueva línea, tabular y para hacer referencia a caracteres no imprimibles.

El lenguaje C++ tiene predefinidas las siguientes secuencias de escape:

Secuencia	Nombre
\n	Ir al principio de la siguiente línea.
\t	Tabulador horizontal.
\v	Tabulador vertical (sólo para la impresora).
\b	Retroceso ( <i>backspace</i> ).
\r	Retorno de carro sin avance de línea.
\f	Alimentación de página (sólo para la impresora).
\a	Alerta, pitido.
\'	Comilla simple.
\"	Comilla doble.
\\\	Barra invertida ( <i>backslash</i> ).
\ddd	Carácter ASCII. Representación octal ( <i>d</i> es un dígito del 0 al 7).
\xdd	Carácter ASCII. Representación hexadecimal ( <i>d</i> es un dígito del 0 al 9 o una letra A - Z o a - z).

Observe en la llamada a cout del ejemplo anterior la secuencia de escape \n.

## TIPOS DE DATOS

Recuerde las operaciones aritméticas que realizaba el proyecto *Aritmetica* que vimos en el capítulo 1. Por ejemplo, una de las operaciones que realizábamos era la suma de dos valores:

```
dato1 = 20;  
dato2 = 10;  
resultado = dato1 + dato2;
```

Para que el compilador C++ reconozca esta operación es necesario especificar previamente el tipo de cada uno de los operandos que intervienen en la misma, así como el tipo del resultado. Para ello, escribiremos una línea como la siguiente:

```
int dato1, dato2, resultado;  
dato1 = 20;  
dato2 = 10;  
resultado = dato1 + dato2;
```

La declaración anterior le indica al compilador C++ que *dato1*, *dato2* y *resultado* son de tipo entero (**int**). Observe que se puede declarar más de una variable del mismo tipo utilizando una lista separada por comas.

Los tipos de datos en C++ se clasifican en: tipos *primitivos* y tipos *derivados*. La razón para ofrecer más de un tipo de datos es permitir al programador aprovechar las características del hardware, ya que diferentes máquinas pueden presentar diferencias significativas en los requerimientos de memoria, tiempo de acceso a memoria y velocidad de cálculo entre los distintos tipos.

3e2ff75c30d222a35aca2773f3e6e40d

ebrary

### Tipos primitivos

Se les llama primitivos porque están definidos por el compilador y se clasifican en: tipos enteros, tipos reales, el tipo carácter ampliado **wchar\_t** y el tipo **bool**.

Tipos enteros: **char**, **short**, **int** y **long**.

Tipos reales: **float**, **double** y **long double**.

Cada tipo primitivo tiene un rango diferente de valores positivos y negativos, excepto el tipo **bool** que sólo tiene dos valores: **true** y **false**. El tipo de datos que se seleccione para declarar las variables de un determinado programa dependerá del rango y tipo de valores que vayan a almacenar cada una de ellas y de si éstos son enteros o fraccionarios. Los ficheros de cabecera **<climits>** y **<cfloat>**, así como la plantilla **numeric\_limits<tipo>** definida en **<limits>**, especifican los valores máximo y mínimo para cada tipo, además de otras características.

3f3e6e40d  
ebrary

Cada tipo entero puede ser calificado por las palabras clave **signed** o **unsigned**. Un entero calificado **signed** es un entero con signo; esto es, un valor entero positivo o negativo. Un entero calificado **unsigned** es un valor entero sin signo, el cual es manipulado como un valor entero positivo. Esta calificación da lugar a los siguientes tipos extras:

```
signed char, unsigned char  
signed short, unsigned short  
signed int, unsigned int  
signed long, unsigned long
```

Si los calificadores **signed** y **unsigned** se utilizan sin un tipo entero específico, se asume el tipo **int**. Por este motivo, las siguientes declaraciones son equivalentes:

```
signed x;           // es equivalente a  
signed int x;
```

Un tipo entero calificado con **signed** es equivalente a utilizarlo sin calificar. Según esto, las siguientes declaraciones son equivalentes:

```
char y;           // es equivalente a  
signed char y;
```

Los tipos enteros y reales (excepto el tipo **long double**) ya fueron comentados en el capítulo 1. El tipo **long double** dependiendo de la implementación C++ utilizada puede ser de 8 bytes de longitud, igual que un **double**, o de 12 bytes.

El tipo **wchar\_t** es utilizado para declarar datos enteros en el rango 0 a 65535 lo que permite manipular el juego de caracteres Unicode. Se trata de un código de 16 bits (valores de 0 a 65535), esto es, cada carácter ocupa 2 bytes, con el único propósito de internacionalizar el lenguaje. Los valores 0 a 127 se corresponden con los caracteres ASCII o ANSI del mismo código (ver los apéndices).

El tipo **bool** se utiliza para indicar si el resultado de la evaluación de una expresión booleana es verdadero o falso. Por definición, **true** toma el valor 1 cuando se convierte a entero y **false** el valor 0, y a la inversa, cualquier valor entero distinto de cero se convierte en **true** y cero en **false**.

## Tipos derivados

Se les denomina tipos derivados porque se construyen a partir de los tipos primitivos y se pueden clasificar en: enumeraciones, matrices, funciones, punteros, re-

ferencias, estructuras, uniones y clases. Todos ellos serán explicados en éste y en sucesivos capítulos.

## Enumeraciones

Crear una enumeración supone definir un nuevo tipo de datos y declarar una variable de ese tipo. La sintaxis es la siguiente:

```
enum enumeración
{
    // constantes enteras que forman la enumeración
};
```

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

donde *enumeración* es un identificador que nombra el nuevo tipo definido.

Después de definir una enumeración, podemos declarar una o más variables de ese tipo, de la forma:

```
[enum] enumeración variable[, variable]...;
```

El siguiente ejemplo declara una variable llamada *color* del tipo enumerado *colores*, la cual puede tomar cualquier valor de los especificados en la lista, por ejemplo *amarillo*.

```
enum colores
{
    azul, amarillo, rojo, verde, blanco, negro
};

colores color;

color = amarillo;
```

Cada identificador de la lista de constantes en una enumeración se corresponde con un valor entero, de tal forma que, por defecto, el primer identificador se corresponde con el valor 0, el siguiente con el valor 1, y así sucesivamente. No obstante, para C++ un tipo enumerado es un nuevo tipo entero diferente de los anteriores. Esto significa que en C++ un valor de tipo **int** no puede ser asignado directamente a una variable de un tipo enumerado, sino que hay que hacer una conversión explícita de tipo (véase *Conversión entre tipos de datos* al final de este capítulo). La conversión inversa sí es posible. Por ejemplo:

```
color = 3; // error: conversión implícita int-colores no permitida.
color = static_cast<colores>(3); // correcto: conversión explícita
                                // de int a colores.

int c = verde; // correcto.
```

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

A cualquier identificador de la lista se le puede asignar un valor inicial entero por medio de una expresión constante. Los identificadores sucesivos tomarán valores correlativos a partir de éste. Por ejemplo:

```
enum colores
{
    azul, amarillo, rojo, verde = 0, blanco, negro
} color;
```

Este ejemplo define un tipo enumerado llamado *colores* y declara una variable *color* de ese tipo. Los valores asociados a los identificadores son los siguientes: *azul* = 0, *amarillo* = 1, *rojo* = 2, *verde* = 0, *blanco* = 1 y *negro* = 2.

A las enumeraciones se les aplica las siguientes reglas:

- Dos o más miembros de una enumeración pueden tener un mismo valor.
- Un mismo identificador no puede aparecer en más de una enumeración.
- Desafortunadamente, no es posible leer o escribir directamente un valor de un tipo enumerado. El siguiente ejemplo aclara este detalle.

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

enum colores
{
    azul, amarillo, rojo, verde, blanco, negro
};

int main()
{
    colores color;
    // Leer un color introducido desde el teclado
    cout << "Color: ";
    // cin >> color; sentencia de lectura no permitida. La
    // sustituimos por las tres siguientes:
    int ncolor;
    cin >> ncolor; // leer un color: 0, 1, 2, etc.
    color = static_cast<colores>(ncolor); // conversión explícita
    // Visualizar un color
    cout << color << '\n';
}
```

Ejecución del programa:

```
Color: 3[Entrar]
3
```

En un próximo capítulo verá con detalle el objeto **cin**; ahora límítense a saber que este objeto le permite asignar un valor introducido por el teclado a la variable especificada. En el ejemplo anterior se observa que no es posible asignar a la variable *color* directamente a través del teclado una constante de la enumeración, por ejemplo *verde*, sino que hay que hacerlo indirectamente leyendo la constante entera equivalente (en el ejemplo, 3). Igualmente, **cout** no escribirá *verde*, sino que escribirá 3. Según esto, se preguntará: ¿qué aportan, entonces, las enumeraciones? Las enumeraciones ayudan a acercar más el lenguaje de alto nivel a nuestra forma de expresarnos. Como podrá ver más adelante, la expresión “si el color es verde,...” dice más que la expresión “si el color es 3,...”.

## Clases

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

El lenguaje C++ es un lenguaje orientado a objetos. La base de la programación orientada a objetos es la *clase*. Una clase es un tipo de objetos definido por el usuario. Por ejemplo, la clase **string** de la biblioteca C++ está definida así:

```
class string
{
    // Atributos
    // Métodos
}
```

Y, ¿cómo se define un objeto de esta clase? Pues, una forma de hacerlo sería así:

```
string sTexto = "abc";
```

Suponiendo que la clase **string** tiene un operador de indexación de acceso público, **[i]**, que devuelve el carácter que está en la posición *i*, la siguiente sentencia devolverá el carácter que está en la posición 1 (la ‘b’):

```
char car = sTexto[1];
```

Una característica muy importante que aporta la programación orientada a objetos es la *herencia* ya que permite la reutilización del código escrito por nosotros o por otros. Por ejemplo, el siguiente código define la clase *ofstream* como una clase derivada (que hereda) de *ostream*:

```
class ofstream : ostream
{
    // Atributos
    // Métodos
}
```

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

La clase *ofstream* incluirá los atributos y métodos heredados de *ostream* más los atributos y métodos que se hayan definido en esta clase. Esto significa que un objeto de la clase *ofstream* podrá ser manipulado por los métodos heredados y por los propios.

## SINÓNIMOS DE UN TIPO

Utilizando la palabra reservada **typedef** podemos declarar nuevos nombres de tipos de datos; esto es, sinónimos de otro tipo ya sean primitivos o derivados, los cuales pueden ser utilizados más tarde para declarar variables de esos tipos. La sintaxis de **typedef** es la siguiente:

```
typedef declaración_tipo sinónimo[, sinónimo]...;
```

donde *declaración\_tipo* es cualquier tipo definido en C++, primitivo o derivado, y *sinónimo* es el nuevo nombre elegido para el tipo especificado.

Por ejemplo, la sentencia siguiente declara el nuevo tipo *ulong* como sinónimo del tipo primitivo **unsigned long**:

```
typedef unsigned long ulong;
```

Una vez definido el tipo *ulong* como sinónimo de **unsigned long**, sería posible declarar una variable *dni* de cualquiera de las dos formas siguientes:

```
unsigned long dni; // o bien  
ulong dni;
```

Las declaraciones **typedef** permiten parametrizar un programa para evitar problemas de portabilidad. Si utilizamos **typedef** con los tipos que pueden depender de la instalación, cuando se lleve el programa a otra instalación sólo se tendrán que cambiar estas declaraciones. Por ejemplo:

```
typedef long int32;
```

Si ahora se porta el programa que contiene esta declaración a otra plataforma donde **long** tiene una longitud de 64 bits en lugar de 32, bastaría con cambiar la sentencia anterior por esta otra, suponiendo que la longitud de un **int** son 32 bits:

```
typedef int int32;
```

## LITERALES

Un literal en C++ puede ser: un entero, un real, un valor booleano, un carácter, una cadena de caracteres y una constante como **NULL**. Por ejemplo, son literales: **5**, **3.14**, **true**, **'a'**, **"hola"**. En realidad son valores constantes.

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

## Literales enteros

El lenguaje C++ permite especificar un literal entero en base 10, 8 y 16.

En general, el signo + es opcional si el valor es positivo y el signo – estará presente siempre que el valor sea negativo. El tipo de un literal entero depende de su base, de su valor y de su sufijo. La sintaxis para especificar un literal entero es:

{[+]|-}literal\_entero{L|U|UL}

Si el literal es decimal y no tiene sufijo, su tipo es el primero de los tipos **int**, **long int** o **unsigned long int** en el que su valor pueda ser representado.

Si es octal o hexadecimal y no tiene sufijo, su tipo es el primero de los tipos **int**, **unsigned int**, **long int** o **unsigned long int** en el que su valor pueda ser representado.

También se puede indicar explícitamente el tipo de un literal entero, añadiendo los sufijos *L*, *U* o *UL* (mayúsculas o minúsculas).

Si el sufijo es *L*, su tipo es **long** cuando el valor puede ser representado en este tipo; si no, es **unsigned long**. Si el sufijo es *U*, su tipo es **unsigned int** cuando el valor puede ser representado en este tipo; si no, es **unsigned long**. Si el sufijo es *UL*, su tipo es **unsigned long**.

Un *literal entero decimal* puede tener uno o más dígitos del 0 a 9, de los cuales el primero de ellos es distinto de 0. Por ejemplo:

4326	constante entera int
1522U	constante entera unsigned int
1000L	constante entera long
325UL	constante entera unsigned long

Un *literal entero octal* puede tener uno o más dígitos del 0 a 7, precedidos por 0 (cero). Por ejemplo:

0326 constante entera int en base 8

Un *literal entero hexadecimal* puede tener uno o más dígitos del 0 a 9 y letras de la A a la F (en mayúsculas o en minúsculas) precedidos por 0x o 0X (cero seguido de x). Por ejemplo:

256	número decimal 256
0400	número decimal 256 expresado en octal
0x100	número decimal 256 expresado en hexadecimal

- 0400      número decimal - 256 expresado en octal
- 0x100     número decimal - 256 expresado en hexadecimal

## Literales reales

Un literal real está formado por una *parte entera*, seguida por un *punto decimal*, y una *parte fraccionaria*. También se permite la notación científica, en cuyo caso se añade al valor una *e* o *E*, seguida por un exponente positivo o negativo.

([+]|-) parte-entera.parte-fraccionaria[(e|E)([+]|-) exponente]

donde *exponente* representa cero o más dígitos del 0 al 9 y *E* o *e* es el símbolo de exponente de la base 10 que puede ser positivo o negativo ( $2E-5 = 2 \times 10^{-5}$ ). Si la constante real es positiva, no es necesario especificar el signo y si es negativa lleva el signo menos (-). Por ejemplo:

- 17.24  
17.244283  
.008e3  
27E-3

Un literal real tiene siempre tipo **double**, a no ser que se añada al mismo una *f* o *F*, en cuyo caso será de tipo **float**. Por ejemplo:

17.24F      constante real de tipo float

## Literales de un solo carácter

Los literales de un solo carácter son de tipo **char**. Este tipo de literales está formado por un único carácter encerrado entre *comillas simples*. Una secuencia de escape es considerada como un único carácter. Algunos ejemplos son:

' '                espacio en blanco  
'x'               letra minúscula x  
'\n'               retorno de carro más avance de línea  
'\x07'            pitido  
'\x1B'            carácter ASCII Esc

El valor de una constante de un solo carácter es el valor que le corresponde en el juego de caracteres de la máquina.

Los literales de un solo carácter ampliado (**wchar\_t**) son de la forma *L'x'*. Por ejemplo:

```
wchar_t wcar = L'a'; // el tamaño de wcar es 2 bytes
```

## Literales de cadenas de caracteres

Un literal de cadena de caracteres es una secuencia de caracteres encerrados entre *comillas dobles* (incluidas las secuencias de escape como \"). Por ejemplo:

```
"Esto es una constante de caracteres"  
"3.1415926"  
"Paseo de Pereda 10, Santander"  
"" // cadena vacía  
"Lenguaje \"C++\""  
// produce: Lenguaje "C++"
```

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

## IDENTIFICADORES

Los identificadores son nombres dados a tipos, literales, variables, funciones, etiquetas de un programa, etc. La sintaxis para formar un identificador es:

{*letra|\_*} [{*letra|dígito|\_*}]...

lo cual indica que un identificador consta de uno o más caracteres (véase el apartado anterior *Letras, dígitos y carácter de subrayado*) y que el *primer carácter* debe ser una *letra* o el *carácter de subrayado*. No pueden comenzar por un dígito ni pueden contener caracteres especiales (véase el apartado anterior *Caracteres especiales y signos de puntuación*).

Los identificadores pueden tener cualquier número de caracteres pero dependiendo del compilador que se utilice (en particular, del enlazador), solamente los *n* primeros caracteres son significativos. Esto quiere decir que un identificador es distinto de otro cuando difieren al menos en uno de los *n* primeros caracteres significativos. Algunos ejemplos son:

Suma  
suma  
Calculo\_Numeros\_Primos  
ordenar  
VisualizarDatos

## PALABRAS CLAVE

Las palabras clave son identificadores predefinidos que tienen un significado especial para el compilador C++. Por lo tanto, un identificador definido por el usu-

rio, no puede tener el mismo nombre que una palabra clave. Algunas de las palabras clave que utiliza el lenguaje C++ son las siguientes:

and	do	int	short	typeid
auto	double	long	signed	union
bool	else	namespace	sizeof	unsigned
break	enum	new	static	using
case	extern	not	struct	virtual
catch	false	operator	switch	void
char	float	or	template	wchar_t
class	for	private	this	while
const	friend	protected	throw	xor
continue	goto	public	true	3e2ff75c30d222a35aca2773f3e6e40d
default	if	register	try	ebrary
delete	inline	return	typedef	

Las palabras clave deben escribirse siempre en minúsculas, como están.

## DECLARACIÓN DE CONSTANTES SIMBÓLICAS

Declarar una constante simbólica significa decirle al compilador C++ el nombre de la constante y su valor. La sintaxis para declarar una constante es así:

```
const tipo nombre = valor
```

El siguiente ejemplo declara la constante real *PI* con el valor 3.14159, la constante de un solo carácter *NL* con el valor '\n' y la constante de caracteres *MENSAJE* con el valor "Pulse una tecla para continuar\n":

```
const double PI = 3.14159;
const char NL = '\n';
const string MENSAJE = "Pulse una tecla para continuar\n";
```

El escribir los nombres de las constantes en mayúsculas para diferenciarlas de las variables es sólo una cuestión de estilo.

Una vez declarada e iniciada una constante, ya no se puede modificar su valor; por eso se inicia al declararla. Por ejemplo, suponiendo declarada la constante *PI*, la siguiente sentencia daría lugar a un error:

```
PI = 3.1416; // error
```

## ¿Por qué utilizar constantes?

Utilizando constantes es más fácil modificar un programa. Por ejemplo, supongamos que un programa utiliza N veces una constante de valor 3.14. Si hemos definido dicha constante como *const double PI = 3.14* y posteriormente necesitamos cambiar el valor de la misma a 3.1416, sólo tendremos que modificar una línea, la que define la constante. En cambio, si no hemos declarado *PI*, sino que hemos utilizado el valor 3.14 directamente N veces, tendríamos que realizar N cambios.

## DECLARACIÓN DE UNA VARIABLE

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo. El valor de una variable, a diferencia de una constante, puede cambiar durante la ejecución de un programa. Para utilizar una variable en un programa, primero hay que declararla. La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo:

*tipo identificador[ , identificador]...*

En el ejemplo siguiente se declaran e inician cuatro variables: una de tipo **char**, otra **int**, otra **float** y otra **double**:

```
char c = '\n';
int main()
{
    int i = 0;
    float f = 0.0F;
3e2ff75c30ddoubleada=7030;e6e40d
ebrary
    // ...
}
```

El tipo, primitivo o derivado, determina los valores que puede tomar la variable así como las operaciones que con ella pueden realizarse. Los operadores serán expuestos un poco más adelante.

En el ejemplo anterior puede observar que hay dos lugares donde se puede realizar la declaración de una variable: fuera de todo bloque, entendiendo por bloque un conjunto de sentencias encerradas entre el carácter '{' y el carácter '}', y dentro de un bloque de sentencias.

Cuando la declaración de una variable tiene lugar dentro de un bloque, dicha declaración en C++ puede realizarse en cualquier parte, pero siempre antes de ser

utilizada. Se aconseja declarar las variables justo en los lugares donde vayan a ser utilizadas.

En nuestro ejemplo, se ha declarado la variable *c* antes de la función **main** (fuera de todo bloque) y las variables *i*, *f* y *d* dentro de la función (dentro de un bloque). Una variable declarada fuera de todo bloque se dice que es *global* porque es accesible en cualquier parte del código que hay desde su declaración hasta el final del fichero fuente. Por el contrario, una variable declarada dentro de un bloque se dice que es *local* porque sólo es accesible dentro de éste. Para comprender esto mejor, piense que generalmente en un programa habrá más de un bloque de sentencias. No obstante, esto lo veremos con más detalle en el capítulo siguiente.

Según lo expuesto, la variable *c* es global y las variables *i*, *f* y *d* son locales.

## Iniciación de una variable

Las variables globales son iniciadas por omisión por el compilador C++: las variables numéricas con 0 y los caracteres con ‘\0’. También pueden ser iniciadas explícitamente, como hemos hecho en el ejemplo anterior con *c*. En cambio, las variables locales no son iniciadas por el compilador C++. Por lo tanto, depende de nosotros iniciarlas o no; es aconsejable iniciarlas, ya que, como usted podrá comprobar, esta forma de proceder evitará errores en más de una ocasión.

## OPERADORES

Los operadores son símbolos que indican cómo son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, relacionales, lógicos, unitarios, a nivel de bits, de asignación, operador condicional y otros. En el capítulo *Introducción a C++* vimos los operadores aritméticos y los de relación de los cuales vemos a continuación algunos ejemplos, para después pasar a describir el resto de operadores.

### Operadores aritméticos

El siguiente ejemplo muestra cómo utilizar los operadores aritméticos (+, -, \*, / y %). Como ya hemos venido diciendo, observe que primero se declaran las variables y después se realizan las operaciones deseadas con ellas.

```
int a = 10, b = 3, c;  
float x = 2.0F, y;  
y = x + a;           // El resultado es 12.0 de tipo float  
c = a / b;          // El resultado es 3 de tipo int
```

```
c = a % b;           // El resultado es 1 de tipo int
y = a / b;           // El resultado es 3 de tipo int. Se
                     // convierte a float para asignarlo a y
c = x / y;           // El resultado es 0.666667 de tipo float. Se
                     // convierte a int para asignarlo a c (c = 0)
```

Según hemos indicado en el apartado anterior, para realizar la suma  $x+a$  el valor del entero  $a$  es convertido a **float**, tipo de  $x$ . No se modifica  $a$ , sino que su valor es convertido a **float** sólo para realizar la suma. Por otra parte, del resultado de  $x/y$  sólo la parte entera es asignada a  $c$ , ya que  $c$  es de tipo **int**. Esto indica que los reales son convertidos a enteros, truncando la parte fraccionaria. No obstante, respecto a esta operación de asignación observaremos que el compilador mostrará un aviso indicando que una conversión de **float** a **int** produce una pérdida de precisión, por si esta apreciación hubiera pasado desapercibida para nosotros. Para evitar que el compilador muestre este aviso tendríamos que especificar explícitamente que deseamos realizar esa conversión así:

```
c = static_cast<int>(x / y); // conversión de x/y a int
```

Un resultado real es redondeado. Observe la operación  $x/y$  para  $x$  igual a 2 e  $y$  igual a 3; el resultado es *0.666667* en lugar de *0.666666* porque la primera cifra decimal suprimida es 6. Cuando la primera cifra decimal suprimida es 5 o mayor de 5, la última cifra decimal conservada se incrementa en una unidad.

Quizás ahora le resulte muy sencillo calcular el área de un determinado triángulo que tenga, por ejemplo, una base de 11,5 y una altura de 3. Veámoslo:

```
#include <iostream>
using namespace std;
int main()
{
    double base = 11.5, altura = 3.0, area = 0.0;

    area = base * altura / 2;
    cout << "Area = " << area << endl;
}
```

*Ejecución del programa:*

*Area = 17.25*

## Operadores de relación

Recordar que un operador de relación ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$  (**not\_eq**) y  $\equiv$ ) equivale a una pregunta relativa a cómo son dos operandos entre sí. Por ejemplo, la expre-

sión `x == y` equivale a la pregunta *¿x es igual a y?* Una respuesta *sí* equivale a un valor **true** y una respuesta *no* equivale a un valor **false**. Por ejemplo:

```
int x = 10, y = 0;
bool r = false;

r = x == y;    // r = false (0) porque x no es igual a y
r = x > y;    // r = true (1) porque x es mayor que y
r = x != y;   // r = true (1) porque x no es igual a y
```

En expresiones largas o confusas, el uso de paréntesis y espacios puede añadir claridad, aunque no sean necesarios. Por ejemplo, las sentencias anteriores serían más fáciles de leer si las escribiéramos así:

```
r = (x == y);    // r = false (0) porque x no es igual a y
r = (x > y);    // r = true (1) porque x es mayor que y
r = (x not_eq y); // r = true (1) porque x no es igual a y
```

Estas sentencias producen los mismos resultados que las anteriores, lo que quiere decir que los paréntesis no son necesarios. ¿Por qué? Porque como veremos un poco más adelante, la prioridad de los operadores `==`, `>` y `!=` es mayor que la del operador `=`, por lo que se evalúan antes que éste. Observe también la utilización de la palabra reservada `not_eq` en lugar del operador `!=`.

Los operadores explícitos como `not_eq`, `and`, `or`, `not`, etc. sólo son soportados por compiladores C++ *estándar*. Estos operadores están definidos en el fichero de cabecera `ciso646`.

## Operadores lógicos

El resultado de una operación lógica (AND, OR y NOT) es un valor booleano verdadero o falso (**true** o **false**). Las expresiones que dan como resultado valores booleanos (véanse los operadores de relación) pueden combinarse para formar expresiones *booleanas* utilizando los operadores lógicos indicados a continuación. Los operandos deben ser expresiones que den un resultado **true** o **false**.

En C++, toda expresión numérica con un valor distinto de 0 se corresponde con un valor booleano **true** y toda expresión numérica de valor 0, con **false**.

Operador	Operación
<code>&amp;&amp;</code> o <code>and</code>	Da como resultado verdadero si al evaluar cada uno de los operandos el resultado es verdadero. Si uno de ellos es falso, el resultado es falso. Si el primer operando es falso, el segundo operando no es evaluado.

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

o or	El resultado es falso si al evaluar cada uno de los operandos el resultado es falso. Si uno de ellos es verdadero, el resultado es verdadero. Si el primer operando es verdadero, el segundo operando no es evaluado (el carácter   es el ASCII 124).
! o not	El resultado de aplicar este operador es falso si al evaluar su operando el resultado es verdadero, y verdadero en caso contrario.

El resultado de una operación lógica es de tipo **bool**. Por ejemplo:

```
int p = 10, q = 0;
bool r = false;

r = (p != 0) && (q > 0); // r = false (0)
```

En este ejemplo, los operandos del operador **&&** son:  $p \neq 0$  y  $q > 0$ . El resultado de la expresión  $p \neq 0$  es verdadero porque  $p$  vale 10 y el de  $q > 0$  es falso porque  $q$  es 0. Por lo tanto, el resultado de *verdadero && falso* es falso.

La expresión booleana anterior es equivalente a la siguiente:

```
r = (p not_eq 0) and (q > 0);
```

Los paréntesis que aparecen en la sentencia anterior no son necesarios pero añaden claridad. No son necesarios porque, como veremos un poco más adelante, la prioridad de los operadores de relación es mayor que la de los operadores lógicos, lo que quiere decir que se ejecutan antes.

## Operadores unitarios

Los operadores unitarios se aplican a un solo operando y son los siguientes: **!**, **-**, **~**, **++** y **--**. El operador **!** ya lo hemos visto y los operadores **++** y **--** los veremos más adelante.

Operador	Operación
~ o compl	Complemento a 1 (cambiar ceros por unos y unos por ceros). El carácter ~ es el ASCII 126. El operando debe ser de un tipo primitivo entero.
-	Cambia de signo al operando (esto es, se calcula el complemento a 2 que es el complemento a 1 más 1). El operando puede ser de un tipo primitivo entero o real.

El siguiente ejemplo muestra cómo utilizar estos operadores:

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

```
int a = 2, b = 0, c = 0;
c = -a;           // resultado c = -2
c = compl b;    // resultado c = -1
```

## Operadores a nivel de bits

Estos operadores permiten realizar con sus operandos las operaciones AND, OR, XOR y desplazamientos, bit por bit. Los operandos tienen que ser enteros.

Operador	Operación
& o <b>bitand</b>	Operación AND a nivel de bits.
o <b>bitor</b>	Operación OR a nivel de bits (carácter ASCII 124).
^ o <b>xor</b>	Operación XOR a nivel de bits.
<<	Desplazamiento a la izquierda llenando con ceros por la derecha.
>>	Desplazamiento a la derecha llenando con el bit de signo por la izquierda.

Los operandos tienen que ser de un tipo primitivo entero.

```
int a = 255, r = 0, m = 32;
r = a & 017; // r=15. Pone a cero todos los bits de a
              // excepto los 4 bits de menor peso.
r = r | m;   // r=47. Pone a 1 todos los bits de r que
              // estén a 1 en m.
r = a & ~07; // r=248. Pone a 0 los 3 bits de menor peso de a.
r = a >> 7;  // r=1. Desplazamiento de 7 bits a la derecha.
r = m << 1;  // r=2. Equivale a r = m * 2
r = m >> 1; // r=1. Equivale a r = m / 2
```

## Operadores de asignación

El resultado de una operación de asignación es el valor almacenado en el operando izquierdo, lógicamente después de que la asignación se ha realizado. El valor que se asigna es convertido implícita o explícitamente al tipo del operando de la izquierda (véase el apartado *Conversión entre tipos de datos*). Incluimos aquí los operadores de incremento y decremento porque implícitamente estos operadores realizan una asignación sobre su operando.

Operador	Operación
++	Incremento.
--	Decremento.
=	Asignación simple.

<b>*=</b>	Multiplicación más asignación.
<b>/=</b>	División más asignación.
<b>%=</b>	Módulo más asignación.
<b>+=</b>	Suma más asignación.
<b>-=</b>	Resta más asignación.
<b>&lt;&lt;=</b>	Desplazamiento a izquierdas más asignación.
<b>&gt;&gt;=</b>	Desplazamiento a derechas más asignación.
<b>&amp;= o <b>and_eq</b></b>	Operación AND sobre bits más asignación.
<b> = o <b>or_eq</b></b>	Operación OR sobre bits más asignación.
<b>^= o <b>xor_eq</b></b>	Operación XOR sobre bits más asignación.

Los operandos tienen que ser de un tipo primitivo. A continuación se muestran algunos ejemplos con estos operadores.

```
int x = 0, n = 10, i = 1;
n++;           // Incrementa el valor de n en 1.
++n;           // Incrementa el valor de n en 1.
x = ++n;       // Incrementa n en 1 y asigna el resultado a x.
x = n++;       // Equivale a realizar las dos operaciones
                // siguientes en este orden: x = n; n++.
i += 2;         // Realiza la operación i = i + 2.
x *= n - 3;    // Realiza la operación x = x * (n-3) y no
                // x = x * n - 3.
n >>= 1;        // Realiza la operación n = n >> 1 la cual desplaza
                // el contenido de n 1 bit a la derecha.
```

El operador de incremento incrementa su operando en una unidad independientemente de que se utilice como sufijo o como prefijo; esto es, **n++** y **++n** producen el mismo resultado. Ídem para el operador de decremento.

Ahora bien, cuando se asigna a una variable una expresión en la que intervienen operadores de incremento o de decremento, el resultado difiere según se utilicen estos operadores como sufijo o como prefijo. Si se utilizan como prefijo, primero se realizan los incrementos o decrementos y después la asignación (ver más adelante la tabla de prioridad de los operadores). Por ejemplo, **y = ++x** es equivalente a **y = (x += 1)**. En cambio, si se utilizan como sufijo, el valor asignado corresponde a la evaluación de la expresión antes de aplicar los incrementos o los decrementos. Por ejemplo, **y = x++** es equivalente a **y = (t=x, x+=1, t)**, suponiendo que **t** es una variable del mismo tipo que **x**. Esta última expresión utiliza el operador coma que será estudiado un poco más adelante en este mismo capítulo.

Según lo expuesto, ¿cuál es el valor de **x** después de evaluar la siguiente expresión?

```
x = (a - b++) * (--c - d) / 2
```

Comprobemos el resultado evaluando esta expresión mediante el siguiente programa. Observamos que en el cálculo de *x* intervienen los valores de *b* sin incrementar y de *c* decrementado, con lo que el resultado será *x* igual a 30.

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

int main()
{
    float x = 0, a = 15, b = 5, c = 11, d = 4;
    x = (a - b++) * (--c - d) / 2;
    cout << "x = " << x << ", b = " << b << ", c = " << c << endl;
}
```

*Ejecución del programa:*

```
x = 30, b = 6, c = 10
```

Una expresión de la complejidad de la anterior equivale a calcular la misma expresión sin operadores `++` y `--`, pero incrementando/decrementando antes las variables afectadas por `++` y `--` como prefijo e incrementando/decrementando después las variables afectadas por `++` y `--` como sufijo. Esto equivale a escribir el programa anterior así:

```
int main()
{
    float x = 0, a = 15, b = 5, c = 11, d = 4;
    --c; // o bien c--
    x = (a - b) * (c - d) / 2;
    b++;
    cout << "x = " << x << ", b = " << b << ", c = " << c << endl;
}
```

La aplicación de la regla anterior se complica cuando una misma variable aparece en la expresión, afectada varias veces por los operadores `++` y `--` (incluso, reutilizada a la izquierda del signo igual). Por ejemplo:

```
x = (a - b++) * (--b - d) * b++ / (b - d);
```

Cuando se aplica la regla anterior a un caso como éste, hay que tener en cuenta que los incrementos/decrementos como prefijo afectan a los cálculos que le siguen en la propia expresión; por eso habrá que intercalarlos en el lugar adecuado.

En cambio, los incrementos/decrementos como sufijo se aplican igual que antes, al final. El ejemplo siguiente realiza los mismos cálculos que la expresión anterior:

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

int main()
{
    float x = 0, a = 20, b = 10, d = 4;

    x = (a - b);
    --b;
    x *= (b - d) * b / (b - d);
    b++;
    b++;
    cout << "x = " << x << ", b = " << b << endl;
}
```

Ejecución del programa:

x = 90, b = 11

Este código es mucho más sencillo de entender que la expresión equivalente anterior, y también menos propenso a introducir errores, por lo que se recomienda esta forma de trabajar.

## Operador condicional

El operador condicional (?:), llamado también operador ternario, se utiliza en expresiones condicionales, que tienen la forma siguiente:

*operando1 ? operando2 : operando3*

La expresión *operando1* debe ser una expresión booleana. La ejecución se realiza de la siguiente forma:

- Si el resultado de la evaluación de *operando1* es verdadero, el resultado de la expresión condicional es *operando2*.
- Si el resultado de la evaluación de *operando1* es falso, el resultado de la expresión condicional es *operando3*.

El siguiente ejemplo asigna a *mayor* el resultado de  $(a > b) ? a : b$ , que será *a* si *a* es mayor que *b* y *b* si *a* no es mayor que *b*.

```
double a = 10.2, b = 20.5, mayor = 0;  
mayor = (a > b) ? a : b;
```

## Otros operadores

Finalmente vamos a exponer los operadores *global* y de *resolución de ámbito*, el operador *tamaño de*, el operador *coma*, y los operadores *dirección de*, *contenido de* y *referencia a*.

### Operador global y de resolución de ámbito (::)

El operador :: permite acceder a una variable global cuya visibilidad ha sido ocultada por una variable local. El siguiente ejemplo define una variable global v y otra local con el mismo nombre, en la función **main**. Observe que para acceder a la variable global se utiliza el operador ::.

```
#include <iostream>  
using namespace std;  
  
float v;  
  
int main()  
{  
    int v = 7;  
    ::v = 10.5; // acceso a la variable global v  
    cout << "variable local v = " << v << '\n';  
    cout << "variable global v = " << ::v << '\n';  
}
```

Después de ejecutar este programa el resultado que se obtiene es el siguiente:

```
variable local v = 7  
variable global v = 10.5
```

También se utiliza para especificar a qué clase pertenece un determinado método; por ejemplo, cuando se define fuera del ámbito de la clase, o cuando se invoca si se declaró **static**, cuestión que estudiaremos en el capítulo dedicado a clases de objetos. Por ejemplo, la siguiente línea de código invoca al método *max* declarado **static** en la clase *numeric\_limits<float>*:

```
cout << "El float más grande es: " << numeric_limits<float>::max();
```

## Operador sizeof

El tamaño de los objetos en C++ es un múltiplo del tamaño de un **char** que por definición es 1 *byte*. Para obtener este tamaño se utiliza el operador **sizeof**. Este operador da como resultado el tamaño en *bytes* de su operando que puede ser el *identificador* o el *tipo* de una variable previamente declarada. Por ejemplo:

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

int main()
{
    int a = 0, t = 0;

    t = sizeof a;
    cout << "El tamaño del entero 'a' es: " << t << " bytes\n"
        << "El tamaño de un entero es: " << sizeof(int) << " bytes\n";
}
```

*Ejecución del programa:*

```
El tamaño del entero 'a' es: 4 bytes
El tamaño de un entero es: 4 bytes
```

Observe que los paréntesis son opcionales, excepto cuando el operando se corresponde con un tipo de datos. El operador **sizeof** se puede aplicar a cualquier variable de un tipo primitivo o de un tipo derivado, excepto a una matriz de dimensión no especificada, a un campo de bits o a una función.

## Operador coma

Un par de expresiones separadas por una coma se ejecutarán de izquierda a derecha y la expresión de la izquierda se descarta. Por ejemplo:

```
int x = 10, y = 0, t = 0;

y = (t=x, x+=1, t); // es equivalente a: y = x++
```

En la sentencia  $y = (t=x, x+=1, t)$ , se evalúan las tres expresiones especificadas entre paréntesis en el orden en el que están, y se asigna a  $y$  el valor que resulte de la última expresión, esto es, de  $t$ , las otras dos se descartan.

Otro ejemplo. Supongamos las siguientes llamadas a las funciones  $f1$  y  $f2$ :

```
f1(a++, b-a); // f1 tiene dos parámetros
f2((a++, b-a)); // f2 tiene un sólo parámetro
```

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

La función *f1* tiene dos parámetros y el orden de evaluación no está definido; dichos parámetros reciben los valores *a* y *b-a*, después *a* se incrementa. En cambio, la función *f2* tiene un solo parámetro (observe los paréntesis internos) que recibe el valor de *b-a*, previamente *a* se incrementa.

### Operador dirección-de

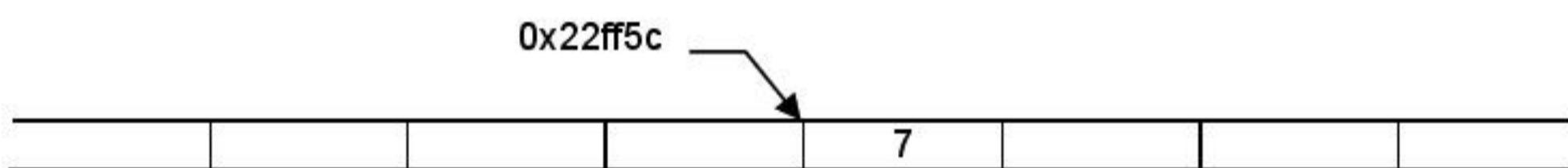
El operador & (dirección de) permite obtener la dirección de su operando. Por ejemplo:

```
int a = 7; // la variable entera 'a' almacena el valor 7
cout << "dirección de memoria = " << &a 3e2ff75c30d222a35aca2773f3e6e40d
    << ", dato = " << a << '\n'; ebrary
```

El resultado de las sentencias anteriores puede ser similar al siguiente:

```
dirección de memoria = 0x22ff5c, dato = 7
```

El resultado desde el punto de vista gráfico puede verlo en la figura siguiente. La figura representa un segmento de memoria de *n* bytes. En este segmento localizamos el entero 7 de cuatro bytes de longitud en la dirección 0x22ff5c. La variable *a* representa al valor 7 y la expresión *&a* es 0x22ff5c (*&a* - dirección de *a* - es la celda de memoria en la que se localiza *a*).



Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el calificador **register**, conceptos que veremos más adelante.

### Operador de indirección

El operador \* (indirección) accede a un valor indirectamente a través de una dirección (un puntero). El resultado es el valor direccionado por el operando; dicho de otra forma, el valor apuntado por el puntero.

Un *puntero* es una variable capaz de contener una dirección de memoria que indica dónde se localiza un dato de un tipo especificado (por ejemplo, un entero). La sintaxis para definir un puntero es:

```
tipo *identificador;
```

donde *tipo* es el tipo del dato apuntado e *identificador* el nombre del puntero (la variable que contiene la dirección de memoria donde está el dato).

El siguiente ejemplo declara un puntero *px* a un valor entero *x* y después asigna este valor al entero *y*.

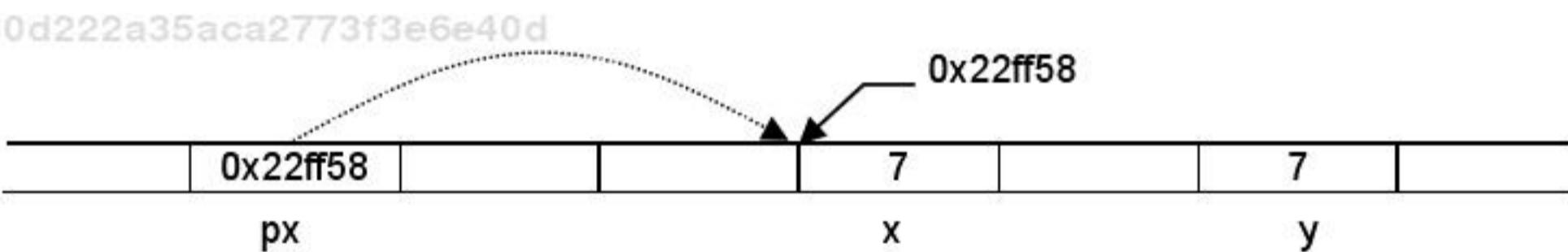
```
#include <iostream>
using namespace std;

int main()
{
    int *px, x = 7, y = 0; // px es un puntero a un valor entero.
    px = &x;               // en el puntero px se almacena la
                           // dirección del entero x.
    y = *px;              // en y se almacena el valor localizado
                           // en la dirección almacenada en px.
    cout << "dirección de memoria = " << &x << ", dato = " << x << '\n';
    cout << "dirección de memoria = " << px << ", dato = " << *px << '\n';
}
```

*Ejecución del programa:*

```
dirección de memoria = 0x22ff58, dato = 7
dirección de memoria = 0x22ff58, dato = 7
```

Observando el resultado se ve perfectamente que el contenido de *px* (*\*px*) es 7. La sentencia *y = \*px* se lee “*y igual al contenido de px*”. De una forma más explícita diríamos “*y igual al contenido de la dirección especificada por px*”. Gráficamente puede imaginarse esta situación de la forma siguiente:



Observe que una vez que *px* contiene la dirección de *x*, *\*px* y *x* hacen referencia al mismo dato, por lo tanto, utilizar *\*px* o *x* es indistinto.

### Operador referencia a

Una referencia es un nombre alternativo (un sinónimo) para un objeto. Su utilización la podremos observar en los siguientes capítulos, en el diseño de clases (por ejemplo, en el constructor copia), en el valor returned por una función para permitir que dicha función sea utilizada a ambos lados del operador de asignación (la función devuelve una referencia), o para permitir que los argumentos en la llama-

da puedan cambiar (paso de parámetros por referencia). La forma general de expresar una referencia es:

*tipo& referencia = variable*

El siguiente ejemplo declara una referencia *x* a una variable *y*.

```
int y = 10;  
int& x = y;
```

Estas sentencias declaran un entero denominado *y*, e indican al compilador que *y* tiene otro nombre, *x*. Las operaciones realizadas sobre *y* se reflejan en *x*, y viceversa. Por lo tanto, en operaciones sucesivas, es indiferente utilizar *x* o *y*.

Toda referencia, excepto las declaradas como parámetros formales en una función, debe ser siempre iniciada; de lo contrario, el compilador mostrará un error.

Una referencia no es una copia de la variable referenciada, sino que es la misma variable con un nombre diferente. Esto significa, en contra de lo que parece, que un operador no opera sobre la referencia, sino sobre la variable referenciada. Por ejemplo:

```
int conta = 0;  
int& con = conta; // con referencia a conta  
  
con++; // conta es incrementado en 1  
cout << conta << '\n'; // resultado: 1  
cout << con << '\n'; // resultado: 1
```

Obsérvese que aunque *con**++* es correcto, no se incrementa la referencia *con*, sino que *++* se aplica al objeto referenciado, que resulta ser el entero identificado por *conta*. Obsérvese también que ambas sentencias **cout** tienen el mismo efecto, puesto que los identificadores *conta* y *con* hacen referencia al mismo objeto.

Una referencia, a efectos de resultados, puede ser considerada como un puntero que accede al contenido del objeto apuntado sin necesidad de utilizar el operador de indirección (\*). Sin embargo, a diferencia de un puntero, una referencia debe ser iniciada y no puede ser desreferenciada utilizando el operador \* (contenido de). Por ejemplo, apoyándonos en el ejemplo anterior, la siguiente línea daría lugar a un error:

```
cout << *con; // indirección ilegal
```

Cuando en una declaración se especifica más de una referencia, cada uno de los identificadores correspondientes debe ser precedido por el operador &. Por ejemplo:

```
int m = 10, n = 20;
int& x = m, &y = n, z = n;
```

Este ejemplo define dos referencias, *x* e *y*, a *m* y *n*, respectivamente, y un entero *z*, al cual se le ha asignado el valor *n*.

## PRIORIDAD Y ORDEN DE EVALUACIÓN

Cuando escribimos una expresión como la siguiente,  $f = a + b * c / d$ , es porque conocemos perfectamente el orden en el que se ejecutan las operaciones. Si este orden no fuera el que esperamos tendríamos que utilizar paréntesis para modificarlo, ya que una expresión entre paréntesis siempre se evalúa primero.

Esto quiere decir que el compilador C++ atribuye a cada operador un nivel de prioridad; de esta forma puede resolver qué operación se ejecuta antes que otra en una expresión. Esta prioridad puede ser modificada utilizando paréntesis. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos. Como ejemplo de lo expuesto, la expresión anterior puede escribirse también así:  $f = (a + ((b * c) / d))$ , lo cual indica que primero se evalúa  $b * c$ , el resultado se divide por *d*, el resultado se suma con *a* y finalmente el resultado se asigna a *f*.

La tabla que se presenta a continuación resume las reglas de prioridad y asociatividad de todos los operadores. Las líneas se han colocado de mayor a menor prioridad. Los operadores escritos sobre una misma línea tienen la misma prioridad.

Operador	Asociatividad
::	ninguna
() [] . -> v++ v-- ..._cast typeid	izquierda a derecha
- + ~ ! * & ++v --v sizeof new delete (tipo)	derecha a izquierda
->* .*	izquierda a derecha
* / %	izquierda a derecha
+	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha

==	!=	izquierda a derecha
&		izquierda a derecha
^		izquierda a derecha
		izquierda a derecha
&&		izquierda a derecha
		izquierda a derecha
? :		derecha a izquierda
= *= /= %= += -= <<= >>= &=  = ^=		derecha a izquierda
,		izquierda a derecha

En C++, todos los operadores binarios excepto los de asignación son evaluados de izquierda a derecha. En el siguiente ejemplo, primero se asigna *z* a *y* y a continuación *y* a *x*.

```
int x = 0, y = 0, z = 15;  
x = y = z; // resultado x = y = z = 15
```

## CONVERSIÓN ENTRE TIPOS DE DATOS

Anteriormente mencionamos que cuando C++ tiene que evaluar una expresión en la que intervienen operandos de diferentes tipos, primero convierte, sólo para realizar las operaciones solicitadas, los valores de los operandos al tipo del operando cuya precisión sea más alta. Cuando se trate de una asignación, convierte el valor de la derecha al tipo de la variable de la izquierda siempre que no haya pérdida de información; en otro caso, C++ avisará de tal hecho.

Las reglas que se exponen a continuación se aplican en el orden indicado, para cada operación binaria perteneciente a una expresión (dos operandos y un operador), siguiendo el orden de evaluación expuesto en la tabla anterior.

1. Si un operando es de tipo **long double**, el otro operando es convertido a tipo **long double**.
2. Si un operando es de tipo **double**, el otro operando es convertido a tipo **double**.
3. Si un operando es de tipo **float**, el otro operando es convertido a tipo **float**.

4. Un **char** o un **short**, con o sin signo, se convertirán a un **int**, si el tipo **int** puede representar todos los valores del tipo original, o a **unsigned int** en caso contrario.
5. Un **wchar\_t** o un tipo enumerado se convierte al primero de los siguientes tipos que pueda representar todos los valores: **int**, **unsigned int**, **long** o **unsigned long**.
6. Un **bool** es convertido en un **int**; **false** se convierte en 0 y **true** en 1.
7. Si un operando es de tipo **unsigned long**, el otro operando es convertido a **unsigned long**.
8. Si un operando es de tipo **long**, el otro operando es convertido a tipo **long**.
9. Si un operando es de tipo **unsigned int**, el otro operando es convertido a tipo **unsigned int**.

Por ejemplo:

```
long a;
unsigned char b;
int c;
float d;
int f;
// ...
f = a + b * c / d;
```

En la expresión anterior se realiza primero la multiplicación, después la división y, por último, la suma. Según esto, el proceso de evaluación será de la forma siguiente:

1. *b* es convertido a **int** (paso 4).
2. *b* y *c* son de tipo **int**. Se ejecuta la multiplicación (\*) y se obtiene un resultado de tipo **int**.
3. Como *d* es de tipo **float**, el resultado de *b* \* *c* es convertido a **float** (paso 3). Se ejecuta la división (/) y se obtiene un resultado de tipo **float**.
4. *a* es convertido a **float** (paso 3). Se ejecuta la suma (+) y se obtiene un resultado de tipo **float**.

5. El resultado de  $a + b * c / d$ , para ser asignado a  $f$ , es pasado a entero por truncamiento, esto es, eliminando la parte fraccionaria. Para esta operación el compilador mostrará un aviso indicando que implica una pérdida de precisión.

Cuando el compilador C++ requiere realizar una conversión y no puede, o bien detecta que se incurre en una pérdida de precisión, avisará de tal acontecimiento. En estos casos, lo más normal es resolver tal situación realizando una conversión explícita. No obstante, la conversión explícita de tipos debe evitarse siempre que se pueda porque siempre que se utiliza anula el sistema de chequeo de tipos de C++.

C++ permite una conversión explícita (conversión forzada) del tipo de una expresión mediante una construcción denominada *cast*, que puede expresarse de alguna de las formas siguientes:

- `static_cast<T>(v)`. Este operador convierte la expresión  $v$  al tipo  $T$ . No examina el objeto que convierte. Se utiliza para realizar conversiones entre tipos relacionados para las que el compilador aplicará una verificación mínima de tipos. Por ejemplo, entre punteros o entre un tipo real y otro entero.
- `reinterpret_cast<T>(v)`. Este operador convierte la expresión  $v$  al tipo  $T$ . Se utiliza para realizar conversiones entre tipos no relacionados. Se trata de conversiones peligrosas. Por ejemplo, este operador permitiría realizar una conversión de `double *` a `int *` que `static_cast` no permite.
- `dynamic_cast<T>(v)`. Este operador convierte la expresión  $v$  al tipo  $T$ . Se utiliza para realizar conversiones verificadas durante la ejecución, examinando el tipo del objeto que convierte. Si la conversión es entre punteros y durante la ejecución no se puede realizar, devuelve un cero (puntero nulo).
- `const_cast<T>(v)`. Este operador convierte la expresión  $v$  al tipo  $T$ . Se utiliza para eliminar la acción ejercida por el calificador `const` sobre  $v$ .

Por ejemplo, el siguiente programa escribe la raíz cuadrada de  $i/2$  para  $i$  igual a 9. Previamente, para obtener un resultado real de la división  $i/2$  se convierte el entero  $i$  a **double**.

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

using namespace std;

int main()
{
    int i = 9;
```

3e2ff75c30d222a35aca2773f3e6e40d  
ebrary

```
double r = 0;  
  
r = sqrt(static_cast<double>(i)/2);  
cout << "La raíz cuadrada es " << r << '\n';  
}
```

*Ejecución del programa:*

*La raíz cuadrada es 2.12132*

## EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

- 1) ¿Cuál de las siguientes expresiones se corresponde con una secuencia de escape?
  - a) ESC.
  - b) \n.
  - c) \0x07.
  - d) n.
- 2) Los tipos primitivos en C++ son:
  - a) int, float y bool.
  - b) bool, char, short, int, long, float y double.
  - c) char, short, int, long, float, double, long double, wchar\_t y bool.
  - d) caracteres, variables y constantes.
- 3) C++ asume que un tipo enumerado es:
  - a) Un tipo entero.
  - b) Un tipo real.
  - c) Un tipo nuevo.
  - d) Una constante.
- 4) 01234 es un literal:
  - a) Decimal.
  - b) Octal.
  - c) Hexadecimal.
  - d) No es un literal.