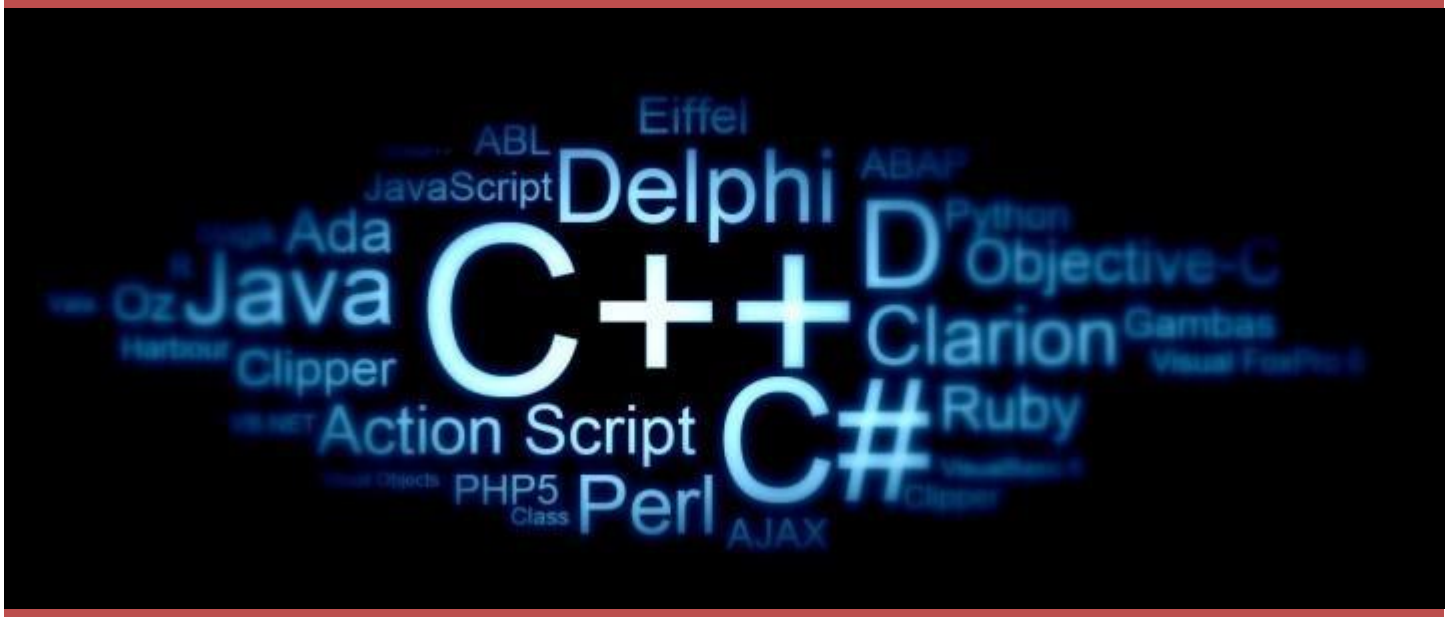
ESBA

BARRIO NORTE

HERRAMIENTAS DE PROGRAMACION

LIC. Eduardo Shimoyama



Tema 4 – Elementos del lenguaje C.

CONTINUACION DE LA UNIDAD II:

DESARROLLO DE LA PROGRAMACION

TEMA 4 – 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 la UNIDAD I. 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, limítese 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

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

char **a***;*

El texto que no aparece en negrita, significa que ahí debe ponerse la información indicada por ese texto. Por ejemplo,

typedef *declaración_tipo sinonimo[, sinonimo] ...;*

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 "1", se elige una, la necesaria dentro de la sentencia. Por ejemplo:

constante_entera[{LIUIUL}]

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:

```
main() {printf (“HOLA, MUNDO \n”);}
```

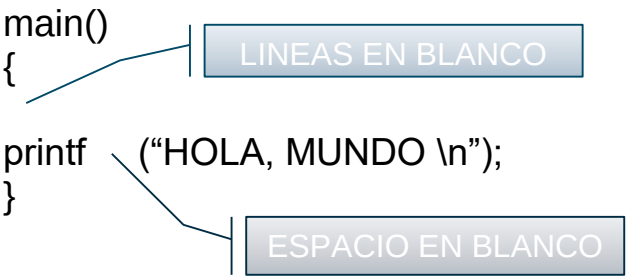
puede escribirse de una forma más legible así:

```
main()
{

printf     (“HOLA, MUNDO \n”);
}
```

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

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 lo que un usuario de C debe hacer para desarrollar un programa.



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 un array; para especificar una determinada operación aritmética, lógica o de relación; etc. Son los siguientes:

' • ; : ? 0 U ( ) [I { } < ! ¡ \ - + # % & A * - >

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 impresora)
\b	Retroceso *(*backspace*)*
\r	Retomo de carro sin avance de línea
\f	Alimentación de página (sólo para impresora)
\a	Alerta, pitido
\'	Comilla simple
\"	Comilla doble
\\	Barra invertida *(backslash)*
\ddd	Carácter ASCII. Representación octal
\ddd	Carácter ASCII. Representación hexadecimal

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

TIPOS DE DATOS

Recuerde las operaciones aritméticas que realizaba el programa aritmeti.c que vimos en el capítulo anterior. Por ejemplo, una de las operaciones que realizábamos era la suma de dos valores:

```
dato1 = 20;

dato2 = 10;

resultado = dato1 + dato2      // ahora la variable resultado vale =30
```

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;

La declaración anterior le dice al compilador C que dato1, dato2 y resultado son de tipo entero (int).

Hay dos clases de tipos: tipos fundamentales y tipos derivados.

Tipos fundamentales

Hay varios tipos fundamentales de datos. Los ficheros de cabecera limits.h y float.h especifican los valores máximo y mínimo para cada tipo. Los podemos clasificar en:

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

Tipos reales:
Float, double y long double.

Otros:
Void

Cada tipo entero puede ser calificado por las palabras clave signed o unsigned, lo que da lugar a los siguientes tipos extras:

signed char, unsigned char
signed short, unsigned short
signed int, unsigned. int
signed long, unsigned long

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.

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

```
signed x;    /* es equivalente a */
signed ints x;

unsigned y;/* es equivalente a */
unsigned int y;
```

char (carácter - 1byte)

El tipo char es utilizado para almacenar un valor entero en el rango -128 a 127.

Los valores 0 a 127 son equivalentes a un carácter del código ASCII. El tipo char es la abreviación de signed char.

De forma análoga el tipo unsigned char puede almacenar valores en el rango de 0 a 255, valores correspondientes a los números ordinales de los 256 caracteres ASCII.

El siguiente ejemplo declara una variable car de tipo char. Cuando trabajemos con esta variable, deberemos tener en cuenta que sólo puede contener valores enteros entre ---128 y 127.

char car;

A continuación, se declara la variable a de tipo char a la que se le asigna el caracter 'z' como valor inicial (observe que hay una diferencia entre 'z' y z; z entre comillas simples es interpretada por el compilador C como un valor, un carácter, y z sin comillas sería interpretada como una variable). Así mismo, sé declara la variable b de tipo signed char con un valor inicial de 7 expresado en exadecimal (0x07) y la variable c de tipo unsigned char con un valor inicial de 32.

```
char a = 'z';
signed char b 0x07;
unsigned char c = 32;
```

Las definiciones anteriores son equivalentes a las siguientes:

```
char a = 122;              /* la z es el ASCII 122 */
signed char b = 7;         /* 7 en base 16 (0x07) es 7 en base 10 */
unsigned char c = ' ';     /* el espacio en blanco es el ASCII 32 */
```

La razón es que un carácter es representado internamente por un entero, que puede ser expresado en decimal, hexadecimal u octal (vea en los apéndices del libro la tabla de caracteres ASCII).

short (entero formato corto - 2 bytes)

El tipo short, abreviación de signed short int, proporciona un entero en el rango de valores:

-32768 a 32767 (-215 a 215-1)

De forma similar el tipo unsigned short puede almacenar valores en el rango 0 a 65535 (O a 216-1).

El siguiente ejemplo declara i y j, como variables enteras que pueden tomar valores entre -32768 y 32767.

```
short i, j;
```

Otros ejemplos son:

```
short int a = -500;
signed short b = 1990;
unsigned short int c = 0xf000;
```

int (entero - 2 o 4 bytes)

Un int, abreviación de signed int, es para C un número sin punto decimal. El tamaño en bytes depende de la arquitectura de la máquina. Igualmente ocurre con el tipo unsigned int. Por ejemplo, para una máquina con un procesador de 16 bits el rango de valores es de:

-32768 a 32767 (-215 a 215-1) para el tipo int

0 a 65535 (0 a 2^{16}-1) para el tipo unsigned

El uso de enteros produce un código compacto y rápido. Para una máquina de 16 bits este tipo es equivalente al tipo short y solamente ocupa 2 bytes de memoria.

En general, podemos afirmar que:

tamaño(short) ≤ tamaño(int)

El siguiente ejemplo declara las variables n y x de tipo entero.

```
int n, x;
```

Otros ejemplos son:

```
int a = 2000;
signed int b        -30;
unsigned int c = Oxf003; unsigned d;
```

long (entero formato largo- 4 u 8 bytes)

El tipo long, abreviación de signed long int, es idóneo para aplicaciones de gestión. Al igual que los tipos anteriores, son números sin punto decimal. Para el caso de que tengan cuatro bytes de longitud, el rango de valores es el siguiente:

-2147483648 a 2147483647 (-231 a 231-1) para el tipo long
0 a 4294967295 (0 a 2^{32}-1) para el tipo unsigned long

En general, podemos afirmar que:

tamaño(int) ≤ tamaño(long)

El siguiente ejemplo declara las variables n y m de tipo entero, pudiendo tomar valores entre -2147483648 y 2147483647.

```
long n, m;
```

Otros ejemplos son:

```
long a = -1L; /* L indica que la constante -1 es long */
signed long b = 125;
unsigned long int c = Ox1f00230f;
```

enum

La declaración de un tipo enumerado es simplemente una lista de valores que pueden ser tomados por una variable de ese tipo. Los

valores de un tipo enumerado se representarán con identificadores, que serán las constantes del nuevo tipo. Por ejemplo:

```
enum dia_semana:
                    {

                    lunes,
                    martes,
                    miercoles,
                    jueves,
                    viernes,
                    sabado,
                    domingo

                    } hoy;


enum dia_semana ayer;
```

Este ejemplo declara las variables hoy y ayer del tipo enumerado dia_semana. Estas variables pueden tomar cualquier valor de los especificados, lunes a domingo. Los valores de las constantes comienzan en cero y aumentan en uno según se lee la declaración de arriba a abajo o de izquierda a derecha. Según esto el valor de lunes es 0, el valor de martes es 1, etc.

Creación de una enumeración

Crear una enumeración es definir un nuevo tipo de datos, denominado tipo enumerado y declarar una variable de este tipo. La sintaxis es la siguiente:

```
    enum tipo_enumerado
                    {
                    /* definición de nombres de constantes enteras */
                    } ;
```

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

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

```
                enum tipo_enumerado [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.

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

color = azul;
```

Como ya hemos dicho, cada identificador de la lista de constantes en una enumeración, tiene asociado un valor. Por defecto, el primer identificador tiene asociado el valor 0, el siguiente el valor 1, y así sucesivamente. Según esto,

color = verde; es equivalente a color = 3;

Nota: Para ANSI C un tipo enumerado es un tipo int. Sin embargo, 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 un tipo enumerado, sino que, habría que hacer una conversión explícita de tipo (vea "Conversión explícita del tipo de una expresión" al final de este capítulo).

```
color= (colores)3; /* conversión explícita al tipo colores */
```

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 un 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 los miembros de una enumeración se les aplica las siguientes reglas:

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

EJEMPLO // enum.c

```
#include <stdio.h>
enum colores
{
azul, amarillo, rojo, verde, blanco, negro
};

main()
{
enum colores color;

// Ejemplo con el color amarillo, pueden utilizar otros para ver diferentes valores
color=amarillo;

// Visualizar un color
printf ("%d\n", color);

}
```

Asignamos el valor de amarillo a color; como antes hemos indicado que no es posible, por ejemplo, asignar a la variable color directamente el valor verde, sino que hay que asignarle la constante entera 3 equivalente. Igualmente, printf no escribirá verde, sino que escribirá el valor al que corresponde esa variable,es decir, 3. Según esto, se preguntará ¿qué aportan, entonces, los tipos enumerados? Los tipos enumerados 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, ...". Pueden copiar estas lineas de codigo y compilarlo para poder ver los resultados obtenidos.

float (reales de precisión simple - 4 bytes)

Los datos reales de precisión simple son los más recurridos en un lenguaje de programación. Un valor real de precisión simple es un número que puede tener un punto decimal y que puede estar comprendido en el rango de:

-3.402823E+38 a -1.175494E-38 para números negativos
1.175494E-38 a 3.402823E+38 para números positivos

Un número real de precisión simple no tiene más de 7 dígitos significativos. El siguiente ejemplo declara la variable x de tipo real de precisión simple.

```
float x;
```

Otros ejemplos son:

```
float a = 3.14159;
float b = 2.2e-5; /* 2.2e-5 = 2.2 x 10-5   */
```

double (reales de precisión doble - 8 bytes)

Un dato real de precisión doble es un valor que puede tener un punto decimal y puede estar comprendido en el rango:

-1.79769E+308 a -2.22507E-308 para números negativos
2.22507E-308 a 1.79769E+308 para números positivos

Un valor real de precisión doble puede tener hasta 16 dígitos significativos, lo que da lugar a cálculos más exactos.

El siguiente ejemplo declara la variable x de tipo real de precisión doble.

```
double x;
```

Otros ejemplos son:

```
double a = 3,1415926;
double b = 2.2e-8;
```

long double (reales de precisión doble formato largo - 10 bytes)

Los valores para este tipo están comprendidos en el rango de:

-1.189731E+4932 a -3.362103E-4932 para números negativos
3.362103E-4932 a 1.189731E+4932 para números positivos

Un número real de precisión doble formato largo puede tener hasta 19 dígitos significativos. Algunos ejemplos son:

```
long double x;
long double y= 3.17e+425;
```

void

El tipo void especifica un conjunto vacío de valores. En realidad, void no es un tipo, aunque por la forma de utilizarlo si lo comparamos con la forma de utilizar los otros tipos fundamentales, se considera como tal. Por esta razón, no se puede declarar una variable de tipo void.

```
void a; /* error: no se puede declarar una variable de tipo void */
```

El tipo void se utiliza:

- Para indicar que una función no acepta argumentos. En el siguiente ejemplo, void indica que la funciónfx no tiene argumentos.

```
double fx(void);
```

- Para declarar funciones que no retoman un valor. En el siguiente ejemplo, void indica que la función fy no retoma un valor.

```
void fy (int, int);
```

- Para declarar un puntero genérico, como veremos más adelante; esto es, un puntero a un objeto de tipo aún desconocido.

```
void *p;
```

Los ejemplos anteriores declaran la función fx, como una función sin argumentos que devuelve un valor de tipo real de doble precisión; la

función fx, como una función con dos argumentos de tipo int que no devuelve valor alguno; y un puntero genérico p.

Tipos derivados

Los tipos derivados son construidos a partir de los tipos fundamentales. Algunos de ellos son: punteros, estructuras, uniones, arrays y funciones. Cada uno de estos tipos será estudiado con detalle en capítulos posteriores.

NOMBRES DE TIPO

Utilizando la declaración typedef podemos declarar nuevos nombres de tipo de datos; esto es, sinónimos de otros tipos ya sean fundamentales 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, fundamental 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 fundamental unsigned long.

typedef unsigned long ulong;

De acuerdo con esta declaración,

unsigned long dni; es equivalente a 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.

El siguiente ejemplo declara el tipo enumerado ***t_colores*** y define la variable color de este tipo

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

typedef enum colores t_colores;

main()
{
t_colores color;
//   ...
}
```

La declaración del tipo t_colores puede realizarse también así:

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

o simplemente así:

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

CONSTANTES

Una constante es un valor que, una vez fijado por el compilador, no cambia durante la ejecución del programa. Una constante en C puede ser un entero, un real, un carácter o una cadena de caracteres.

Constantes enteras

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

En general, si la constante es positiva, el signo + es opcional y si es negativa, lleva el signo-. El tipo de una constante entera depende de su base, de su valor y de su sufijo. La sintaxis para especificar una constante entera es:

{[+]1-}constante_entera[{LIUIUL}]

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

También se puede indicar explícitamente el tipo de una constante entera, 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. Por ejemplo,

1522U constante entera unsigned int
1000L constante entera de tipo long
325UL constante entera de tipó unsigned long

Una constante decimal puede tener uno o más dígitos del O a 9, de los cuales el primero de ellos es distinto de cero. Por ejemplo:

4326 constante entera int
432600 constante entera long

Una ***constante octal*** puede tener 1 o más dígitos del 0 a 7, precedidos por 0 (cero). Por ejemplo:

0326 constante entera int

Una ***constante hexadecimal*** puede tener 1 o más caracteres del 0 a 9 y de la A a la F (en mayúsculas o en minúsculas) precedidos por 0x o 0X (cero más x). Por ejemplo:

256 especifica el n° 256 en decimal
0400 especifica el n° 256 en octal
0x100 especifica el n° 256 en hexadecimal
-0400 especifica el n° -256 en octal
-0x100 especifica el n° -256 en hexadecimal

Constantes reales

Una constante real está formada 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.

{[+] 1-} 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 x 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

Una constante real tiene siempre tipo **double**, a no ser que se añada a la misma una f o F, en cuyo caso será de tipo **float**, o una l o L para indicar que es de tipo long double. Por ejemplo:

17.24F constante real de tipo float

Constante de un solo carácter

Las constantes de un solo carácter son de tipo char. Este tipo de constantes 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 ’ nueva linea
‘\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.

Constante de caracteres

Una constante de caracteres es una cadena de caracteres encerrados entre comillas dobles. Por ejemplo,

"Esto es una constante de caracteres" "3.1415926"
"Paseo Pereda 10, BARILOCHE"

En el ejemplo siguiente el carácter \n fuerza a que la cadena "o pulse Entrar" se escriba en una nueva línea.

```
printf("Escriba un número entre 1 y 5\n O pulse Entrar");
```

Cuando una cadena de caracteres es demasiado larga puede utilizarse el carácter "\" como carácter de continuación. Por ejemplo,

```
printf("Esta cadena de caracteres es dema\
siado larga.\n");
```

Este ejemplo daría lugar a una sola línea:

Esta cadena de caracteres es demasiado larga.

Dos o más cadenas separadas por un espacio en blanco serían concatenadas en una sola cadena.

Por ejemplo:

```
printf("Primera  cadena, "
"segunda cadena.\n");
```

Este ejemplo daría lugar a una sola cadena y se veria asi por pantalla:

Primera cadena, segunda cadena.

Los caracteres de una cadena de caracteres son almacenados en localizaciones sucesivas de memoria. Cada carácter ocupa un byte. Cada cadena de caracteres es finalizada automáticamente por el carácter nulo representado por la secuencia de escape \0.
Por ejemplo, la cadena "***hola***" sería representada en memoria así:

h	o	l	a	\0

IDENTIFICADORES

Los identificadores son nombres dados a constantes, variables, tipos, funciones y etiquetas de un programa. La sintaxis para formar un identificador es la siguiente:

{letral_}[{letraldígitol_}]...

lo cual indica que un identificador consta de uno o más caracteres (letras, dígitos y el carácter de subrayado) y que el primer carácter debe ser una letra o el carácter de subrayado.

Las letras pueden ser mayúsculas o minúsculas. Para C una letra mayúscula es un carácter diferente a esa misma letra en minúscula. Por ejemplo, los identificadores Suma, suma y SUMA son diferentes.

Los identificadores pueden tener cualquier número de caracteres pero dependiendo del compilador que se utilice solamente los n caracteres primeros (n;:::31) son significativos. Esto quiere decir que un identificador es distinto de otro cuan- do difieren al menos en uno de los n primeros caracteres. Algunos ejemplos son:

Suma
Suma
Calculo_Numeros_Primos
fn_ordenar
ab123

PALABRAS CLAVE

Las palabras clave son identificadores predefinidos que tienen un significado especial para el compilador C. Un identificador definido por el usuario, no puede tener el mismo nombre que una palabra clave. El lenguaje C, tiene las siguientes palabras clave:

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

auto	double	int	struct
break	Else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

COMENTARIOS

Un comentario es un mensaje a cualquiera que lea el código fuente. Añadiendo comentarios se hace más fácil la comprensión del programa. La finalidad de los comentarios es explicar el código fuente. Casi todos los compiladores C soportan comentarios estilo C y estilo C++.

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

```
main()  /* Función principal */

/* Este es un comentario
*  que ocupa varias
*  líneas.
*/
```

Un comentario estilo C++ comienza con los caracteres // y termina al final de la línea. Estos comentarios no pueden ocupar más de una línea. Por ejemplo,

```
main()  // Función principal
```

Un comentario estilo C puede aparecer en cualquier lugar donde se permita aparecer un espacio en blanco. El compilador trata un comentario como a un espacio en blanco.

DECLARACIÓN DE CONSTANTES SIMBÓLICAS

Declarar una constante simbólica significa decirle al compilador C el nombre de la constante y su valor. Esto se hace generalmente antes de la función main utilizando la directriz #define, cuya sintaxis es así:

```
#define NOMBRE VALOR
```

El siguiente ejemplo declara la constante real Pl 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".

```
#define PI 3.14159
#define NL '\n'
#define MENSAJE "Pulse una tecla para continuar\n"
```

Observe que no hay un punto y coma después de la declaración. Esto es así, porque una directriz no es una sentencia e, sino una orden para el preprocesador.

El tipo de una constante es el tipo del valor asignado. Suele ser habitual escribir el nombre de una constante en mayúsculas.

Constantes C++

C++ y algunos compiladores e admiten una forma adicional de declarar una constante; anteponer el calificador const al nombre de la constante. Utilizando el calificador const se le dice al compilador e el tipo de la constante, su nombre y su valor. Por ejemplo:

```
const int K= 12;
```

El ejemplo anterior declara la constante entera K con el valor 12.

Una vez que se haya declarado un objeto constante no se le puede asignar un valor. Por ello, al declararlo debe ser inicializado. Por ejemplo, como K ha sido declarado como constante, las siguientes sentencias darían lugar a un error:

```
K = 100;     /* error */
K++;         /* error */
```

Entonces ¿Por qué y para que utilizar constantes?:

Utilizando constantes es más fácil modificar un programa. Por ejemplo, supongamos que el programa utiliza una constante de valor 100, veinte veces. Si hemos declarado una constante K = 100 y posteriormente necesitamos cambiar el valor de la constante a 120, tendremos que modificar una sola línea, la que declara la constante. En cambio, si no hemos declarado K, sino que hemos utilizado el valor 100 directamente veinte veces, tendríamos que hacer veinte cambios.

DECLARACIÓN DE UNA VARIABLE

El valor de una variable, a diferencia de una constante, puede cambiar a lo largo de la ejecución de un programa. Cada variable de un programa, debe declararse antes de ser utilizada.

La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo. El tipo determina los valores que puede tomar la variable así como las operaciones que con ella pueden realizarse.

La sintaxis correspondiente a la declaración de una variable es la siguiente:

tipo identificador[, identificador] ...

donde tipo especifica el tipo fundamental o derivado de la variable (char, int, float, double, ...) e identificador indica el nombre de la variable. Por ejemplo:

```
int contador;
main()
{
int dia, mes, anyo;
float suma, precio;

//   ...
}
```

El ejemplo anterior declara cuatro variables de tipo int y dos variables de tipo float. Observe 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, al principio del mismo en ANSI C y en cualquier parte en caso de C++.

En nuestro ejemplo, se ha declarado la variable contador antes de la función main, fuera de todo bloque, y las variables dia, mes, anyo, suma y precio dentro del cuerpo de la función, dentro de un bloque de sentencias.

Una variable declarada fuera de todo bloque es por defecto global y es accesible en el resto del fichero fuente en el que está declarada. Por el contrario, una variable declarada dentro de un bloque, es por defecto local y es accesible solamente dentro de éste. Para comprender esto,

piense que generalmente en un programa habrá más de un bloque de sentencias. No obstante, esto lo veremos con más detalle próximamente.

Según lo expuesto, la variable contador es global y las variables dia, mes, anyo, suma y precio son locales.

Inicialización de una variable

Si queremos que algunas o todas las variables que intervienen en un programa tengan un valor inicial cuando éste comience a ejecutarse, tendremos que inicializar dichas variables. Una variable puede ser inicializada, cuando se declara o, si está declarada dentro de un bloque, a continuación de ser declarada. A diferencia de las constantes, este valor puede cambiarse a lo largo de la ejecución del programa.
Por ejemplo:

```
int contador = 1;
main()
{
int dia = 20, mes = 9, anyo = 2014;
float suma = O, precio;
precio = 100;
//   ...
}
```

No hay ninguna razón para no inicializar una variable. Tiene que saber que el compilador C inicializa automáticamente las variables globales a cero, pero no hace lo mismo con las variables locales. Las variables locales no son inicializadas por el compilador por lo que tendrán un valor, para nosotros, indefinido; se dice entonces que contienen basura (un valor que en principio no sirve).

EXPRESIONES NUMÉRICAS

Una expresión es una secuencia de operadores y operandos que especifican una operación determinada. Por ejemplo,

```
a++
suma += e
cantidad * precio
7 * sqrt(a) - b / 2    //(sqrt indica raíz cuadrada)
```

OPERADORES

Los operadores son símbolos que indican como son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, lógicos, relacionales, unitarios, lógicos para manejo de bits, de asignación, operador condicional y otros.

Operadores aritméticos

Los operadores aritméticos los utilizamos para realizar operaciones matemáticas y son los siguientes:

OPERADOR	OPERACIÓN
+	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.

El siguiente ejemplo muestra como utilizar estos operadores. Como ya hemos venido diciendo, observe que primero se declaran las variables y después se reali- zan las operaciones deseadas con ellas.

```
main()
{
int a=10, b=3, c;   /*<- PRESTAR ATENCION las variables se separan
                       por “comas” mientras que al finalizar la
                       enunciacion se utiliza un “punto y coma” */
float x=2.0, 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) */
}
```

Cuando en una operación aritmética los operandos son de diferentes tipos, ambos son convertidos al tipo del operando de precisión más alta; por ejemplo, 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.

El resultado obtenido en una operación aritmética es convertido al tipo de la variable que almacena dicho resultado. Por ejemplo, del resultado de x/y sólo la parte entera es asignada a e, ya que e es de tipo int. Esto indica que los reales son convertidos a enteros, truncando la parte fraccionaria.

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.

Operadores lógicos

El resultado de una operación lógica (AND, OR y NOT) es un valor verdadero o falso (1 o 0). Por definición, un valor distinto de cero es siempre verdadero y un valor cero es siempre falso. Los operadores lógicos de C++ son los siguientes:

Operador	Operación
&&	AND. Da como resultado el valor lógico 1 si ambos operandos son distintos de cero. Si uno de ellos es cero el resultado es el valor lógico 0. Si el primer operando es igual a cero, el segundo operando no es evaluado.
||	OR. El resultado es 0 si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de 0, el resultado es 1. Si el primer operando es distinto de cero, el segundo operando no es evaluado (el carácter 1 es el ASCII 124).
!	NOT. El resultado es 0 si el operando tiene un valor distinto de cero, y 1 en caso contrario.

El resultado de una operación lógica es de tipo int. Los operandos pueden ser enteros, reales o punteros. Por ejemplo,

```
main()
{
int p = 10, q = 0, r = 0; /*<- ATENCION nuevamente las variables se
                              separan por “comas” mientras que al finalizar
                              la enunciacion se utiliza un “punto y coma” */

r = p && q;   /* da como resultado 0 */
r = p | | q;  /* da como resultado 1 */
r =! p;       /* da como resultado 0 */
}
```

Operadores de relación

El resultado de una operación de relación es un valor verdadero o falso (1 o 0). Los operadores de relación son los siguientes:

Operador	Operación
<	Primer operando *menor que* el segundo.
>	Primer operando *mayor que* el segundo.
<=	Primer operando *menor o igual que* el segundo.
>=	Primer operando *mayor o igual que* el segundo.
! =	Primer operando *distinto que* el segundo.
==	Primer operando *igual que* el segundo.

Los operandos pueden ser de tipo entero, real o puntero. Por ejemplo,

```
main()
{
int x = 10, y = 0, r = 0;

r = x == y;   /* da como resultado 0*/
r = x > y;    /*da como resultado 1 */
r = x !=y;    /* da como resultado 1 */
{
```

Un operador de relación equivale a una pregunta relativa a cómo son dos operandos entre sí. Por ejemplo, la expresión x==y equivale a la pregunta ¿x es igual a y? Una respuesta sí equivale a un valor 1 (verdadero) y una respuesta no equivale a un valor O (falso).

Expresiones de Boole

Una expresión de Boole da como resultado 1 (verdadero / true) o 0 (falso / false). Desde un análisis riguroso, los operadores booleanos son los operadores lógicos && (AND), || (OR) y ! (NOT). Ahora bien, por ejemplo, piense que las comparaciones producen un resultado de tipo boolean. Quiere esto decir que el resultado de una comparación puede utilizarse como operando en una expresión de Boole. ¡Según esto, podemos enunciar que los operadores que intervienen en una expresión de Boole pueden ser de relación (<,>, <=,>=, ==, y! =) y lógicos (&&, ||, !).

El ejemplo siguiente, combina en una expresión operadores lógicos y operadores de relación. Más adelante, en este mismo capítulo, veremos la precedencia de los operadores.

```
main()
}
int r = 0, s = 0;                    /* INICIALIZO LAS VARIABLES */
float x = 15, y = 18, z = 20;
r =   (x < y) && (y <= z) || s;   /* resultado r = 1 */
{
```

El resultado es 1 (verdadero) porque x <y es verdadero (1), y<=z es verdadero (1), y s, como vale O, es falso (un valor distinto de cero es siempre verdadero y un valor cero es siempre falso). Por lo tanto, 1 && 1 || 0 = 1 || 0 = 1.

Operadores unitarios

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

Operador	Operación
-	Cambia de signo al operando (complemento a dos). El operando puede ser entero o real.
~	Complemento a 1. El operando tiene que ser entero (carácter ASCII 126).

El siguiente ejemplo muestra como utilizar estos operadores.

```
main()
{
int a = 2, b = 0, c = 0;
c = ~a;       /* resultado c = -2 */
c = ~b;       /* resultado c = -1 */
}
```

Operadores lógicos para manejo de bits

Estos operadores permiten realizar operaciones AND, OR, XOR y desplazamientos, bit por bit de los operandos.

Operador	Operación
&	Operación AND a nivel de bits.
|	Operación OR a nivel de bits (carácter ASCII 124).
∧	Operación XOR a nivel de bits.
<<	Desplazamiento a la izquierda.
>>	Desplazamiento a la derecha.

Los operandos para este tipo de operaciones tienen que ser de tipo entero (char, int, long, o enum), no pueden ser reales. Por ejemplo,

```
main()
{
unsigned char 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 están 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 */
}
```

En las operaciones de desplazamiento el primer operando es desplazado tantas posiciones como indique el segundo. Si el desplazamiento es a izquierdas, se rellena con ceros por la derecha; si el desplazamiento es a derechas, se rellena con ceros por la izquierda si el operando es de tipo unsigned, en otro caso se rellena con el bit de signo.

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 al tipo del operando de la izquierda. Incluimos aquí los operadores de incremento y decremento porque implícitamente estos operado- res realizan una asignación sobre su operando.

Operador	Operación
++	Incremento.
--	Decremento.
=	Asignación simple.
*=	Multiplicación más asignación.
/=	División más asignación.
%=	Módulo más asignación.
+=	Suma más asignación.
--	Resta más asignación.
<<=	Desplazamiento a izquierdas más asignación.
>>=	Desplazamiento a derechas más asignación.
&=	Operación AND sobre bits más asignación.
|=	Operación OR sobre bits más asignación.
∧=	Operación XOR sobre bits más asignación.

A continuación, se muestran algunos ejemplos con estos operadores.

```
main()
{
Int x = 0, n = 10, i = 1;

X++;          /* incrementa el valor de x en 1 */
++x;          /* incrementa el valor de x en 1 */
x = - - n;    /* decrementa n en 1 y asigna el resultado de x */
x = n - -;    /* asigna el valor de n a x y después decrementa n en 1 */
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 un bit a la derecha */
}
```

Operador condicional

C tiene un operador temario (?:) que se utiliza en expresiones condicionales, las cuales tienen la forma:

operando1 ? operando2 : operando3

El valor resultante de la expresión operando] debe ser de tipo entero, real o puntero. La ejecución se realiza de la siguiente forma:

- Si el resultado de la evaluación de operando] es distinta de O, el resultado de la expresión condicional es operando2.

- Si el resultado de la evaluación de operando] es O, el resultado de la expresión condicional es operando].

¿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.

```
main()
{
float a = 10.2, b = 20,5, mayor = 0;
mayor = (a > b) ?   a : b;       /* mayor de a y b */
}
```

Operador coma

Un par de expresiones separadas por una coma son evaluadas de izquierda a derecha. Todos los efectos secundarios de la expresión de la izquierda son ejecutados antes de evaluar la expresión de la derecha, a continuación, el valor de la expresión de la izquierda es descartado. El tipo y el valor del resultado son el tipo y el valor del operando de la derecha. Algunos ejemplos son:

```
aux = v1, v1 = v2, v2 = aux;
for (a = 256, b = 1; b < 512; a/=2, b *=2)
```

En las líneas del ejemplo anterior, la coma simplemente hace que las expresiones separadas por la misma se evaluen de izquierda a derecha.

```
fx (a, (b =   10, b - 3), c, d)
```

Esta línea es una llamada a una función fx. En la llamada se pasan cuatro argumentos, de los cuales, (b = 1O, b - 3) tiene un valor 7.

Operador de dirección-de

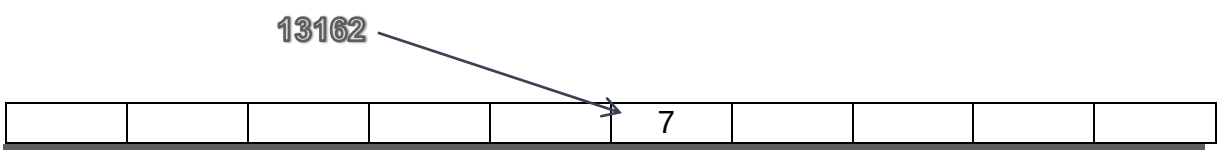
El operador & (dirección de) da la dirección de su operando. Por ejemplo,

```
Int a = 7; /* la variable entera 'a' almacena el valor 7 */
printf ("dirección de memoria= %d, dato=  %d\n", &a, a);
```

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

dirección de memoria = 13162, 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 dos bytes de longitud, en la dirección 13162. La variable a representa al valor 7 y &a (dirección de a; esto es, donde se localiza a) es 13162.



Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el calificador register.

Operador de indirección

El operador * (indirección) accede a un valor indirectamente a través de un puntero. El resultado es el valor direccionado (apuntado) por el operando.

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

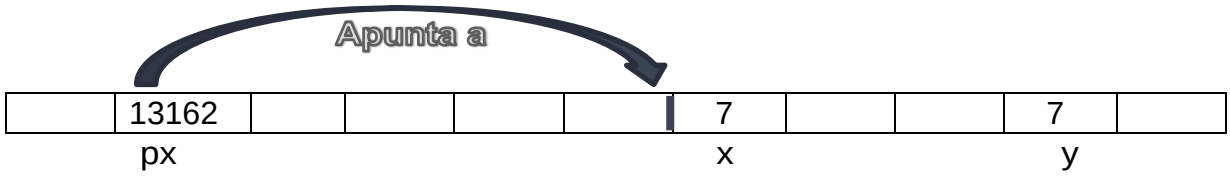
tipo *identificador;

donde tipo es el tipo del objeto apuntado e identificador el nombre del puntero.

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

```
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 de x */
y = *px;           /* a y se le asigna el valor localizado en la dirección
                      almacenada en px */
}
```

Suponiendo que la dirección de x es 13162, el resultado expresado de una forma gráfica es la siguiente:



lo que indica 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 dada por px".

Operador sizeof

El operador sizeof da como resultado el tamaño en bytes de su operando. El operando o es el identificador del objeto o es el tipo del objeto.

Por ejemplo:

```
#include <stdio.h>
main()
{
int a = 0, t = 0;
t = sizeof  a;

printf("El tamaño del entero a es: %d\n", t);

printf("El tamaño de un float es: %d\n", sizeof(float));
}
```

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 objeto de un tipo fundamental o de un tipo definido por el usuario, excepto al tipo void, a un array de dimensión no especificada, a un campo de bits o a una función.

PRIORIDAD Y ORDEN DE EVALUACIÓN

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

Una expresión entre paréntesis, siempre se evalúa primero. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos.

Operador	Operación
() [] . -> sizeof	izquierda a derecha
- ~ ! * & ++ -(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
?:	derecha a izquierda
= *= /= %= += -= <<= >>= &= |= ^=	derecha a izquierda
,	izquierda a derecha

En el siguiente ejemplo, primero se asigna *z* a *y* y a continuación *y* a *x*. Esto es así porque la asociatividad para este operador es de derecha a izquierda.

```
void main ()
{
int x =0, y = 0, z = 15;
x = y = z;            /* resultado x = y = z = 15  */
}
```

CONVERSIÓN DE TIPOS

Cuando los operandos que intervienen en una operación son de tipos diferentes, antes de realizar la operación especificada, se convierten a un tipo común, de acuerdo con las reglas que se exponen a continuación.
Las reglas que se exponen, se aplican en ese orden, para cada operación binaria perteneciente a una expresión, siguiendo el orden de evaluación expuesto anteriormente.

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) Si un operando es de tipo unsigned long, el otro operando es convertido a unsigned long.

6) Si un operando es de tipo long, el otro operando es convertido a tipo long.

7) 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 el desde 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.

CONVERSIÓN EXPLÍCITA DEL TIPO DE UNA EXPRESIÓN

En C, está permitida una conversión explícita del tipo de una expresión mediante una construcción denominada cast, que tiene la forma:

(nombre-de-tipo) expresión

La expresión es convertida al tipo especificado si esa conversión está permitida; en otro caso, se obtendrá un error. La utilización apropiada de construcciones *cast* garantiza una evaluación consistente, pero siempre que se pueda, es mejor evitarla ya que suprime la verificación de tipo proporcionada por el compilador y por consiguiente puede conducir a resultados inesperados.

Por ejemplo, la función raíz cuadrada (sqrt), espera como argumento un tipo double. Para evitar resultados inesperados en el caso de pasar un argumento de otro tipo, podemos escribir:

sqrt((double)(n+2))

En C++ una construcción cast puede expresarse también así:

nombre-de-tipo(expresión)

sintaxis similar a la llamada a una función, por lo que recibe el nombre de notación funcional. De acuerdo con esto, el ejemplo anterior lo escribiríamos así:

sqrt(double(n+2))

La notación funcional sólo puede ser utilizada con tipos que tengan un nombre simple. Si no es así, utilice typedef para asignar al tipo derivado un nombre simple. Por ejemplo,

```
typedef int * pint;
int *p = pint(0x1F5)
```