GUÍA DE SINTAXIS DEL LENGUAJE C++
(ESTÁNDAR C++ ANSI)

(Fundamentos de Programacion de Luis Aguilar)

C++ es considerado un C más grande y potente. La sintaxis de C++ es una extensión de C, al que se han añadido numerosas propiedades, fundamentalmente orientada a objetos. ANSI C ya adoptó numerosas características de C++, por lo que la migración de C a C++ no suele ser difícil.
En este apéndice se muestran las reglas de sintaxis del estándar clásico de C++ recogidas en al Annotated Manual (ARM) de Stroustrup y Ellis, así como las últimas propuestas incorporadas al nuevo borrador de C++ ANSI, que se incluyen en las versiones 3 (actual) y 4 (futura de AT&T C++).

B.1. ELEMENTOS DEL LENGUAJE

Un programa en C++ es una secuencia de caracteres que se agrupan en componentes léxicos (tokens) que comprenden el vocabulario básico del lenguaje. Estos componentes de léxico son: palabras reservadas, identificadores, constantes, constantes de cadena, operadores y signos de puntuación.

B.1.1. Caracteres

Los caracteres que se pueden utilizar para construir elementos del lenguaje (componentes léxicos o *tokens)* son:

abcdefghi jklmnopqrstuvwxyz
ABCDEFCHIJKLMNOPQRSTUVWXYZ
0123456789
caracteres espacio (blancos y tabulaciones)

B.1.2. Comentarios

C++ soporta dos tipos de comentarios. Las líneas de comentarios al estilo C y C ANSI, tal como:

```
/* Comentario estilo C */
/* Comentario mas extenso, pero también es estilo C y ANSI C */
```

El otro tipo de comentarios se pueden utilizar por los programadores de C++: La versión /*...* / se utiliza para comentarios que excedan una línea de longitud y la versión //... se utiliza, sólo, para comentarios de una línea. *Los comentarios no se anidan.*

B.1.3. Identificadores

Los *identificadores* (nombres de variables, constantes,...) deben comenzar con una letra del alfabeto (mayúscula o minúscula) o con un carácter subrayado, y pueden tener uno o más caracteres. Los caracteres segundo y posteriores pueden ser: letras, dígitos o un subrayado, no permitiéndose caracteres no alfanuméricos ni espacios.

```
tescprueba  // legal
Xl23        // legal
multi_palabra // legal
var25       // legal
l5var       // no legal
```

C++ es sensible a las mayúsculas.

```
Paga_mes    es un identificador distinto a paga_mes
```

Una buena práctica de programación aconseja utilizar identificadores significativos que ayudan a documentar un programa.

```
nombre apellidos salario precio_neto
```

B.1.4.Palabras reservadas

Las palabras reservadas o claves no se deben utilizar como identificadores, debido a su significado estricto en C++; tampoco se deben redefinir. La Tabla B. 1 enumera las palabras reservadas de C++ según el ARM(Siglas del libro de **BJARNE STROUSTRUP** en el que se definen las reglas de sintaxis del lenguaje C++ estándar).

Tabla B.1. Palabras reservadas de C++

asm*	continue	float	new*	signed	try
auto	default	for	operator*	sizeof	typedef
break	delete*	friend*	private*	static	union
case	do	goto	protected*	struct	unsigned
catch*	double	if	public*	switch	virtual*
char	else	inline*	register	template*	void
class*	enurn	int	return	this*	volatile
const	extern	long	short	throw*	while

*Estas palabras no existen en ANSI C.

Los diferentes compiladores comerciales de C++ pueden incluir, además, nuevas palabras reservadas. Estos son los casos de Borland, Microsoft y Sysmantec.

Tabla B.2. Palabras reservadas de Turbo/Borland C++

asm	_ds	interrup	short
auto	else	_loadds	signed
break	enum	long	sizeof
case	_es	_near	_ss
catch	export	near	static
_cdecl	extern	new	struct
_cdecl	far	operator	switch
char	far	pascal	template
class	float	pascal	this
const	for	private	typedef
continue	friend	protected	union
_cs	goto	pubic	unsigned
default	huge	register	virtual
delete	if	return	void
do	inline	_saveregs	volatile
_double	int	_seg	while

Tabla B.3. Palabras reservadas de Microsoft Visual C/C++ 1.5/2.0

asm	else	int	signed
auto	enum	_interrupt	sizeof
based	_except	_leave	static
break	_export	_loadds	_stcall
case	extern	long	struct
_cdecl	_far	maked	switch
char	_fastcall	_near	thread
const	_finally	_pascal	_try
continue	float	register	typedef
_declspec	for	return	union
default	_fortran	_saveregs	unsigned
dllexport	goto	_self	void
dllimport	_huge	_segment	volatile
do	if	_segname	while
doble	_inline	short	

El comité ANSI ha añadido nuevas palabras reservadas (Tabla B.4).

Tabla B.4. Nuevas palabras reservadas de ANSI C++

bool	false	reinterpretcast	typeid
cons_cast	mutable	static_cast	using
dynamic_cast	namespace	true	wchart

B.2. TIPOS DE DATOS

Los tipos de datos simples en C++ se dividen en dos grandes grupos: integrales (datos enteros) y de coma flotante (datos reales). La Tabla B.5. muestra los diferentes tipos de datos en C++,

Tabla B.5. Tipos de datos simples en C++

```
char             signed char      unsigned char
short            int              long
usigned short    unsigned         unsigned long
float            double           long double
```

Los tipos derivados en C++ pueden ser:

- enumeraciones (enum),
- estructuras (struct),
- uniones (union),
- arrays,
- clases (class y struct),
- uniones y enumeraciones anónimas,
- punteros,

B.2.1. Verificación de tipos

La verificación o comprobación de tipos en C++ es más rígida (estricta) que en C. Algunas consideraciones a tener en cuenta son:
Usar funciones declaradas. Esta acción es ilegal en C++, y está permitida en C:

```
int main()
{
      //...
      printf(x)   //C int printf(x)
                  //C++ es ilegal, ya que printf no esta declarada
      return 0;
}
```

- *Fallo al devolver un valor de una función.* Una función en C++ declarada con un tipo determinado de retomo, ha de devolver un valor de ese tipo. En C, está permitido no seguir la regla.
- *Asignación de punteros* void. La asignación de un tipo void* a un puntero de otro tipo se debe hacer con una conversación explícita en C++. En C, se realiza implícitamente.
- *Inicialización de constantes de cadena.* En C++ se debe proporcionar un espacio para el carácter de terminación nulo cuando se inicializan constantes de cadena. En C, se permite la ausencia de ese carácter

```
int main()
{
   //......
    char car[7] = “Cazorla“;    // legal en C
                                //error en C++
   //......

   return 0;
}
```

Una solución al problema que funciona tanto en C como en C++ es: `char car [] ="Cazorla";`

B.3. CONSTANTES

C++ contiene constantes para cada tipo de dato simple (integer, char,...). Las constantes pueden tener dos sufijos, u, l y f. Que indican tipos unsigned, long y float, respectivamente. Así mismo, se pueden añadir los prefijos o y ox, que representan constantes octales y hexadecimales.

```
456   0456 0x476  // constante enteras : decima1, octal , hexadecimal
1231  123ul       // constante enteras :long, unsigned, long
 ´B´  ´b´  ´4´    // constante de tipo char
3.1415f           // constante reales de diferente posición
“cadena de caracteres” // Constante de cadena
```

Las cadenas de caracteres se encierran entre comillas, y las constantes de un solo carácter se encierran entre comillas simples.

```
" " // cadena vacía '\0'
```

B.3.1. Declaración de constantes

En C++, los identificadores de variables/constantes se pueden declarar *constantes*, significando que su valor no se puede modificar. Esta declaración se realiza con la palabra reservada **const**.

```
contar double PI= 3.11416;
const char BLANCO = ´ ´;
const double PI_EG = -I;
const double DOBLE_I = 2 * PI ;
```

El modificador de tipos const se utiliza en C++, también para proporcionar protección de sólo lectura para variables y parámetros de funciones. Las funciones miembro de una clase que no modifican los miembros dato a que acceden pueden ser declarados const. Este modificador evita también que parámetros por referencia sean modificados:

```
void copy (const char *  fuente, char * destino):
```

B.4. CONVERSIÓN DE TIPOS

La conversiones explícitas se fuerzan mediante *moldes* (casts). La conversión forzosa de tipos de C tiene el formato clásico:

(tipo) expresion

C++ ha modificado la notación anterior por una notación funcional como alternativa sintáctica:

nombre del tipo (expresion)

Las notaciones siguientes son equivalentes:

```
float(x);   //notacion de casteo en C++
(float)x;   //notacion de casteo en C
```

B.5. DECLARACIÓN DE VARIABLES

En ANSI C, todas las declaraciones de variables y funciones se deben hacer al principio del programa o función. Si se necesitan declaraciones adicionales, el programador debe volver al bloque de declaraciones al objeto de hacer los ajustes o inserciones necesarios. Todas las declaraciones deben hacerse antes de que se ejecute cualquier sentencia. Así, la declaración típica en C++, `NombreTipo Nombrevariablel, Nombrevariable2`; proporciona declaraciones tales como:

```
int saldo, meses;
double clipper, salario;
```

Al igual que en C, se pueden asignar valores a las variables en C++:

```
int mes =4, dia, anio=1995;
doble salario = 45.675;
```

En C++, las declaraciones de variables se pueden situar en cualquier parte de un programa. Esta característica hace que el programador declare sus variables en la proximidad del lugar donde se utilizan las sentencias de su programa. El siguiente programa es legal en C++ pero no es válido en C:

```
#include <stdio.h>
int main ()
{
      int i ;
      for ( i= 0; i<100; ++i)
            printf ( "%d\n" , i);
      double j;
      for ( j= 1.7547; j<25.4675; j+= .001)
            printf ( "%lf\n" , j);
}
```

El programa anterior se podría reescribir, haciendo la declaración y la definición dentro del mismo bucle:

```
int main ()
{
      for ( int i= 0; i<100; ++i)
            cout<< i << endl;
      for ( int j= 1.7547; j<25.4675; j+= .001)
            cout << j <<;
}
```

B.6. OPERADORES

C++ es un lenguaje muy rico en operadores. Se clasifican en los siguientes grupos:

- Aritméticos.

- Relacionales y lógicos.
- Asignación.
- Acceso a datos y tamaño.
- Manipulación de bits.
- Varios.

Como consecuencia de la gran cantidad de operadores, se producen también una gran cantidad de expresiones diferentes.

B.6.1. Operadores aritméticos

C++ proporciona diferentes operadores que relacionan operaciones aritméticas.

Tabla B.6. Operadores aritméticos en C++

Operador	Nombre	Propósito	Ejemplo
+	Más unitario	Valor positivo de x	x = +y,+5
-	Negación	Valor negativo de x	x = –y;
+	Suma	Suma x e y	z = x + y;
-	Resta	Resta y de x	z = x – y;
*	Multiplicación	Multiplica x por y	y = x * y;
/	División	Divide x por y	z = x / y;
%	Módulo	Resto de division	z= x % y;
++		Incremento Incrementa x después de usar	x++
--		Decremento Decrementa x antes de usar	--x

Ejemplo:

```
-i              // menos unitario
+w;             // mas unitario
a * b           // multiplicación
b / c           // división
a % t;          // modulo
a + b           // suma
a - b           // resta
int a = 5/2;    // a toma el valor 2, porque a es entero
float b = 5/2;  // b toma el valor 2.5, porque b es real
```

Los operadores de incremento y decremento sirven para incrementar y decrementar en uno los valores almacenados en una variable.

```
variable ++     // postincremento
++variab1e      // preincremento
variable --     // postdecremento
--variable      // predecremento
++a o a++;  equivale a  a = a + 1;
--b o b--;  equivale a  b = b – 1;
```

Los formatos postfijos se conforman de modo diferente según la expresión en que se aplica

```
b = ++a ;   equivale a  a = a + l; b = a;
b = a--;    equivale a  b = a;   a = a - 1;

int i, j, k = 5;
k++;            // k vale 6, igual efecto que ++k
--k;            // k vale ahora 5, igual efecto que k- –
i = 4 * k++;    // k es ahora 6 e i es 20
k = 5;
j = 4* ++k;     // k es ahora 6 e i es 24
```

B.6.2. Operadores de asignación

El operador de asignación (=) hace que el valor situado a la derecha del operador se adjudique variable situada a su izquierda. La asignación suele ocurrir como parte de una expresión de asignación y las conversiones se producen implícitamente.

```
z = b + 5;  //asigne (b + 5) a variable z
```

C++ permite asignaciones múltiples en una sola sentencia. Así:

`a = b + (c=l0);` equivale a: `c = 10; a = b + c;`

otros ejemplos de expresiones válidas y no válidas son:

```
// expresiones legales       //expresiones no legales
a = 5 * (b+a);               a+3=b;
doble x = y;                 PI = 3; // siendo PI una constante
a = b = c;                   x++ = y;
```

C++ proporciona operadores de asignación que combinan operadores de asignación y otros diferentes, produciendo operadores tales como +=, /=, -=, *= y %= . C++ soporta otros de operadores de asignación para manipulación de bits.

Tabla B.7. Operadores aritméticos de asignación

```
+=    x=x+y;      x+=y;
-=    x=x-y;      x-=y;
=     x=x*y;      x*=y;
/=    x=x/y;      x/=y;
%=    x=x%y;      X%=Y;
```

Ejemplos

```
a + b;          equivale   a     a =   a + b;
a * = a+b;      equivale   a     a =   a * (a + b);
v + = e;        equivale   a     v =   v + e;
v – = e,        equivale   a     v =   v - e;
```

Expresiones equivalentes:

```
n = n+1
n+=1;
n++;
++n;
```

B.6.3. Operadores lógicos y relacionales

Los operadores lógicos y relacionales son los bloques de construcción básicos para constructores de toma de decisión en un lenguaje de programación. La Tabla B8 muestra los operadores lógicos y relacionales:

Tabla B.8. Operadores lógicos *y* relacionales

Operador	Descripción	Ejemplo
&&	AND (y) lógico	a && b
||	OR (o) lógico	c || b
!	NOT (no) lógico	!c
<	Menor que	i<c
<=	Menor o igual que	i<=0
>	Mayor que	j>50
>=	Mayor o igual que	j>=8.5
==	Igual a	x==’\0’
!=	No igual a	c!=’\n’
?:	Asignación condicional	k= (i < 5) ? i=1 ;

El operador ?: se conoce como expresión condicional, La expresión condicional es una abreviatura de la sentencia condicional if_else. La sentencia if

```
if(condicion)
            variable = exprexion1;
      else
            variable = exprexion2;
es equivalente a
variable=(condicion) ? expresion1:expresion2;
```

La expresión condicional comprueba la condición. Si esa condición es verdadera, se asigna *expresionl* a *variable;* en caso contrario, se asigna ***expresion2*** a *variable.*

Reglas prácticas

Los operadores lógicos y relacionales actúan sobre valores lógicos el valor/es puede ser o bien 0, o bien el puntero nulo, o bien 0.0; el valor *verdadero* puede ser cualquier valor distinto de cero.

La siguiente tabla muestra los resultados de diferentes expresiones.

```
x >  y      distinto a 0 si x excede a y, sino 0
x >= y      distinto a 0 si x es mayor o igual a y,sino 0
x <  y      distinto a 0 si x es menor que y, sino 0
x <= y      distinto a 0 si x es menor que o igual a y, sino 0
x == y      distinto a 0 si x es igual a y, sino 0
x != y      distinto a 0 si x e y son distintos, sino 0
!x          distinto a 0 si x es 0,sino 0
x || y      0 si ambos x e y son 0, sino distinto a 0
```

Evaluación en cortocircuito

C++, igual que C, admite reducir el tiempo de las operaciones lógicas; la evaluación de las expresiones se reduce cuando alguno de los operandos toman valores concretos.

1. *Operación lógica AND (&&)* Si en la expresión *expr1 && expr2, expr1* toma el valor cero y la operación lógica AND (y) siempre será cero, sea cual sea el valor de *expr2.* En consecuencia, *expx2* no se evaluará nunca.
2. *Operación lógica OR* (II). Si *expr1* toma un valor distinto de cero, la expresión *expr1 || expr2* se evaluará a 1, cualquiera que sea el valor de *expr2;* en consecuencia, *expr2* no se evaluará.

B.6.4. Operaciones de manipulación de bits

C++ proporciona operadores de manipulación de bits, así como operadores de asignación de manipulación de bits.

Tabla B.8. Operadores de manipulación de bits *(bitwise)*

Operador	Descripción	Ejemplo
&	AND bit a bit	*x* & 128
|	OR bit a bit	j | 64
^	XOR bitabít	j ^ 2
~	NOT bit a bit	~ j
<<	Desplazar a la izquierda (mult. por 6)	i << 3
>>	Desplazar a la derecha (div. por 8)	j >> 4

Tabla B.10. Operadores de asignación de manipulación de bits

Operador	Formato largo	Formato corto
&=	x = x & y	x& = y;
|=	x = x | y	x | = y;
^=	x = x ^ y	x ^= y ;
<<=	x = x << y	x << = y ;
>>=	x = x >> y	x >> = y ;

Ejemplo*:*

~x	Cambia *los bits 1* a 0 y *los bits 0 a 1*
y & y	*Operación lógica AND (y) bit a bit de x e y*
x | y	*Operación lógica GR (o) bit a bit de x e y*
x << x	*se desplaza a la izquierda (en y posiciones)*
x >> x	*se desplaza a* la *derecha (en y posiciones)*

B.6.5. El Operador sizeof

El operador sizeof proporciona el tamaño en bytes de un tipo de dato o variable sizeof toma el argumento correspondiente (tipo escalar, *array*, *record*, etc.). La sintaxis del operador es:

sizeof *(nombre variable | tipo de dato)*

Ejemplo:

```
int m , n[12]
sizeof(m)    // proporciona 4, en maquinas de 32 bits
sizeof(n)    // proporciona 48
sizeof (15)  // proporciona 4
sizeof (int) // da el tamaño en memoria de un int (generalmente 32 bits)
```

B.6.6. Prioridad y asociatividad de operadores

Cuando se realizan expresiones en las que se mezclan operadores diferentes es preciso establecer una precedencia (prioridad) de los operadores y la *dirección* (o secuencia) de evaluación (orden de evaluación: izquierda-derecha, derecha-izquierda), denominada ***asociatividad.*** *L*a Tabla B.11 muestra la precedencia y asociatividad de operadores.

Ejemplo:

a * b/c+d *equivale* a (a * b) / (c + d)

B.6.7. Sobrecarga de operadores

La mayoría de los operadores de C++ pueden ser sobrecargados o redefinidos para trabajar con nuevos tipos de datos. La Tabla B. 12. lista los operadores que pueden ser sobrecargados.

Tabla B.11. Precedencia *y* asociatividad de operadores

:: ( ) [ ] . ->	Izquierda-Derecha
++ --	Derecha-Izquierda
& *(dirección)* ~ *(tipo)* ¡ - +	
sizeof *(tipo)* new delete ***** *(indireccion)*	
.* ->*	Izquierda-Derecha
* / %	Izquierda-Derecha
+ —	Izquierda-Derecha
	Izquierda-Derecha
	Izquierda-Derecha
<< >>	Izquierda-Derecha
< <= > >=	Izquierda-Derecha
= = ¡=	Izquierda-Derecha
&	Izquierda-Derecha
^	Izquierda-Derecha
|	Izquierda-Derecha
&&	Izquierda-Derecha
||	Izquierda-Derecha
= += - = *= /= %= >>= <<= &= |= ^=	Derecha-Izquierda
, *(operador coma)*	Izquierda-Derecha

Tabla B.12. Operadores que se pueden sobrecargar

+ - * / % ^ & | ~ ! = < > >> << >>= <<= != == ( ) -> , []

B.7. ENTRADAS Y SALIDAS BÁSICAS

Al contrario que muchos lenguajes, C++ no tiene facilidades incorporadas para manejar entrada o salida. En su lugar, se manejan por rutinas de bibliotecas. Las clases que C++ utiliza para entrada y salida se conocen como *flujos.* Un *flujo (stream)* es una secuencia de caracteres junto con una colección de rutinas para insertar caracteres en flujos (a pantalla) y extraer caracteres de un flujo (de teclado).

B.7.1.Salida

El flujo cout es el flujo de salida estándar que corresponde a stdout en C. Este flujo se deriva de la clase ostream construida en iostream.

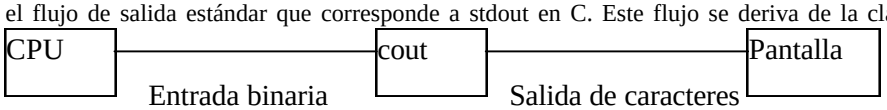


Figura B.1. Uso de flujos para salida de caracteres.

Si se desea visualizar el valor del objeto int llamado i, se escribe la sentencia:

```
cout << i;
```

El siguiente programa visualiza en pantalla una frase:

```
#include <iostream.h>
int main()
{
 cout << “Hola Mundo\n”;
}
```

Las salidas en C++ se pueden conectar en cascada, con una facilidad de escritura mayor que en C.

```
#include <isostrean.h>.
int main()
      {
      int i;
      i = 1099;
      cout << ”El valor de i es”  << i << “\n”;
      }
```

Otro programa que muestra la conexión en cascada es:

```
#include <iostream.h>
int  main(){
      int x = 45;
      double y = 496.125;
      char *c = "y es multiplicada por x=";
      cout << c << y * x << "\n”;
}
```

B.7.2. Entrada

La entrada se maneja por la clase istream. Existe un objeto predefinido istream, llamado *cin*, que se refiere al dispositivo de entrada estándar (el teclado). El operador que se utiliza para obtener un valor del teclado es el *operador de extracción* >>. Por ejemplo, si i era un objeto int**,** se escribirá:

```
cin >> i;
```

que obtiene un número del teclado y lo almacena en la variable i.
Un programa simple que lee un dato entero y lo visualiza en pantalla es:

```
#include <iostream.h>
int main()i
{
      int i;
      cin  >> i;
      cout << i;
}
```

Al igual que en el caso de cout, se pueden introducir datos en cascada:

```
#include <iostream.h>
int main()
{
      char c[60];
      int x,y;
      cin >> c >> x >> y;
      cout << c << " " << x << " " << y <<  "\n";
}
```

B.7.3. Manipuladores

Un método fácil de cambiar la anchura del flujo y otras variables de formato es utilizar un operador especial denominado *manipulador*. Un manipulador acepta una referencia de flujo como un argumento y devuelve una referencia al mismo flujo.
El siguiente programa muestra el uso de manipuladores específicamente para conversiones de número (dec, oct, y hex):

```
#include <iostream.h>
int main ()
{
      int i = 36;
      cout << dec << i << oct << i <<  " "  << hex <<  i << "\n";
}
```

La salida de este programa es: 36 44 24

Otro manipulador típico es endl, que representa al carácter de nueva línea *(salto de línea)*, es equivalente a \ n . El programa anterior se puede escribir también así:

```
#include <iostream.h>

int main ()
{
      int  i = 36;
      cout  <<    dec << i << oct << i << hex << i << endl;
}
```

B.8. SENTENCIAS

Un programa en C++ consta de una secuencia de sentencias. Existen diversos tipos de sentencias. El punto y coma se utiliza como elemento terminal de cada sentencia.

B.8.1.Sentencia de declaración

Se utilizan para establecer la existencia y, opcionalmente, los valores iniciales de objetos identificados por nombre.

nombreTipo identificador, ***…;***
nombreTipo identificador = expresión, ***…;***
const NombreTipo identificador = expresión, ***…;***

Algunas sentencias válidas en C++ son:

```
char cl;
int p, q = 5, r =a + b;          //suponiendo que a y b han sido declaradas e inicializadas antes
const double IVA = 16.0;
```

B.8.2.Sentencias expresión

Las sentencias de expresiones hacen que la expresión sea evaluada. Su formato general es:

expresión;

Ejemplo:

```
n++;
425;                        // legal, pero no hace nada
a + b;                      // legal, pero no hace nada
n = a < b || b != 0;
a += b = 3;                 // sentencia compleja
```

C++ permite asignaciones múltiples en una sentencia.

```
m = n + (p=5)   equivale a p=5;  m = n + p ;
```

B.8.3.Sentencias compuestas

Una *sentencia compuesta* es una serie de sentencias encerradas entre llaves. Las sentencias compuestas tienen el formato:

{
sentencia
sentencia
sentencia
....
}

Las sentencias encerradas pueden ser cualquiera: declaraciones, expresiones, sentencias compuestas, etc. Un ejemplo es:

```
int i = 5;

double x = 3.14, y = -4.25;
int j = 4 - i
x = 4.5 * (x - y);
```

El cuerpo de una función C++ es siempre una sentencia compuesta.

B.9. SENTENCIAS CONDICIONALES: if

if *(expresion)* if *(expresion){*
sentencia <secuencia de sentencias>
}

Si *expresión* es verdadera (distinta de cero), entonces se ejecuta *sentencia* o *secuencia de sentencias;* en caso contrario, se salta la sentencia. Después que la sentencia if se ha ejecutado, el control pasa a la siguiente sentencia:

Ejemplo 1:

```
if ( a < 0 )
   negativos++;
```

Si la variable a es negativa, se incrementa la variable negativos.

Ejemplo 2:

```
if ( numerodedias < 0 )
      numerodedias = 0;
if ( (altura - 5) < 4 )
{
      area = 3.14 * radio * radio;
      volumen = area * altura;
}
```

Ejemplo 3:

```
if (temperatura >= 45)
```

```
cout << “Estoy en Sonora - Hermosillo, en Agosto”;
cout << ”Estoy en Veracruz ” << temperatura <<endl;
```

La frase ‘Estoy en Sonora-Hermosillo, en Agosto” se visualiza cuando temperatura es mayor o igual que 45. La sentencia siguiente se ejecuta siempre.

La sentencia if_else tiene el formato siguiente:

1.if *(expresión)*
sentencia 1
else
sentencia 2

2.if *(expresión)*
<secuencia de sentencia 1>
else
<secuencia de sentencias 2>

Si *expresión* es distinto de cero, la *sentencia1* se ejecuta y *sentencia2* se salta; si *expresión* es cero, la *sentencial* se salta y *sentencia2* se ejecuta. Una vez que se ha ejecutado la sentencia if else, el control pasa a la siguiente sentencia.

Ejemplo 4:

```
if ( numero == 0 )
      cout << “No se calculara la media”;
else
      media = total / numero;
```

Ejemplo 5:

```
if (cantidad > 10)
{
      descuento = 0.2;
      precio = cantidad * precio * (1 - descuento);
}
else
{
      descuento = 0;
      precio = cantidad * precio;
}
```

B.9.1. Sentencias if_else *anidadas*

C++ permite anidar sentencias if_else para crear una sentencia de alternativa múltiple:

if *(expresionl)*
sentencial; ¡ {sentencia compuesta}
else if *(expresion2)*
sentencia2;) {sentencia compuesta}

else if *(expresion N)*
sentencia N; ¡ {sentencia compuesta}
[else
sentencia N + 1; ¡ {sentencia compuesta}]

Ejemplo:

```
if (a > 100)
      if (b <= 0)
            sumap = 1;
      else
            suman =  1;
else
      numero = 1;
```

B.9.2. Sentencias de alternativa múltiple: switch

La sentencia switch ofrece una forma de realizar decisiones de alternativas múltiples. El formato de switch es:

switch *(expresion)**{***
case *constante 1:*
sentencias
break;
case *constante 2:*
sentencias
.
.
break;
case *constante n:*
sentencias
break;
default: //opcional
sentencias
}

La sentencia switch requiere una expresión cuyo valor sea entero. Este valor puede ser una constante, una variable, una llamada a función o una expresión. El valor de *constante* ha de ser una constante. Al ejecutar la sentencia se evalúa *expresión* y si su valor coincide con una *constante* se ejecutan las sentencias a continuación de ella, en caso contrario se ejecutan las sentencias a continuación de default.

```
switch (puntos)
{
      case 10:
         nota = a;
         break;
      case 9:
         nota =5;
         break;
      case 7,8:
         nota = c;
         break;
       case 5.6:
         nota = d;
         break;
      default:
         nota = f;
}
```

B.10 .BUCLES: SENTENCIAS REPETITIVAS

Los *bucles* sirven para realizar tareas repetitivas. En C++, existen tres diferentes tipos de sentencias repetitivas:

- while
- do
- for

B.10.1. Sentencia while

La sentencia while es un bucle condicional que se repite mientras la condición es verdadera. El bucle while nunca puede iterar si la condición comprobada es inicialmente falsa. La sintaxis de la sentencia while es:

while *(expresión)*
sentencia;
o bien:
while *(expresión)* {
< *secuencia de sentencias* >
}

Ejemplo:

```
int n, suma = 0;
int i = 1;

while ( i <= 100 )
{
      cout << “Entrar”;
      cin >> n;
      suma += n;
      i++;
}
cout << “La media es” << double(suma) / 100.0;
```

B.10.2. Sentencia do

La sentencia do actúa como la sentencia while. La única diferencia real es que la evaluación y la prueba de salida del bucle se hace después que el cuerpo del bucle se ha ejecutado, en lugar de antes. El formato es:

do {
sentencias
}while (*expresión);*
sentencia siguiente

Se ejecuta sentencia y a continuación se evalúa expresión y, si es verdadero (distinto de cero), el control se pasa de nuevo al principio de la sentencia do y el proceso se repite, hasta que expresión es falso(cero) y el control pasa a la sentencia siguiente.

Ejemplo 1:

```
int n, suma=0, i = 1;
do
{
      cout << ”entrar”;
      cin >> n;
      suma += n;
      i++;
}
while ( i <= 100)
cout << “La media es” << double (suma)/100.0;
```

Ejemplo 2:

El siguiente ejemplo visualiza los cuadrados de 2 a 10:

```
int i=2;
do
  cout << i << “12 = ” << i * i++ << endl ;
while (i < 11);
```

B.10.3. La sentencia for

Una sentencia for ejecuta la iteración de un bucle un número determinado de veces. for tiene tres componentes: *expresion1*, inicializa las variables de control del bucle; *expresion2*, es la condición que determina si el bucle realiza otra iteración; la última parte del bucle for es la cláusula que incrementa o decrementa las variables de control del bucle. El formato general de for es:

for *(expresionl; expresion2; expresion3)*
sentencia; <secuencia de sentencias>;

expresionl se utiliza para inicializar la variable de control de bucle; a continuación *expresion2* se evalúa, si es verdadera (distinta de cero), se ejecuta la sentencia y se evalúa *expresion3* y el control pasa de nuevo al principio del bucle. La iteración continúa hasta que *expresion2* es falsa (cero), en cuyo momento el control pasa a la sentencia siguiente al bucle.

***Ejemplos**:*

1. `for (int i = 0; i < 5; i++) // se realizan 5 iteraciones sentenciasles`
2. Suma de 100 *números* int o, suma = 0;

```
for (int i = 0; i < 100; i++)
{
 cout <<"Entrar";
 cin >> n;
 suma+=n:
}
```

B.10.4. Sentencias break y continue

El flujo de control ordinario de un bucle se puede romper o interrumpir mediante las sentencias break y continue.
La sentencia break produce una salida inmediata del bucle flor en que se encuentra situada:

```
for (i = 0; i < 100; i++)
{
      cin » X;
      if (x < 0.0)
      {
            cout <<"salir del bucle" <<endl;
            break;
      }
      cout << sqrt(x)<< endl;
}
```

La sentencia break también se utiliza para salir de la sentencia switch.
La sentencia continue termina la iteración que se está realizando y comenzará de nuevo la siguiente iteración:

```
for (i = 0; i < 100; ++i)
{
   cin >> x;
   if (x < 0.0)
   continue;
}
```

Advertencia:

- Una sentencia break puede ocurrir únicamente en el cuerpo de una sentencia for, while, do o switch.
- Una sentencia continue sólo puede ocurrir dentro del cuerpo de una sentencia for, while o do.

B.10.5. Sentencia nula

La sentencia nula se representa por un punto y coma, y no hace ninguna acción.

```
char cad[80]="Cazorla";
int i;
for (i = 0; cad[i] != '\0'; i++);
```

B.10.6. Sentencia return

La *sentencia* ***return*** detiene la ejecución de la función actual y devuelve el control a la función llamada. Su sintaxis es:

return expresión;

donde el valor de *expresión* se devuelve como el valor de la función.

B.11. PUNTEROS (APUNTADORES)[2]

Un puntero o apuntador es una referencia indirecta a un objeto de un tipo especificado. En esencia, un puntero contiene la posición de memoria de un tipo dado. 2 El termino puntero es el más utilizado en España, mientras que en Latinoamérica se suele **utilizar** apuntador.

B.11.1.Declaración de punteros

Los punteros se declaran utilizando el operador unitario*****. En las sentencias siguientes se declaran dos variables: n es un entero, y p es un puntero a un entero.

```
int n;    //  es un tipo de dato entero
int *p;  //   p es un puntero a un entero
```

Una vez declarado un puntero, se puede fijar la dirección o posición de memoria del tipo al que apunta.

```
p = &n; //p se fija a la dirección de a
```

Un puntero se declara escribiendo:

*NombreTipo * Nombre Variable*

Una vez que se ha declarado un puntero, p el objeto al que apunta se escribe *p y se puede tratar como cualquier otra variable de tipo NombreTipo.

```
int *p, *q, o;    // dos punteros a int, y un int
*P = 101;          // *p a 101
*q= n + *p;        // *q a n mas el contenido de lo que apunta p
```

C++ trata los punteros a tipos diferentes como tipos diferentes,

```
int *ip;
double *dp;
```

Los punteros ip y dp son incompatibles, de modo que es un error escribir: `dp = ip;`
Se pueden, sin embargo, realizar asignaciones entre contenidos, ya que se realizaría una conversión explícita de tipos: *dp =ip

Existe un puntero especial *(nulo)* que se suele utilizar con frecuencia en programas C++. El puntero NULL tiene un valor cero, que lo diferencia de todas las direcciones válidas. El conocimiento nos permite comprobar si un puntero p es el puntero NULL evaluando la expresión (p==0). Los punteros NULL se utilizan sólo como señales de que ha sucedido algo. En otras palabras, si p es un puntero NULL, es incorrecto referenciar *p.

B.11.2. Punteros a arrays

A los arrays se accede a través de los índices:

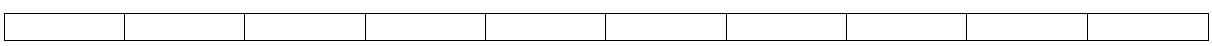
```
int lista [5];
lista [3] = 5;
```

Sin embago, también se puede acceder a través de punteros:

```
int lista [5];    //array de 5 elementos
int *ptr;         //puntero a entero
ptr = lista;      //fija puntero al primer elemento del array
ptr =+ 3;         //suma 3 a por; ptr apunta al 4to elemento
*ptr = 5;         //establece el 4to elemento a 5.
```

El nombre de un array se puede utilizar también como si fuera un puntero al primer elemento del array.

```
double a [10];
double *p = a; // p y a se refieren al mismo array
```



0 1 2 3 4 5 6 7 8 9

Este elemento se puede llamar por. a[10], p ;o bien p[0]

Este elemento se puede llamar por: a[6], *(p + 6): *o bien p[0]*

Si nombre apunta al primer elemento del array, entonces nombre + 1 apunta al segundo elemento. El contenido de lo que se almacena en esa posición se obtiene por la expresión:**(nombre + 1).*

Aunque las funciones no pueden modificar sus argumentos, si un array se utiliza como un argumento de una función, la función puede modificar el contenido del array.

B.11.3. Punteros a estructuras

Los punteros a estructuras son similares y funcionan de igual forma que los punteros a cualquier otro tipo de dato.

```
struct familia
{
      char marido[100];
      char esposa[100];
      char hijo[100];
}

familia Mackoy;                          //Mackoy estructura de tipo familia
struct familia *p;                       //p, un puntero a familia
p = &Mackoy;                             //p, contiene dirección de mackoy
strcpy (p->marido,“Luis Mackoy”);        //inicialización
strcpy (p ->esposa, “Vilma Conzalez”);   //inicialización
strcpy (p ->hijo, “Luisito Mackoy”);     //inicialización
```

B.11.4. Punteros a objetos constantes

Cuando se pasa un puntero a un objeto grande, pero se trata de que la función no modifique el objeto (por ejemplo, el caso de que sólo se desea visualizar el contenido de un array), se declara el argumento correspondiente de la función como puntero a un objeto constante.
La declaración: `const NombreTipo *v` establece v como un puntero a un objeto que no puede ser modificado. Un ejemplo puede ser: `void Visualizar(const ObjetoGrande *v);`

B.11.5. Punteros a void

El tipo de dato void representa un valor nulo. En C++, sin embargo, el tipo de puntero void se suele considerar como un puntero a cualquier tipo de dato. La idea fundamental que subyace en el puntero void en C++ es la de un tipo que se puede utilizar adecuadamente para acceder a cualquier tipo de objeto, ya que es más o menos independiente del tipo.

Un ejemplo ilustrativo de la diferencia de comportamiento en C y C++ es el siguiente segmento de programa:

```
int main()
{
   void *vptr;
   int *iptr;

   vptr = iptr;
   iptr = vptr;           //Incorrecto en C++, correcto en C
   iptr = (int *) vprr;   //Correcto en C++
}
```

B.11.6 Punteros y cadenas

Las cadenas en C++ se implementan como arrays de caracteres, como constantes de cadena y como punteros a caracteres.

Constantes de cadena
Su declaración es similar a

char *cadena = “Mi profesor”;

o bien su sentencia equivalente

char varcadena[ ] = “Mi profesor”;

Si desea evitar que la cadena se modifique, añada const a la declaración:

const char *varcadena = “Mí profesor”;

Los puntos a cadena se declaran:

`char s[1]` *o bien:* `char *s`

Punteros a cadenas
Los punteros a cadenas no son cadenas. Los punteros que localizan el primer elemento de una cadena almacenada.

```
char *varCadena;
const char *cadenafija;
```

Consideraciones prácticas
Todos los arrays en C++ se implementan mediante punteros:

```
char cadenal[l6] = “Concepto Objeto”;
char * cadena2 = cadenal;
```

Las declaraciones siguientes son equivalentes y se refieren al carácter C’:

```
cadenal[0] = ‘a’;
char car = ‘a’;
cadena1[0] = car;
```

B.11.7.Aritmética de punteros

Dado que los punteros son números (direcciones), pueden ser manipulados por los operadores aritméticos. Las operaciones que están permitidas sobre punteros son: suma, resta y comparación. Así, si las sentencias siguientes se ejecutan en secuencia:

```
char *p;    // p, contiene la dirección de un carácter
char a[l0]; // array de diez caracteres
p = &a[0];  // p, apunta al primer elemento del array
p++;        // p, apunta al segundo elemento del array
p++;        // p, apunta al tercer elemento del array
p--;        // p, apunta al segundo elemento del array
```

Un elemento de comparación de punteros, es el siguiente programa:

```
#include <iostream.h>
main (void)
{
      int *ptr1, *ptr2;
      int a[2] = {l0,l0};
      ptrl = a;
      cout << “ptr1 es ” <<  ptr1 << ” *ptr1 es “ << *ptr1 << endl;
      cout << “ptr2 es ” <<  ptr2 << ” *ptr2 es “ << *ptr2 << endl;

      //comparar dos punteros
      if (*ptrl == *ptr2)
            cout << ”ptr1 es igual a *ptr2 \n”;
      else
            cout << “*ptrl no es igual a *ptr2 \n”;
}
```

B.12. LOS OPERADORES NEW Y DELETE

C++ define un método para realizar asignación dinámica de memoria, diferente del utilizado en mediante los operadores new y delete.

El operador new sustituye a la función malloc tradicional en C, y el operador delete sustituye a la función free tradicional, también en C. new, asigna memoria, y devuelve un puntero al new *Nombre Tipo* y un ejemplo de su aplicación es:

```
int *ptrl
double *ptr2;
ptrl = new int;     // memoria asignada para el objeto ptrl
ptr2 = new double; // memoria ampliada para el objeto ptr2
*ptrl = 5;
*ptr2 = 6.55;
```

Dado que new devuelve un puntero, se puede utilizar ese puntero para inicializar el puntero de una sola definición, tal como:

```
int* p = new int;
```

Si new no puede ocupar la cantidad de memoria solicitada devuelve un valor NULL. El operador delete libera la memoria asignada mediante new.

```
delete prt1;
```

Un pequeño programa que muestra el uso combinado de new y delete es:

```
include <iostream.h>
void main (void)
{
      char *c;
      c = new char[512];
      cin >> c;
      cout << c <<endl;
      delete [] c;
}
```

Los operadores new y delete se pueden utilizar para asignar memoria a arrays, clases y otro tipo de datos.

```
int *i
i = new int[2][35];     //crear el array de 2 x 35 dimensiones
…                       //asignar el array
delete i;               //destruir el array
```

Sintaxis de new y delete

new *nombre-tipo*	new int new char[l00]
new *nombre-tipo ínicializador*	new int(99) new char('C')
new *nombre-tipo*	new (char*)
delete *expresión*	delete p
delete[] *expresión*	delete []p

B.13. ARRAYS

Un *array*[3] *(matriz, tabla)* es una colección de elementos dados del mismo tipo que se identifican por medio de un índice. Los elementos comienzan con el índice 0. [3]En Latinoamérica, este término se traduce al español por la palabra *arreglo.*

Declaración de arrays

Un declaración de un array tiene el siguiente formato:

nombreTipo nombreVariable [n]

Algunos ejemplos de arrays unidimensionales:

```
int ListaNum[2] ;            //array de dos enteros
char ListaNombres[l0];      //array de 10 caracteres
```

Arrays *multidimensionales* son:

nombretipo *nombreVariabie*[nl] [n2] ... [nx];

El siguiente ejemplo declara un array de enteros 4 x 10 x 3

```
int multidim [4] [1O] [3];
```

El ejemplo tabla declara un array de 2 x 3 elementos

```
int tabla [2][3]; //array  enteros de 2 x 3 = 6 elementos
```

B.13.1. Definición de arrays

Los arrays se inicializan con este formato:

```
int a[3]    = {5, 10, 15};
char cad[5] = {‘a’, ‘b’ , c’, ‘d’, ‘e’};
int tabla [2][3] = {{l,2,3} {3,4,5}};
```

```
Las tres siguientes definiciones son equivalentes:

char saludo [5] = “hola”;
char saludo [] = “hola”;
char saludo [5] = {‘h’, ‘o’, ‘l’, ‘a‘}
```

1. Los arrays se pueden pasar como argumentos a funciones.
2. Las funciones no pueden devolver arrays.
3. No está permitida la asignación entre arrays. Para asignar un array a otro, se debe escribir el código para realizar las asignaciones elemento a elemento.

B.14. ENUMERACIONES, ESTRUCTURAS Y UNIONES

En C++, un nombre de una enumeración, estructura o unión es un nombre de un tipo. Por consiguiente, la palabra reservada **struct**, union, o enum no son necesarias cuando se declara una variable.

El tipo de dato *enumerado* designa un grupo de constantes enteros con nombres. La palabra reservada enum se utiliza para declarar un tipo de dato enumerado o *enumeración.* La sintaxis es:

enum *nombre*
{
   lista-simbolos
}

donde *nombre* es el nombre de la variable declarada enumerada, LISTA-SIMBOLOS es una lista de tipos enumerados, a los que se asigna valores cuando se declara la variable enumerada y puede tener un valor de inicialización. Se puede utilizar el nombre de una enumeración para declarar una variable de ese tipo (variable de enumeración). Ej: *nombre* ver;
Considérese la siguiente sentencia:

```
enum color [Rojo, Azul, Verde, Amarillo];
```

Una variable de tipo enumeración color es:

```
color pantalle = Rojo; //Estilo C++
```

Una *estructura* es un tipo de dato compuesto que contiene una colección de elementos de tipos de datos diferentes combinados en una única construcción del lenguaje. Cada elemento de la colección se llama miembro y puede ser una variable de un tipo de dato diferente. Una estructura representa un nuevo tipo de dato en C++.

La sintaxis de una estructura es:

```
struct nombre
{
   miembros
};
```

Un ejemplo y una variable tipo estructura se muestran en las siguientes sentencias:

```
struct cuadro
{
   int i;
   float f;
};
 struct cuadro nombre;     //Estilo C
cuadro nombre;             //Estilo C++
```

Una *unión* es una variable que puede almacenar objetos de tipos y tamaños diferentes. Una unión puede almacenar tipos de datos diferentes, sólo puede almacenar uno cada vez, en oposición a una estructura que almacena simultáneamente una colección de tipos de datos. La sintaxis de una unión es:

```
union nombre
{
   miembros
};
```

Un ejemplo de una unión es:

```
union alfa
{
      int x;
      char o;
};
```

Una declaración de una variable estructura es:

```
alfa w;
```

El modo de acceder a los miembros de la estructura es mediante el operador punto.

```
u.x = 145;
u.c = 'z';
```

C++ admite un tipo especial de unión llamada *unión anónima,* que declara un conjunto de miembros que comparten la misma dirección de memoria. La unión anónima no tiene asignado un nombre y, en consecuencia, se accede a los elementos de la unión directamente. La sintaxis de una unión anónima es:

```
union
{
   int nuevolD;
   int contador;
}
```

Las variables de la unión anónima comparten la misma posición de memoria y espacio de datos.

```
int main()
{
      union
      {
            int x;
            float y;
            double z;
      }

      x = 25;
      y = 245.245; //el valor en y sobreescribe el valor de x
      z = 9.41415; //el valor en z sobreescribe el valor de z
}
```

B.15. CADENAS

Una *cadena* es una serie de caracteres almacenados en bytes consecutivos de memoria. Una cadena se puede almacenar en un array de caracteres (char) que tenían en un carácter nulo (cero):

```
char perro[5] = { `m`,`o`,`r`,`g`,`a`,`n`} // no es una cadena
char gato[5] = { ´f´,´e´,´l´,´i´,´n´,´o´,´\0´,}; // es una cadena
```

Lectura de una cadena del teclado

```
#include <iostream.h>
main ()
{
      char cad [80];
      cout << “introduzca una cadena:”; // lectura del teclado cm » cad;
      cin >> cad;
      cout << “Su cadena es:”;
      cout << cad;
      return 0;
}
```

Esta lectora del teclado lee una cadena hasta que se encuentra el primer carácter blanco. Así, cuando se lee “Hola que tal” la primera cadena, en cad sólo se almacena Hola. Para resolver el problema, utilizará la función gets () que lee una cadena completa leída del teclado. El programa anterior quedaría así:

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

main()
{
      char cad[80];
      cout « “Introduzca una cadena:”;
      gets (cad);
      cout « Su cadena es:”; cout << cad;
      return 0;
}
```

B.16. FUNCIONES

Una *función* es una colección de declaraciones y sentencias que realizan una tarea única. Cada función tiene cuatro componentes: (1) su nombre, (2) el tipo de valor que devuelve cuando termina su tarea, (3) la información que toma al realizar su tarea, y (4) la sentencia o sentencias que realizan su tarea. Cada programa C++ tiene al menos una función: la función main.

B.16.1. Declaración de funciones

En C++, se debe declarar una función antes de utilizarla. La declaración de la función indica al compilador el tipo de valor que devuelve la función y el número y tipo de argumentos que toma. La declaración en C++ se denomina *prototipo:*

tipo NombreFuncion (lista argumentos);

Ejemplos:

```
double Media(double x, double y);
void print(char* formato, ...);
extern max(const int*, int);
char LeerCaracter();
```

B.16.2. Definición de funciones

La definición de una función es el cuerpo de la función que se ha declarado con anterioridad.

```
double Media (double x, double y) // Devuelve la media de x e y
{
      return (x + y)/2.0
}

char LeerCaracter() //Devuelve un carácter de la entrada estándar
{
      char c;
      cin >> c;
      return c;
}
```

B.16.3. Argumentos por omisión

Los parámetros formales de una función pueden tomar valores por omisión, o argumentos cuyos valores se inicializan en la lista de argumentos formales de la función.

```
int Potencia (int n, int k = 2);
Potencia(256); //256 elevado al cuadrado
```

Los parámetros por omisión no necesitan especificarse cuando se llama a la función. Los parámetros con valores por omisión deben estar al final de la lista de parámetros:

```
void fund (int i =3, int j);                // ilegal
void func2 (int i,  int j=0, int k = 0);    // correcto
void func3 (int i = 2, int j, int k = 0);   // ilegal
void ImprimirValores (int cuenta, double cantidad= 0.0);

Llamadas a la función ImprimirValores:
      ImprimirValores ( n , a);
      ImprimirValores ( n );       //equivalente a ImprimirValores (n, 0.0)

Otras declaraciones y llamadas a funciones son:
      double fl(int n, int m, int p=0);             //legal
      double f2(int n, int m= 1, int p = 0);        //legal
      double f3(int n = 2, int m= 1, int p =0);           //legal
      double f4(int n, int m = 1, int p);                 //ilegal
      double f5(int n = 2, int m, int p = 0);       //ilegal
```

B.16.4. Funciones en línea (inline)

Si una declaración de función está precedida por la palabra reservada inline, el compilador sustituye cada llamada a la función con el código que implementa la función.

```
inline int max(int a, int b)
{
      if (a > b) return a;
      return b;
}

main ()
{
      int x = 5, y = 4;
      int z = Max(x,y);
}
```

Los parámetros con valores por omisión deben entrar al final de la lista de parámetros: Las funciones f1, f2 y f3 son válidas, mientras que las funciones f4 y f5 no son válidas.

Las funciones *en línea (inline)* evitan los tiempos suplementarios de las llamadas múltiples a funciones. Las funciones declaradas en línea deben ser simples con sólo unas pocas sentencias de programa; sólo se pueden llamar un número limitado de veces y no son recursivas.

```
inline int abs(int i);
inline int min(int v1, int v2);
int mcd(int vl, int v2);
```

Las funciones en línea deben ser definidas antes de ser llamadas.

```
#include <iostream.h>
int incrementar (int I);

inline incrementar (int i)
{
      i ++;
      return ;
}
```

```
main (void)
{
      int i = 0;
      while (i < 5)
      {
            i = incrementar (i )
            cout « “i es:”« i «endl;
      }
}
```

B.16.5. Sobrecarga de funciones

En C++, dos o más funciones distintas pueden tener el mismo nombre. Esta propiedad se denomina *sobrecarga*. Un ejemplo es el siguiente:

```
int max (int, int t);
double max (double, double);
```

o bien este otro:

```
void sumar (char i);
void sumar (float j);
```

Las funciones sobrecargadas se diferencian en el número y tipo de argumentos, o en el tipo que devuelven las funciones, y sus cuerpos son diferentes en cada una de ellas.

```
#include <iostream.h>
void suma (char);
void suma (float);

main (void)
{
      int i = 65;
      float j = 6.5;
      char c = ‘a’;


      suma ( i );
      suma ( j ) ;
      suma ( c ) ;
}

void suma (char i)
{
      cout << “Suma interior(char) “ <<endl;
}
void suma ( float j )
{
      cout << “Suma interior (float) “<< endl;
}
```

B.16.6. El modificador const

El modificador de tipos *const* se utiliza en C++ para proporcionar protección de sólo lectura para variables y parámetros de funciones. Cuando se hace preceder un tipo de argumento con el modificador *const* para indicar que este argumento no se puede cambiar, el argumento al que se aplica no se puede asignar un valor ni cambiar.

```
void copia (const char * fuente, char* dest);
void funcdemo (const int I);
```

B.16.7. Paso de parámetros a funciones

En C++ existen tres formas de pasar parámetros a funciones:

1. *Por valor* La función llamada recibe una copia del parámetro y este parámetro no se puede modificar dentro de la función

```
void intercambio (int x, int y)
{
    int aux = y;
    y = x ;
    x = aux;
}
//.....
intercambio (i , j ); // las variables i, j, no se intercambian
```

2. *Por dirección.* Se pasa un puntero al parámetro. Este método permite simular en C/C++ la llamada por referencia, utilizando tipos punteros en los parámetros formales en la declaración de prototipos. Este método permite modificar los argumentos de una función.

```
void intercambio (int*x, int*y)
{
    int aux = * y;
    *y = *x
    *x = aux;
}
// ...
intercambio (&i, &j);  // i , j se intercambian sus valores
```

3. *Por referencia.* Se pueden pasar tipos referencia como argumentos de funciones, lo que permite modificar los argumentos de una función.

```
void intercambio (int& x, int& y)
{
    int aux = y;
    y = x;
    x = aux;
}
//.....
intercambio (i,  j); //  i , j intercambian sus valores
```

Si se necesita modificar un argumento de una función en su interior, el argumento debe ser un tipo de referencia en la declaración de la función.

B.16.8. Paso de arrays

Los arrays se pasan por referencia. La dirección del primer elemento del array se pasa a la función; los elementos individuales se pasan por valor. Los arrays se pueden pasar indirectamente por su valor si el array se define como un miembro de una estructura.

Ejemplo 1:
Paso del array completo.

```
#include <iostream.h>
void func1 (int x[] ); // prototipo de función
void main( )
{
      int a[3] = {l, 2, 3};
      func1 (a) ; // sentencias
      func1 (&a[0]) ;  // equivalentes
}

void func(int x[])
{
      int i;
      for (i = 0; 1 < 3; i + 1)
            cout << 1 << x[i]<< ´\n’;
}
```

Ejemplo 2:
Pasa de a un elemento de un array.

```
#include <iostream.h>

const int n + 3;
void func2(int x);
void main( )
{
      int a[n] = { l, 2, 3};
      func2 (a[2] );
}

void func2(int x)
{
      cout << x << ‘\n’;
}
```