APÉNDICE B

GUÍA DE SINTAXIS *ANSI/ISO* ESTÁNDAR C++

CONTENIDO

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 orientadas a objetos. C ANSI[1] ya adoptó numerosas características de C++, por lo que la emigración 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 Reference Manual (ARM),* de Stroustrup & Ellis, así como las últimas propuestas incorporadas al nuevo borrador de C++ ANSI, que se incluyen en las versiones 3.0 (actual) y 4.0 (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:

```
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
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
0 1 2 3 4 5 6 7 8 9
+ - * / = ( ) { } [ ]  < > ´ " ! @ ≠ $ ^ & % - : . , ; ? \ |
```

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*/, se puede extender
/* hasta que aparece la marca de cierre */
// Este tipo de comentario termina al final de la línea
// Sólo es posible una línea de comentario
```

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, etc.) deben comenzar con una letra del alfabeto (mayúscula o minúscula) o con un carácter subra-

[1] Se utiliza indistintamente los términos ANSI C (nombre inglés) y C ANSI, traducción al español muy usada en la vida profesionbal y académica.

yado 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.

```
test_prueba      //legal
X123             //legal
multi_palabra    //legal
var25            //legal
15var            //no legal
```

C++ es sensible a las mayúsculas. Las letras mayúsculas y minúsculas se consideran diferentes.

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

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

```
nombre   apellidos   salario     precio_neto
Edad     Longitud    Altura      Salario_Mes
```

B.1.4. Palabras reservadas

Las palabras reservadas o claves no se pueden utilizar como identificadores, debido a su significado estricto en C++; tampoco se pueden redefinir. La Tabla B.1 enumera las palabras reservadas de C++ según el ARM[2].

TABLA B.1. Palabras reservadas (*Keywords*) de ANSI/ISO C++

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

* Estas palabras no existen en C ANSI.

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

[2] Siglas del libro de Margaret Ellis y Bjarne Stroustrup en el que se definen las reglas de sintaxis del lenguaje C++ estándar, *Annotated Reference Manual,* Addlson-Wesley, 1992.

TABLA B.2. Palabras reservadas de Borland C++ 5

__asm	__cdecl	__cs	__declspec	__ds
__es	__except	__export	__far	__fastcall
__finally	__huge	__import	__interrupt	__loadds
__near	__pascal	__rtti	__saveregs	__seg
__ss	__stdcall	__thread	__try	_asm
_cdecl	_cs	_ds	_es	_export
_far	_fastcall	_huge	_import	_interrupt
_loadds	_near	_pascal	_saveregs	_seg
_ss	_stdcall	asm	auto	bool
break	case	catch	cdecl	char
class	const	const_cast	continue	default
delete	do	double	dynamic_cast	else
enum	explicit	extern	false	far
float	for	friend	goto	huge
if	inline	int	interrupt	long
mutable	namespace	near	new	operator
pascal	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof
static	static_cast	struct	switch	template
this	throw	true	try	typedef
typeid	typename	union	unsigned	using
virtual	void	volatile	wchar_t	while

TABLA B.3. Palabras reservadas de Visual C++ 6

_asm	_fastcall	public	signed
_except	new	unsigned	void
_virtual_inheritance	typedef	using directive	do
throw	class	default	_leave
case	goto	_int8	struct
_finally	register	sizeof	auto
operator	using declaration	volatile	explicit
typeid	uuid	double	mutable
const	delete	long	true
if	_int16	switch	catch
reinterpret_cast	static	template	flota
return	wmain	_based	private
_uuidof	dynamic-cast	extern	typename
dllexport	man	naked	const-cast
_int32	_multiple _inheritance	try	inline
static-cast	this	_cdecl	_inline
while	bool	for	short
else	false	protected	virtual
enum	namespace	union	dllimport
-single-inheritance	_try	continue	_int64
thread	char	_dec1spec	_stdcall
break	friend	int	xalloc

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

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

bool	false	reinterpret_cast	typeid
cons_cast	mutable	static_cast	using
dynamic_cast	namespace	true	wchar_t

B.2. Tipos de datos

Los tipos de datos 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
unsigned short	unsigned	unsigned long
float	double	long double

Los tipos derivados en C++ pueden ser:

- enumeraciones (enum)
- estructuras (struc)
- 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.

- *Usar funciones declaradas*. Esta acción es ilegal en C++ y está permitida en C:

```
int main ()
{
  // ...
  printf (x);     //C: int printf ();
                  //C++ es ilegal, ya que printf no está
                  //permitida devuelve un int
  return 0;
}
```

- *Fallo al devolver un valor de una función*. Una función en C + + declarada con un tipo determinado de retorno 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 conversión 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[8] = "Cazorla";
```

B.3. CONSTANTES

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

```
456   0456   Ox456       //constantes enteras: decimal, octal,
                         //hexadecimal
1231  123ul              //constantes enteras: long, unsigned
                         //long
'B' 'b' '4'              //constantes tipo char

3.1415f     3.14159L     //constantes 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'
```

Una constante literal es un valor escrito directamente en el programa siempre que se necesite. Por ejemplo,

```
int miEdad = 25;
```

miEdad es una variable de tipo int; 25 es una constante literal.

B.3.1. Declaración de constantes con `const`

En C++, los identificadores de variables/constantes se pueden declarar cons*tantes,* significando que su valor se inicializa pero no se puede modificar. Estas constantes se denominan *simbólicas*. Esta declaración se realiza con la palabra reservada `const`.

```
const double PI = 3.1416;
const char BLANCO =  '  ';
const double PI_EG = PI;
const double DOBLE_PI  = 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 parados por referencia sean modificados:

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

B.3.2. Declaración de constantes con `define#`

C++ soporta también el método tradicional de declaración de constantes, aunque ahora está obsoleto. Para declarar una constante por este método se debe realizar con la palabra reservada `#define`.

```
#define estudiantesPorClave 50
#define PI 3.1416
#define hex 16
```

B.4. CONVERSIÓN DE TIPOS

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

```
(tipo) expresión
```

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

```
nombre del tipo (expresión)
```

Las notaciones siguientes son equivalentes:

```
z = float(x);       //notación de moldes en C++
z = (float)x;       //notación de moldes en C
```

B.5. DECLARACIÓN DE VARIABLES

En C ANSI, 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 NombreVariable1, NombreVarible2, ...
```

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, anyo = 1995;
double 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 <iostream.h>
int main()
{
  int i;
  for (i=0; i < 100; ++i)
  cout << i << endl;

  double j;
  for (j = 1.7547; j < 25.4675; j+= .001)
    cout << i << endl;
}
```

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

```
int main()
{
  for (int i=0; i<100; ++i)
    cout << i << endl;

  for (double j - 1.7545; j<25.4675;j += .001)
    cout << i << endl;
}
```

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	`z = x * y;`
/	División	Divide x por y	`z = x/y;`
%	Módulo	Resto de x dividido por y	`z = x%y;`
++	Incremento	Incrementa x después de usar	`x++`
--	Decremento	Decrementa x antes de usar	`--x`

Ejemplos

```
-i + w;        //menos unitario más unitario
a*b/c%d        //multiplicación, división, módulo
a+b  a-b       //suma y resta binaria
a=5/2;         //a toma el valor 2, si se considera a entero
a=5/2;         //a toma el valor 2.5, si a es real
```

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

```
variable++          //postincremento
++variable          //preincremento
variable--          //postdecremento
-- variable         //predecremento
++a;       equivale a a =    a +1;
--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+l;
```

```
int i, j, k = 5;
k++;              //k vale 6, igual efecto que ++k
--k;              //k vale ahora 5, igual efecto que k--
k = 5;
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 adjudica a la 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;          //asigna (b+5) a variable z
```

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

```
a = b+(c=10);
```

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;
double x = y;                  PI = 3;
a=b=6;                         x++ = y;
```

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

TABLA B.7. Operadores aritméticos de asignación

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;

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 construcciones de toma de decisión en un lenguaje de programación. La Tabla B.8 muestra los operadores lógicos y relacionales.

TABLA B.8. Operadores lógicos y relacionales

Operador	Nombre	Ejemplo
&&	AND (*y*) lógico	a && b
||	OR (*o*) lógico	C || d
!	NOT (*no*) lógico	!C
<	Menor que	i < 0
<=	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)? 1= i;

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

```
if (condición)
  variable = expresión1;
else
  variable = expresión2;
```

es equivalente a

```
variable =(condición) ? expresión1 : expresión2;
```

La expresión condicional comprueba la condición. Si esa condición es verdadera, se asigna *expresión1* a *variable;* en caso contrario se asigna *expresión2* a *variable*.

Reglas prácticas

Los operadores lógicos y relacionales actúan sobre valores lógicos: el valor *falso* 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	1, si x excede a y,	si no 0
x >= y	1, Si x es mayor que o igual a y,	si no 0
x < y	1, si x es menor que y,	si no 0
x <= y	1, si x es menor que o igual a y,	si no 0
x == y	1, si x es igual a y,	si no 0
x! = y	1, si x e y son distintos,	si no 0
!x	1, si x es 0,	si no 0
x || y	0, si ambos x e y son 0,	si no 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 toma 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, *expr2* no se evaluará nunca.

2. *Operación lógica OR* (||). 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. Operadores de manipulación de bits

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

TABLA B.9. Operadores de manipulación de bits *(bitwise)*

Operador	Significado	Ejemplo
&	AND bit a bit	`x & 128`
|	OR bit a bit	`j | 64`
^	XOR bit a bit	`j ^ 12`
~	NOT bit a bit	`~j`
<<	Desplazar a izquierda	`i << 3`
>>	Desplazar a derecha	`j >> 4`

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

Operador	Formato largo	Formato reducido
&=	`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

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

Ejemplos

```
int m, n[12];
sizeof(m)        //proporciona 4, en máquinas de 32 bits
sizeof(n)        //proporciona 48
sizeof(15)       //proporciona 4
tamanyo = sizeof(long) - sizeof(int);
```

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*.

La Tabla B.11 muestra la precedencia y asociatividad de operadores.

TABLA B.11. Precedencia y asociatividad de operadores

Operador	Asociatividad	Prioridad
`::.->[]()++--`	Izquierda-Derecha	1
`++ -- & (dirección) tipo) ! - ++ sizeof (tipo) new delete* (indirección)`	Derecha-Izquierda	2
`.* ->*`	Izquierda-Derecha	3
`* / %`	Izquierda-Derecha	4
`+ -`	Izquierda-Derecha	5
`<< >>`	Izquierda-Derecha	6
`< <= > >=`	Izquierda-Derecha	7
`== !=`	Izquierda-Derecha	8
`&`	Izquierda- Derecha	9
`^`	Izquierda-Derecha	10
|	Izquierda-Derecha	11
`&&`	Izquierda-Derecha	12
||	Izquierda-Derecha	13
`?:`	Derecha-Izquierda	14
`= += -= *= /= %= >>= <<= &= /= ^=`	Derecha-Izquierda	15
(operador coma)	Izquierda-Derecha	16

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.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. Estas operaciones se realizan mediante rutinas de bibliotecas. Las clases que C + + utiliza para entrada y salida se conocen como *flujos*. Un *flujo* 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`.

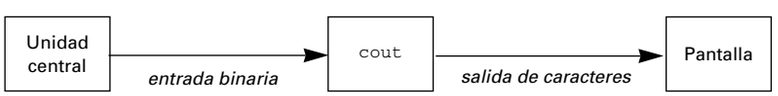


FIGURA B.1. Uso de flujos para salida de caracteres.

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 <iostream.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 = 495.125;
  char *c = "y 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()
{
  int i;
  cin >> i;
  cout << i << "\n";
}
```

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 <<        z<< 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),* y 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. Sentencias 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 c1;

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 de expresión

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

expresión;

Ejemplos

```
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*

El formato general de una sentencia `if` es:

```
if (expresión)                if (expresión) {
   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)              2. if (expresión)"{
     sentencia1;                   < secuencia de sentencias 1 >
   else                            else
     sentencia2;                   < secuencia de sentencias 2 >
```

Si *expresión* es distinto de cero, la *sentencia1* se ejecuta y *sentencia2* se salta; si *expresión* es cero, la *sentencia1* 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 calculará la media";
else
  media = total / Numero;
```

Ejemplo 5

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

```
else {
  descuento = 0;
  precio = n * Precio;
}
```

B.9.1. Sentencias *if _else* anidadas

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

```
if (expresión 1)
   sentencia 1; | {sentencía compuesta}
else if (expresión 2)
   sentencia 2; | {sentencia compuesta}
else if (expresion N)
   sentencia N; | {sentencia compuesta}
[else
   sentencia N+1; | {sentencía 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 = 'B';
    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
  sentencia
while (expresion);
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

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

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

El siguiente ejemplo visualiza los cuadrados de 2 a 10:

```
int i = 2;
do
{
  cout << i << "por" << i <<" =" << i * i++ << endl;
}while (i < 11);
```

B.10.3. Sentencia *for*

Una sentencia `for` ejecuta la iteración de un bucle un número determinado de veces. `for` tiene tres componentes: *expresión1* inicializa las variables de control del bucle; *expresión2* 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 (expresión1; expresión2; expresión3)
     sentencia;|{<secuencia de sentencias>};
```

expresión1 se utiliza para inicializar la variable de control de bucle; a continuación, *expresión2* se evalúa; si es verdadera (distinta de cero), se ejecuta la sentencia y se evalúa *expresión3*, y el control pasa de nuevo al principio del bucle. La iteración continúa hasta que *expresión2* es falsa (cero), en cuyo momento el control pasa a la sentencia siguiente al bucle.

Ejemplos

```
1. for (int i = 0; i < n; i++) //se realizan n iteraciones
       sentencias

2. Suma de 100 números

   int n, 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 `for` 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 expresion;
```

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

B.11. PUNTEROS (APUNTADORES)[3]

Un puntero o apuntador es una referencia indirecta a un objeto de un tipo específicado. En esencia, un puntero contiene la posición de memoria de un tipo dado.

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;     //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 n
```

Un puntero se declara escribiendo:

*NombreTipo *NombreVariable*

[3] El término *puntero* es el más utilizado en España, mientras que en Latinoamérica se suele utilizar *apuntador.*

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, n;     //dos punteros a int, y un int
n = -25;            //n se fija a -25
*p = 105;           //*p a 105
*q = n +*p;         //*q a 80
```

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;   //Error, no se pueden asignar punteros a tipos
           //diferentes
```

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 correcto referenciar `*p`.

B.11.2. Punteros a arrays

A los arrays se accede a través de los índices:

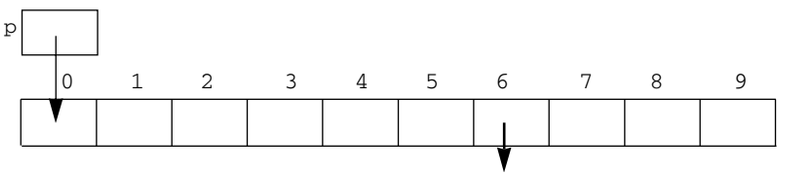
```
int lista[5];
lista[3] = 5;
```

A los arrays 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 += 2;      //suma 3 a ptr; ptr apunta al 4° elemento
*ptr = 5;      //establece el 4° 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
```



Este elemento se puede llamar por: `a[0], *p` *o bien* `p[0]`

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

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;
  char *esposa;
  char *hijo;
};

familia mackoy;        //mackoy estructura de tipo familia
familia *p;            //p, un puntero a familia
p = &mackoy;           //p, contiene dirección de mackoy
```

```
p -> marido  = "Luis MackoY";      //iniciación
p -> esposa  = "Vilma González";   //iniciación
p -> hijo    = "Luisito Mackoy";   //iniciación
```

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, en caso de que sólo se desee visualizar el contenido de un array), se declara el argumento correspondiente de la función como un puntero a un objeto constante. La declaración es,

```
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 *) vptr; //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 = "Mi profesor";
```

> Los punteros a cadena se declaran:
>
> `char s[]` *o bien* `char *s`

Punteros a cadenas

Los punteros de 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 cadena1[16]= "Concepto Objeto";
char *cadena2    = cadena1;
```

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

```
cadena1[0]   cadena1   cadena2
```

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[10];      // 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 ejemplo de comparación de punteros es el siguiente programa:

```
#include <iostream.h>
main (void)
{
  int *ptr1, *ptr2;
  int a[2] = {10,10};
  ptr1 =a;
  cout << "ptr1 es" << "ptr1 << "*ptr1 es" << *ptr1 << endl;
  ptr2 = ptr1 + 1;
  cout << "ptr2 es" << ptr2 << "*Ptr2 es" << *ptr2 << endl;

//comparar dos punteros
if (ptr1 == ptr2)
    cout << "ptr1 no es igual a ptr2 \n";

if (*ptr1 == *ptr2)
    cout << ptr1 es igual a *ptr2 \n";
else
    cout << ptr1 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 C, 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 objeto últimamente creado. Su sintaxis es:

```
new NombreTipo
```

y un ejemplo de su aplicación es:

```
int *ptr1;
double *ptr2;
ptr1 = new int;    //memoria asignada para el objeto ptr1
ptr2 = new double; //memoria ampliada para el objeto ptr2
*ptr1= 5;
*ptr2= 6.55;
```

Dado que `new` devuelve un puntero, se puede utilizar ese puntero para inicializar el puntero en 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 ptr1;
```

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
                           //asignar el array
...
delete i;                  //destruir el array
```

Sintaxis de `new` y `delete`:

`new` *nombre_tipo*	`new int`	`new char[100]`
`new` *nombre_tipo inicializador*	`new int(99)`	`new char('C')`
`new` *nombre_tipo*	`new (char*)`	
`delete` *expresión*	`delete p`	
`delete[]` *expresión*	`delete[]`	

B. 13. ARRAYS

Un **array**[4] (*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.

Declaración de arrays

Una 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[10] ;  //array de 10 caracteres
```

Arrays multidimensionales son:

```
nombretipo      nombreVariable[nl] [n2] ... [nx];
```

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

```
int multidim[4][10][3];
```

El ejemplo `tabla` declara un array de 2 × 3 elementos.

```
int tabla[2][3];   //array de enteros de 2x3 = 6 elementos
```

B.13.1. Definición de arrays

Los arrays se inicializan con este formato:

```
int a[31            = {5, 10, 15};
char cad[5]         = {a', 'b', 'c', 'd', 'e'};
int tabla[2][3]     = {{1,2,3}{3,4,5}};
```

Las tres siguientes definiciones son equivalentes:

```
char saludo[5]= "hola";
char saludo[] = "hola";
char saludo[5] = {'h', 'o', 'l', 'a', '\o'};
```

[4] En Latinoamérica, este término se traduce al español por la palabra **arreglo**.

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-símbolos* 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).

```
nombre var;
```

Considérese la siguiente sentencia:

```
enum color {Rojo, Azul, Verde, Amarillo};
```

Una variable de tipo enumeración color es:

```
color pantalla = 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 muestra 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 una 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 estructura es:

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

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 nuevoID;
  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 termina en un carácter nulo (cero, `'\0'`).

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

Lectura de una cadena del teclado

```
#include <iostream.h>
main()
{
  char cad[80];

  cout <<"Introduzca una cadena:";  //lectura del teclado
  cin >> cad;
  cout <<"Su cadena es:";
  cout << cad;

  return 0;
}
```

Esta lectura del teclado lee una cadena hasta que se encuentra el primer carácter blanco. Así, cuando se lee "Sierra Mágina. Jaén", en cad sólo se almacena Sierra. Para resolver el problema se utiliza la función gets (), que lee una cadena completa leída del teclado. El programa anterior se escribe así para leer toda la cadena introducida:

```
#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 5) 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:*

```
tipoNombreFuncion (lista argumentos);
```

Ejemplos válidos son:

```
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  func1  (int i=3, int j);              //ilegal
void  func2  (int i, int j=0, ínt 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 son:

```
ImprimirValores (n,a);
ImprimirValores (n); //equivalente a ImprimirValores (n, 0.0)
```

Otras declaraciones y llamadas a funciones son:

```
double f1(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 v1, 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);
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;

  int i = 6.5;

  suma(i);
  suma(j);
}

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, al argumento al que se aplica no se puede asignar un valor ni cambiar.

```
void copig (const char * fuente, char* dest);
void func_demo (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 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.

```
//Paso del array completo. Ejemplo 1
#include <iostream.h>
void func](int x[]);        //prototipo de función
void  main(){
  int a[3] = {1,2,3};
  func1(a);                 //sentencias
  func1(&a[0]);             //equivalentes
}

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

El siguiente ejemplo pasa un elemento de un array:

```
#include <iostream.h>

{
const int N=3;
void func2(int x);
void main() {
  int a[N] = {1,2,3};
  func2(a[2]);
}

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

B.17. CLASES

Una clase es un tipo definido por el usuario que contiene un *estado* (datos) y un *comportamiento* (funciones que manejan los datos). Una clase es como una estructura `(struc)` en C con la diferencia de que contiene funciones incorporadas. Además, una clase puede tener algunos miembros que sean *privados* y a los que no se puede acceder desde el exterior de la clase. Esta propiedad se llama *encapsulamiento* y es un medio útil para ocultar detalles que el resto del programa no ve.

Las clases son distintas de los objetos que definen la clase como un tipo. Las clases en C++ son comparables a los tipos primitivos tales como `int, char` y `double` y un nombre de clase puede aparecer en cualquier texto que puede hacerlo `int, char` y `double.` Un objeto, en contraste, es como un valor de un entero individual, un carácter o de coma flotante. Un objeto tiene un estado particular, mientras que una clase es una descripción general del código y los datos y no contiene información. Por cada clase puede haber muchos objetos. Un objeto se conoce normalmente como una *instancia* o un *ejemplar.*

Sintaxis

Se puede declarar una clase en C++ utilizando `class, struct` o `union:`

```
class | struct | union nombre[declaraciones_clase_base]
{
 declaraciones
} [definiciones_de_objeto];
```

|, *indica que una de las tres palabras reservadas se debe utilizar al principio de la declaración*. [ ], *el contenido de su interior es opcional*.

Cada una de las tres palabras reservadas, `class, struct` y `union,` crea una clase con estas diferencias:

- El nivel de acceso a miembros por omisión es *privado* si se utiliza **`class.`** El nivel de acceso a miembros es *público* si se utiliza **`union`** o **`struct.`**
- Las palabras reservadas **`struct`** y **`class`** crean un tipo similar a una estructura en C. Una **`union`** crea un tipo en la que todos los miembros datos comienzan en la misma dirección en memoria.

```
class CCad{
private:
  char *pDatos;
  int nLongitud;
public:
  CCad();      //Constructor
  -CCad();     //Destructor
  char *obtener(void) {return pDatos;}
  int obtenerlongitud(void) {return nLongitud;}
  char * copy(char * s);
  char * cat(char * s);
}
```

B.17.1. Constructor

Un constructor es una función miembro que se llama automáticamente cuando se crea un objeto; su nombre es el mismo que el nombre de la clase. Cuando se crean objetos dinámicamente, se utilizan `new` y `delete` en vez de `malloc` y `free`. Los constructores no tienen tipo de retorno (ni incluso **`void`**). La propia clase es el tipo de retorno implícito. Los constructores son como otras funciones miembro, aunque no se heredan. Es conveniente que los constructores sean declarados como públicos para que el resto del programa tenga acceso a ellos.

```
class CDibujo {
private:
  long coste;
  int nEstrellas;
  CCad sDirector;
public:
  //Constructores
    CDibujo();
    CDibujo(long c, int n, CCad dir);
    ~CDibujo() {delete[ ];} / / destructor
};

//definicion de los constructores
CDibujo::CDibujo() {
}

CDibujo::CDibujo(long c, int n, CCad dir){
  coste = c;
  nEstrellas = n;
  sDirector  = dir;
}
```

B.17.2. Constructor por omisión

El constructor por omisión en cada clase es el constructor que no tiene argumentos. C++ lo invoca automáticamente en las siguientes situaciones: cuan-

do se define un objeto de la clase sin argumentos, cuando un array de objetos se declara con miembros no inicializados, o cuando el operador **new** se utiliza, pero no se especifica ningún argumento. La sintaxis de un constructor por omisión es:

```
class()
```

```
vector::vector () { ... }
vector::vector(int i = 0) { ... }
```

```
class Punto {
public:
  Punto()
  }
    x = 0;
    y = 0;
  }
private:
  int x;
  int y;
};
```

C++ crea automáticamente un constructor por omision cuando no existe otro constructor.

B.17.3. Constructor de copia

Este constructor se crea automáticamente por el compilador. El constructor de copia se llama automáticamente cuando se pasa un objeto por valor; se construye una copia local del objeto. El formato es:

```
    tipo: : tipo (const tipo& x)
```

```
Punto::Punto (const Punto &p2)
{
  cerr << "Se llama al constructor de copia.\n";
  x = p2.x;
  y = p2.y;
  return *this;
}
```

el parámetro `p2` no se puede modificar por la función.

B.17.4. Arrays de objetos de clases

Un array de objetos de clases es útil cuando se requieren instancias múltiples de la misma clase. Así por ejemplo, si se define un array de objetos `Punto` llamado `figura`, el constructor por omisión `Punto` se llama para cada miembro del array

```
Punto figura[3];
```

B.17.5. Destructores

Un *destructor* es una función miembro especial que se llama automáticamente cuando se desea borrar un objeto de la clase. El nombre de un destructor es el nombre de la clase, precedido por el carácter ~ (tilde). Si no se declara explícitamente un destructor C++ crea automáticamente uno vacío.

```
class Punto{
public:
  ~Punto()
  {
    cout << "Destructor Punto llamado \n";
  }
  //...
};
```

Un destructor no tiene parámetros, ni incluso `void` y no tiene tipo de retorno. Una clase puede tener sólo un constructor.

B.17.6. Clases compuestas

Cuando una clase contiene miembros dato que son por sí mismos objetos de otras clases, se denomina *clase compuesta*.

```
class DatosAcademicos { //...};
```

```
class Direccion { // ...};
```

```
class Estudiante{
public:
  Estudiante()
  }
    LeerId(0);
    LeerNotaMedia(0.0);
  }
  void LeerId(long);
  void LeerNotaMedia(float);
private:
  long id;
  DatosAcademicos da;
  Direccion dir;
  float NotaMedia;
};
```

B.17.7. Funciones miembro

Las funciones miembro son funciones declaradas dentro de una clase. En consecuencia, ellas tienen acceso a miembros `public, private` y `protected` de esa clase. Si se definen dentro de la clase, se tratan como funciones `inline` y se tratan también como funciones sobrecargadas.

```
class vector {
public:
  vector(int n = 50);           //constructor por defecto
  vector (const vector& v)      //constructor de copia
  vector(const int a[ ], int n);

  ...
int teo() const { return(size-1); }  //función miembro en línea
private:
  ...
};
```

Las funciones miembro se invocan normalmente mediante el uso de operadores punto `(.)` o bien `->`.

```
vector a (50) , b;
vector* ptr_v = &b;
int teo15 = a.teo();
teo15 = ptr_v -> teo()
```

B.17.8. Funciones miembro constante

Una *función miembro constante* es aquella que garantiza no se modifica el estado del objeto de la clase útil.

```
class Punto {
public:
  Punto (int xval, int yval);

  int LeerX() const;

  void Fijarx (int xval);
  //...
};
```

La palabra reservada `const` debe aparecer también en la implementación de la función

```
int Punto::LeerX() const
{
return x;
}
```

B.17.9. Funciones clases amigas `(friend)`

Una amiga (`friend`) de una clase tiene acceso a todos los miembros de esa clase. Si una función `F`, amiga de una clase `C`, utiliza un objeto de la clase `C`, es como si todos los miembros de `C` fueran declarados públicos. El tipo más común de amigo de una clase es una función. Las clases también pueden ser amigas de otras clases.

Para declarar una función como amiga de una clase, se ha de incluir un prototipo de la función interior a la declaración de la clase, precedida de la palabra reservada **`friend.`**

```
class nombre {
  ...
   friend prototipo_de_funcion;
  ...
};
```

para declarar una clase como amiga, se utiliza la siguiente sintaxis:

```
class nombre {
  ...
  friend class nombre_clase;
};
```

Función amiga

```
class Estudiante;

class Empleado {
public:
  Empleado (long idVal);
  friend void RegistrarEstudiante (Estudiante &S,
              Empleado &E, float tasa);
private:
  long id;      //número de matrícula
  float PagoTasa;
};
```

Clase amiga

```
class clase_1 {
  friend class clase_2;
  //...
};

class clase_2 {
  friend class clase_3;
};
```

B.17.10. El puntero `this`

Dentro de una función miembro, la palabra reservada `this` es el nombre de un puntero implícito al objeto actual. Cuando se desea utilizar el puntero oculto en el código de la función miembro, se utiliza la palabra reservada **`this`**. La mayoría de las veces, el uso de `this` es innecesario, ya que C++ supone el uso de `this` siempre que se refiera a miembros dato.

```
Cpunto::fijar(double nuevox, double nuevoy) {

  x = nuevox;          //x significa lo mismo que this->  x
  y = nuevoy;          //y significa lo mismo que this->  y
}
```

B.18. HERENCIA

Una **herencia** es una relación *es-un* entre dos clases, en la que una nueva clase se deriva de otra clase —denominada *clase base*—. La nueva clase, se denomina *clase derivada*. La relación **es-un** se manifiesta como «un estudiante de doctorado *es-un* estudiante que está escribiendo una tesis».

Sintaxis

```
class nombre_clase_derivada: (public|protected|private)
                             clase_base {
  declaraciones
};
```

clase_base es el nombre de la clase del que se deriva la clase actual —*derivada*— los especificadores de acceso pueden ser `public, protected o private`. Un miembro `public` es accesible a través de su ámbito; los miembros de *clase_base* se heredan sin modificación en su estado. Un miembro privado `(private)` es accesible a otras funciones miembro dentro de su propia clase. Un miembro protegido `(protected)` es accesible a otras funciones miembro dentro de su clase y a cualquier clase derivada de ella. Los modificadores de acceso se pueden utilizar dentro de una declaración de una clase en cualquier orden y con cualquier frecuencia.

```
    class D : public A {
    ...
};
```

La *herencia múltiple* permite que una clase sea derivada de más de una clase base. En el siguiente ejemplo, `D` es la clase derivada, y `B1`, `B2` y `B3` son clases base.

```
class D : public B1, public B2, private B3 {
    ...
  };
```

La palabra reservada `virtual` es un especificador de función que proporciona un mecanismo para seleccionar en tiempo de ejecución la función miembro apropiada a partir de la clase base y clases derivadas

```
class D : public virtual B1, public virtual B2 {
  ...
};

class Estudiante : public virtual Persona {
//...
class Empleado : public virtual Persona {
//...
```

B.19. SOBRECARGA DE OPERADORES

La *sobrecarga de operadores* se refiere a la técnica que permite proveer un nuevo significado a los operadores estándar tales como `=, < <, ...` Dicho de otro modo, se pueden definir funciones operador para sus clases.

Una *función operador* es una función cuyo nombre consta de la palabra reservada `operator` seguida por un operador binarlo o unitario con el formato:

```
  operator operador

// Sobrecarga de operadores aritméticos y
// de asignación

class Punto
}
  //...
  public:
  //...
   Punto operator * (const Punto& p);
   Punto operator / (const Punto& p);
   Punto operator += (const Punto& p);
  //...
};
//implementación de función miembro sobrecargada
//p1 * p2
inline Punto Punto::operator * (const Punto& p)
{
  return Punto (x * p.x, y * p.y, z * p.z);
}
//p1 / p2
...
```

Una vez declarada y definida la clase se pueden escribir expresiones tales como:

```
Punto p, q, r;
//...
r = p * q;          //multiplicación
//...
r = p + = q;        //asignación encadenada y aritmética
//...
```

B.19.1. Funciones operador unitario

@operando @, *operador*

El compilador evalúa la expresión llamando a una de las siguientes funciones, dependiendo de la que está definida:

```
tipo_retorno tipo::operator@()
tipo_retorno operator@ (tipo)
```

Dependiendo del propio, la expresión que utiliza un operador unitario puede ser de la forma

operator@

B.19.2. Funciones operador binario

Se puede escribir una función operador binario para un operador (tal como `+, -, *, /`) que C++ acepta como un operador binario

operando1 @ *operando2* @ *operador*

el compilador evalúa la expresión llamando a una de las siguientes funciones, dependiendo de en cuál esté definida

```
tipo_retorno tipo1::opera tor@ (tipo2)
tipo_retorno tipo2::operator@ (tipo1, tipo2)
```

tipo1 y *tipo2* son los tipos de `operando1` y `operando2`, respectivamente.

B.19.3. Funciones operador de asignación

La declaración de esta función es:

```
class & clase::operator = (const clase &)
```

Un ejemplo que ilustra una función operador de asignación

```
class Cpunto  {
private:
  double x, y;
public:
Cpunto& operator = (const Cpunto& punto);
  ...
};
```

B.19.4. Operadores de incremento y decremento

Los operadores incremento (+ +) y decremento (– –) siguen la mayoría de las reglas de los operadores unitarios, pero también pueden ser prefijo y postfijo. Los siguientes ejemplos ilustran las versiones prefijas y postfijas del operador de incremento (+ +).

```
class Cpunto {
private:
  double x, y;
public:
  Cpunto& operator++();              //prefijo
  Cpunto operator++(int);            //postfijo
};

class P {
public:
  void operator++(int);
  void operator--(int);
};
```

B.20. PLANTILLAS *(templates)*

La palabra reservada **`template`** se utiliza para implementar **tipos parametrizados** o **plantillas.** C++ reconoce dos tipos de plantillas: *plantillas de clases* y *plantillas de funciones*.

Sintaxis

```
template < argumentos_plantilla >  //plantilla de clase
  declaración_de_clase

template < argumentos_plantilla >  //plantilla de función
  definición_función

< argumentos_plantilla > es class arg_nombre
```

B.20.1. Plantillas de funciones

Una plantilla de función debe tener al menos un tipo de parámetro, pero puede tener más. Su formato es:

```
template <classT>
tipo-retorno nombre-función(T parámetro)
```

`T` es un parámetro de plantilla

```
template <class T>
void Presentar (const T &valor)
{
  cout << valor;
}
```

Una función plantilla puede tener parámetros adicionales

```
template <class T>
void Intercambio(T& x, T& y)
{
  T aux;
  aux = x;
  x = y;
  y = aux;
}

int m, n;
Estudiante S1;
Estudiante S2;
//...
Intercambio(m, n);           //llamada con enteros
Intercambio(S1, S2);         //llamada con estudiantes
```

B.20.2. Plantillas de clases

Las plantillas de clases ofrecen la posibilidad de generar nuevas clases. Los tipos más comunes de plantillas son clases contenedoras, tales como arrays, listas y conjuntos. Cada plantilla puede trabajar con una variedad de tipos de datos. El formato básico para definir una plantilla de clase es:

```
template <class T>
class MiClase {
  //...
};
```

`T` puede ser un tipo o una expresion. Las plantillas se instancian para crear clases a partir de ellas.

```
MiClase <int> x;
MiClase <Estudiante> miEstudiante;
```

Una plantilla puede tener múltiples parámetros

```
template <class T1, class T2>
class Circulo {
  //...
private:
  Tl x;
  T2 y;
  T2 radio;
};
//...
Circulo <int, long> c;
Circulo <unsigned, float> D;
```

Un ejemplo clásico es una Pila de datos

```
template <class T>
class pila {
  T *pilap;
  int longitud;
  int indice;
public:
  T sacar(void) {return pilap[--indice];}
  void meter(T item){pilap[indice++] = item; }
  ...
};
```

Esta plantilla se puede instanciar del modo siguiente:

```
pila<int> elementos(30);    //pila de enteros
pila<CCad> cadenas(20);     //pila de objetos Ccad
```

B.21. EXCEPCIONES

El *manejo* o *manipulación de excepciones* es el mecanismo para detectar y manipular excepciones. Un *manipulador de excepciones* es un bloque de código que procesa condiciones de error específicas. Una *excepción* es, casi siempre, un error en tiempo de ejecución, tal como «falta de memoria», «fallo al abrir un archivo», «errores de rango de subíndice» o «división por cero».

En C++, cuando se genera una excepción, el error no se puede ignorar o terminará el programa. Un programa lanza o dispara *(throws)* una excepción en el punto en que se detecta el error. Cuando sucede esto, un programa C++ busca automáticamente el *manipulador de excepciones,* que responde a la excepción de algún modo apropiado. Esta respuesta se denomina *«capturar una excepción» (catching)*. Si no se puede encontrar un manipulador de excepciones el programa terminará.

B.21.1. Lanzamiento de excepciones

Pueden ocurrir muchos tipos diferentes de excepciones, de modo que cuando se lanza una excepción, una expresión `throw` *excepción* (o sólo `throw`) identifica el tipo de excepción. La sintaxis adopta dos formatos:

```
throw
throw expresión
//lanzamiento de una excepción, creando un subíndice
//está fuera de rango
const unsigned LongArray = 500;
unsigned i;
.
.
.
if (i > = LongArray)          //es el subíndice válido
   throw ErrorRango;
```

La palabra reservada `try`, junto con las sentencias que le siguen encerradas entre llaves, se llama un *bloque try.* Debe ser seguido inmediatamente por uno o más *manejadores de excepciones*. Cada manejador de excepciones comienza con la palabra reservada `catch` seguida por un bloque que contiene sentencias.

```
try {
  lista_de_sentencias
}
catch (lista_de_parámetros){
  lista_de_sentencias
}
catch (lista_de_parámetros){
  lista_de_sentencias
}
```

B.21.2. Manejador de excepciones

Un bloque `catch` se conoce también como *manejador de excepciones* y se parece a una declaración de función de un argumento sin un tipo de retorno.

```
catch (const char* mensaje)
{
  cerr << mensaje << endl;
  exit(1);
}
```

B.21.3. Especificación de excepciones

Sintácticamente, una especificación de excepciones es parte de una declaración de funciones y tiene el formato:

cabecerafunción `throw (`*lista de tipos*`)`

La lista de tipos son los tipos que pueden tener una sentencia `throw` *expresión* dentro de la llamada de la función; si la lista es vacía, el compilador puede suponer que ninguna sentencia `throw` se ejecutará por la función, bien directa o bien indirectamente

```
void demo() throw(int, desbordamiento);
void nodemo(int i) throw();
```

Así, una declaración de función:

```
void Demo() throw (A, B)
{
  //...
}
```

se tratará del siguiente modo:

```
void Demo()
  { try {
  //...
      llamada a Demo()
  }
catch (A)
    throw;
  }
catch (B)
  }
    //...
  }
catch (...)
{
  //unexpected
}
```

Si una función no está especificada con una declaración `throw` puede lanzar cualquier excepción. La declaración `throw()` indica que una función no lanza ninguna excepción.

Si una función lanza una excepción que no está en su declaración, se llama a la función `unexpected`. En la mayoría de las implementaciones, `unexpected` llama a `terminate` que llama a `abort`. La función `set_unexpected` permite definir el manejo de excepciones no previstas. Todas estas declaraciones de funciones se encuentran en al archivo *except o* bien *except.h*.

B.21.4. Excepciones en la biblioteca estándar

Las excepciones de la biblioteca estándar se derivan de la clase base `exception`. Dos de las clases derivadas son `logic_error` y `runtime_error`. Los errores de tipo lógico incluyen `bad_cast`, `out_of range` y `bad_typeid`, y los errores de tiempo de ejecución incluyen `range_error`, `overflow_error` y `bad_alloc`.

```
class logic_error: public exception
{
  public:
    logic_error;
    {
      const string & que_arg; //...
    }
};
```

Ejemplo de excepción lógica

Los errores lógicos informan de un problema que es detectable *antes* de que un programa se ejecute

```
class domaín-error : public logic_error
{
  public:
    domain_error
    {

    const string & que_arg
}
};
```

Ejemplo de excepción de tiempo de ejecución

Los errores en tiempo de ejecución ocurren durante la ejecución del programa

```
class range_error : public runtime-error
{
  public:
    range_error;
    {
      const string & que_arg;
      }
};
```

B.21.5. Resumen de sintaxis de excepciones

- **throw** valor;
- **try** {
  sentencias
  {
  catch (tipo_excepción){
  sentencias
  }
- try {
  sentencias
  }
  catch (dec_argumentos1) {
  sentencias
  }
  catch (dec_argumentos2) {
  sentencias
  }
  ...
  throw;

B.22. ESPACIO DE NOMBRES (*Namespaces*)

Un espacio de nombres *(namespace)* es un mecanismo para expresar agrupamientos lógicos. Es decir, si algunas declaraciones se pueden agrupar lógicamente de acuerdo a algún criterio, se pueden poner en un espacio de nombres común que expresen dicho hecho. La palabra reservada `namespace` define un espacio de nombres

```
namespace nombre {
  cuerpo_espacio_de_nombres
}
```

Este formato crea un espacio de nombre con el calificador `nombre`. Dentro de los caracteres llave `({ })`, `cuerpo_espacio_de_nombres` puede incluir variables, definiciones de funciones y prototipos, estructuras, clases,

enumeraciones *(enum)*, definiciones de tipos *(typedef)* o bien otras definiciones de espacios de nombres. *Obsérvese* que un espacio de nombre no termina con un punto y coma. Las definiciones de espacio de nombres aparecen en archivos de cabecera y en módulos independientes con definiciones de funciones.

Ejemplo

```
namespace Rojo {
  int j;
  void imprimir(int i)
    {cout << "Rojo::imprimir() " << i << endl;}
}

namespace Azul {
  int j;
  void imprimir(int);
}
void sub1() {...}     //puede acceder a espacio de nombres
                      //Azul y Rojo
void sub() {...}      //puede acceder a espacio de nombres
                      //Azul y Rojo
```

Los espacios de nombre `Rojo` y `Azul` tienen dos miembros con el mismo nombre: entero `j` y función `imprimir()` (normalmente estas definiciones no podrían convivir en un espacio global, pero un espacio de nombre elimina ese problema). Un espacio de nombre puede incluir definiciones de funciones y prototipos de funciones. Las definiciones de funciones de `sub1()` y `sub2()` tienen acceso a todos los miembros del espacio de nombres `Rojo` y `Azul`.

B.22.1. Extensiones

Los espacios de nombre son extensibles; es decir, se pueden añadir declaraciones posteriores a espacios de nombre definidos anteriormente. Las *extensiones* del espacio de nombres pueden aparecer también en archivos independientes de la definición original del espacio de nombre.

```
namespace Azul{ //definición original de espacio de nombre
  int j;
  void imprimir (int);
}

namespace Azul{ //extensión del espacio de nombre
  char car;
  char bufer[20];
}

...
```

B.22.2. Acceso a los miembros de un espacio de nombre

El operador de resolución de ámbito proporciona acceso a miembros de un espacio de nombre.

```
nombre_espacio_de-nombre::nombre_miembro
```

nombre_espacio_de_nombre es un cualificador de espacio de nombre definido con anterioridad y *nombre_miembro* debe estar declarado dentro de `nombre_espacio_de_nombre`.

```
void Azul::imprimir(int k){
  cout << "Azul::imprimir() = " << k << endl;
}
```

El operador de resolución de ámbito asocia `imprimir()` con el espacio de nombre `Azul;` sin el mismo, el compilador define `imprimir()` como una función global.

B.22.3. Espacio de nombre sin nombres

Es muy útil, en ocasiones, envolver un conjunto de declaraciones en un espacio de nombre simplemente para proteger frente a la posibilidad de conflictos de nombres. El formato es:

```
namespace {
  cuerpo_nombre_de_espacio
}
```

Todos los miembros definidos en *cuerpo_nombre_de_espacio* están en un espacio de nombre sin nombre que se garantiza único para cada unidad de traducción.

B.22.4. Directivas *using*

El acceso a los miembros de espacio de nombre puede tener dificultades, especialmente con calificadores de espacio de nombre largos y espacios de nombre anidados. Las directivas `using` proporcionan acceso a todos los miembros del espacio de nombre sin un cualificador de espacio de nombre y el operador de ámbito. El formato es:

```
using namespace nombre;
```

El cualificador *nombre* debe ser un nombre de espacio definido anteriormente. Pueden aparecer las directivas *using* en ámbitos locales y globales.

B.22.5. Declaraciones `using`

Las directivas *using* hacen a todos los miembros del espacio de nombre accesibles sin cualificación. En contraste, las declaraciones `using` cualifican sólo miembros individuales de espacio de nombre. El formato es:

```
using nombre_espacio_de_nombre::nombre-miembro;
```

```
namespace Negro{          //define espacio de nombre Negro
  int j;
  void imprimir(int);
  char car;
}
namespace  Blanco{
  int j;
  void imprimir(int);
  double vision;
}

using Blanco::vision;
```

B.22.6. Alias de espacio de nombres

Los alias de espacio de nombres crean nombres sinónimos de espacio de nombres. Es buena práctica utilizar alianzas cuando un espacio de nombre tiene un nombre largo. El formato es:

```
namespace nombre_alias = nombre_de_espacio;
```

Ejemplo

```
namespace Universidad_Pontificia-deSalamanca_enMadrid{
  //...
}
Universidad_Pontificia_deSalamanca_enMadrid::String c3 =
  "Fundación";
Universidad_Pontificia_deSalamanca_enMadrid::String c4  =
  "Pablo VI";
```

El código anterior es poco práctico en código real; un alias corto resuelve el inconveniente

```
// uso de alias para abreviaturas
namespace UPSAM = Universidad_Pontificia_deSalamanca_enMadrid;

UPSAM::String c3 = "Fundación";
UPSAM::String c4 = "Pablo VI";

namespace Geometria{
  struct Punto {
    double x, y;
  };
  double pendiente (Punto, Punto);
}
```

El espacio de nombres incluye un tipo `Punto` y una función `pendiente`. La implementación del archivo `geometria.c`

```
// geometria.c
#include "geometria.h"

namespace Geometria {   // extensión de espacio de nombre
  Punto origen = {0, 0};
}

double Geometria::pendiente (Punto p1, Punto p2){
  double dy = p2.y - p1.y;
  double dx = p2.x - p1.x;
  if (dx = = 0)
    throw "pendiente() : pendiente no definida";
  return dy / dx;
}
```

Una extensión del espacio de nombre ha añadido un miembro `origen` a `Geometria`. La función `pendiente()` calcula la pendiente de los puntos `p1` y `p2` y levanta una excepción de cadena de caracteres si el denominador de la pendiente es cero.

B.22.7. Espacio de nombres anidados

Los espacios de nombres son ilegales en el interior de funciones, pero pueden aparecer en otras definiciones de espacio de nombre.

```
namespace Exterior
  int i, j;
namespace Interior
  const int Max = 20;
  char car;
  char bufer[Max];
  void imprimir();
  }
}
```

El acceso a los miembros de un espacio de nombres anidados se resuelve con el operador de resolución de ámbito (::)

```
void Exterior::Interior::imprimir() {
  for (int i = 0; i < Max; i++)  //i es local al bucle for
    cout << bufer[i] << '';
  cout << endl;
}
```

B.22.8. Un programa completo con espacio de nombres

El archivo de cabecera `geometria.h` define un espacio de nombre para una biblioteca de geometria (`geometria`).

```
#ifndef GEOMETRIAH
#define GEOMETRIAH
// archivo geometria.h
```

Aplicación. Utilizar el espacio de nombre `Geometria` para calcular pendientes de las variables `Punto`.

```
#include <iostream.h>
#include "geometria.h"

namespace Geo = Geometria;           //alias

using Geo::Punto;
using Geo::pendiente;

namespace {
  Punto origen = {10, 10};
}
int main()
{
  try {
  Punto a = {3, 5}
  Punto b = {6, 10};
  cout    << "La línea ab tiene la pendiente"
          << pendiente(a,b) << endl;
  cout    << "La linea origen_a tiene la pendiente"
          << pendiente (origen,a)
          << endl;
}
catch (char *msg) {
  cout << msg << endl;
  return 1;
  }
  return 0;
}
#endif
```

Al ejecutar el programa se obtendrá:

```
La línea ab tiene la pendiente 1.66667
La línea origen_a tiene la pendiente 0.714286
```