

# Lenguaje C++

## Introducción

Un concepto muy importante introducido por la programación estructurada es la abstracción. La abstracción se puede definir como la capacidad de examinar algo sin preocuparse de los detalles internos. En un programa estructurado es suficiente conocer que un procedimiento dado realiza una tarea específica. El cómo se realiza la tarea no es importante; mientras el procedimiento sea fiable se puede utilizar sin tener que conocer cómo funciona su interior. Esto se conoce como **abstracción funcional**.

Una debilidad de la programación estructurada aparece cuando programadores diferentes trabajan en una aplicación como un equipo. Dado que programadores diferentes manipulan funciones separadas que pueden referirse a tipos de datos mutuamente compartidos, los cambios de un programador se deben reflejar en el trabajo del resto del equipo. Otro problema de la programación estructurada es que raramente es posible anticipar el diseño de un sistema completo antes de que se implemente realmente.

En esencia, un defecto de la programación estructurada, como se acaba de ver, consiste en la separación conceptual de datos y código. Este defecto se agrava a medida que el tamaño del programa crece.

**Abstracción de datos:** permite no preocuparse de los detalles no esenciales. Existe en casi todos los lenguajes de programación. Las estructuras de datos y los tipos de datos son un ejemplo de abstracción. Los procedimientos y funciones son otro ejemplo. Sólo recientemente han emergido lenguajes que soportan sus propios tipos abstractos de datos (TAD), como Pascal, Ada, Modula-2 y C++.

### ¿Qué es la programación orientada a objetos?

Se puede definir **POO** como una técnica o estilo de programación que utiliza objetos como bloque esencial de construcción.

Los objetos son en realidad como los **tipos abstractos de datos**. Un **TAD** es un tipo definido por el programador junto con un conjunto de operaciones que se pueden realizar sobre ellos. Se denominan abstractos para diferenciarlos de los tipos de datos fundamentales o básicos.

En **C** se puede definir un tipo abstracto de datos utilizando typedef y struct y la implementación de las operaciones con un conjunto de funciones.

**C++** tiene muchas facilidades para definir y utilizar un tipo **TAD**.

Al igual que los tipos de datos definidos por el usuario, un objeto es una colección de datos, junto con las funciones asociadas, utilizadas para operar sobre esos datos. Sin embargo la potencia real de los objetos reside en las propiedades que soportan: herencia, encapsulación y polimorfismo, junto con los conceptos básicos de objetos, clases, métodos y mensajes.

### Trabajando con objetos

En programación convencional los programas se dividen en dos componentes: procedimientos y datos. Este método permite empaquetar código de programa en procedimientos, pero ¿qué sucede con los datos? Las estructuras de datos utilizadas en programación son globales o se pasan como parámetros.

En esencia los datos se tratan separadamente de los procedimientos.

En **POO** un programa se divide en componentes que contienen procedimientos y datos. Cada componente se considera un objeto.

Un objeto es una unidad que contiene datos y las funciones que operan sobre esos datos. A los elementos de un objeto se les conoce como miembros; las funciones que operan sobre los datos se denominan métodos (en **C++** también se llaman funciones miembro) y los datos se denominan miembros datos. En **C++** un programa consta de objetos. Los objetos de un programa se comunican entre sí mediante el paso o envío de mensajes (acciones que debe ejecutar el objeto).

En **POO** los objetos pueden ser cualquier entidad del mundo real:

- Objetos físicos:

- \* automóviles en una simulación de tráfico
- \* aviones en un sistema de control de tráfico aéreo
- \* animales mamíferos, etc.

- Elementos de interfaces gráficas de usuarios

- \* ventanas
- \* iconos
- \* menús
- \* ratones

- Estructuras de datos

- \* arrays
- \* pilas
- \* árboles binarios

- Tipos de datos definidos por el usuario

- \* números complejos
- \* hora del día

### **Definición de objetos**

Un **objeto** es una unidad que contiene datos y las funciones que operan sobre esos datos. Los datos se denominan miembros dato y las funciones métodos o funciones miembro.

Los datos y las funciones se encapsulan en una única entidad. Los datos están ocultos y sólo mediante las funciones miembro es posible acceder a ellos.

### **Clases**

Una **clase** es un tipo definido por el usuario que determina las estructuras de datos y las operaciones asociadas con ese tipo. Cada vez que se construye un objeto de una clase, se crea una instancia de esa clase. En general, los términos objetos e instancias de una clase se pueden utilizar indistintamente.

Una clase es una colección de objetos similares y un objeto es una instancia de una definición de una clase.

La comunicación con el objeto se realiza a través del paso de mensajes. El envío a una instancia de una clase produce la ejecución de un método o función miembro. El paso de mensajes es el término utilizado para referirnos a la invocación o llamada de una función miembro de un objeto.

## Mensajes: activación de objetos

Los objetos pueden ser activados mediante la recepción de mensajes. Un mensaje es simplemente una petición para que un objeto se comporte de una determinada manera, ejecutando una de sus funciones miembro. La técnica de enviar mensajes se conoce como paso de mensajes.

### Estructuralmente un mensaje consta de tres partes:

- la identidad del objeto receptor
- la función miembro del receptor cuya ejecución se ha solicitado
- cualquier otra información adicional que el receptor pueda necesitar para ejecutar el método requerido.

En C++, la notación utilizada es **nombre\_del\_objeto.función\_miembro**

Ejemplo: Se tiene un objeto **o1** con los siguientes miembros dato: **nombre\_alumno** y **curso** y con las funciones miembro: **leer\_nombre** e **imprimir**. Si el objeto **o1** recibe el mensaje imprimir, esto se expresa: **o1.imprimir()**

La sentencia anterior se lee: "enviar mensaje imprimir al objeto o1". El objeto **o1** reacciona al mensaje ejecutando la función miembro de igual nombre que el mensaje.

El mensaje puede llevar parámetros: **o1.leer\_nombre("Pedro Pérez")**

Sin los mensajes los objetos que se definan no podrán comunicarse con otros objetos. Desde un punto de vista convencional, el paso de mensajes no es más que el sinónimo de llamada a una función.

## Programa orientado a objetos

Un programa orientado a objetos es una colección de clases. Necesitará una función principal que cree objetos y comience la ejecución mediante la invocación de sus funciones miembro.

Esta organización conduce a separar partes diferentes de una aplicación en distintos archivos. La idea consiste en poner la descripción de la clase para cada una de ellas en un archivo separado. La función principal también se pone en un archivo independiente. El compilador ensamblará el programa completo a partir de los archivos independientes en una única unidad.

En realidad, cuando se ejecuta un programa orientado a objetos, ocurren tres acciones:

1. Se crean los objetos cuando se necesitan
2. Los mensajes se envían desde uno objetos y se reciben en otros
3. Se borran los objetos cuando ya no son necesarios y se recupera la memoria ocupada por ellos

## Herencia

La **herencia** es la propiedad que permite a los objetos construirse a partir de otros objetos.

Una clase se puede dividir en subclases. En **C++** la clase original se denomina clase base; las clases que se definen a partir de la clase base, compartiendo sus características y añadiendo otras nuevas, se denominan clases derivadas.

Las clases derivadas pueden heredar código y datos de su clase base añadiendo su propio código y datos a la misma.

La **herencia** impone una relación jerárquica entre clases en la cual una clase hija hereda de su clase padre. Si una clase sólo puede recibir características de otra clase base, la herencia se denomina herencia simple.

Si una clase recibe propiedades de más de una clase base, la herencia se denomina herencia múltiple.

## Polimorfismo

En un sentido literal, significa la cualidad de tener más de una forma. En el contexto de **POO**, el polimorfismo se refiere al hecho de que una misma operación puede tener diferente comportamiento en diferentes objetos. Por ejemplo, consideremos la operación sumar. El operador **+** realiza la suma de dos números de diferente tipo. Además se puede definir la operación de sumar dos cadenas mediante el operador **suma.e**

## Comentarios

C++ soporta dos tipos de comentarios:

1. Viejo estilo C: Entre **/\*** y **\*/**.
2. Nuevo estilo C++: Comienzan con **//**. Su efecto termina cuando se alcanza el final de la línea actual.

## Identificadores

Son los nombres elegidos para las variables, constantes, funciones, clases y similares. El primer carácter debe ser una letra o un subrayado. El resto del nombre puede contener dígitos. Los identificadores que comienzan con dos subrayados están reservados para uso interno del compilador C++.

## Constantes

C++ permite utilizar varios tipos de constantes:

1. Constantes enteras **44 0 -345**
2. Constantes enteras muy grandes. Se identifican situando una **L** al final de la constante entera **33L -105L**
3. Constantes octales o hexadecimales. Un **0** a la izquierda indica una constante octal y un **0x** o bien **0X** indican una constante hexadecimal 0 02 077 0123 equivalen a 0 2 63 83 en octal 0x0 0x2 0x3F 0x53 equivalen a 0 2 63 83 en hexadecimal
4. Constantes reales (coma flotante) 0.0 3.1416 -99.2 C++ permite especificar constante de coma flotante de simple precisión (sufijo **f** o **F**) y doble precisión larga (sufijo **l** o **L**). 32.0f 3.1416L
5. Constantes carácter 'z' '5'
6. Constantes cadena "hola" "hoy es lunes"

## Tipos de datos

C++, igual que C, contiene tipos fundamentales y tipos derivados o estructurados.

Los fundamentales son: **int, char, long int, float, double, long double**.

- **Tipo vacío**. El tipo vacío (void) se utiliza principalmente para especificar:
  - \* Funciones que no devuelven valores.
  - \* Punteros void, que referencian a objetos cuyo tipo es desconocido.

- **Tipos enumerados**. Un tipo enumerado o enumeración está construido por una serie de constantes simbólicas enteras. Los tipos enumerados se tratan de modo ligeramente diferente en C++ que en ANSI C. El nombre de la etiqueta **enum** se considera como un nombre de tipo igual que las etiquetas

de **struct** y **union**. Por tanto se puede declarar una variable de enumeración, estructura o **union** sin utilizar las palabras **enum**, **strcut** o **union**.

C define el tipo de **enum** de tipo **int**. En C++, sin embargo, cada tipo enumerado es su propio tipo independiente. Esto significa que C++ no permite que un valor **int** se convierta automáticamente a un valor **enum**. Sin embargo, un valor enumerado se puede utilizar en lugar de un int.

Ejemplo:  
enum lugar{primero,segundo,tercero};  
lugar pepe=primero; //correcto  
int vencedor=pepe; //correcto  
lugar juan=1; //incorrecto

La última sentencia de asignación es aceptable en C pero no en C++, ya que 1 no es un valor definido en lugar.

- **Tipos referencia**. Las referencias son como alias. Son alternativas al nombre de un objeto. Se define un tipo referencia haciéndole preceder por el operador de dirección **&**. Un objeto referencia, igual que una constante debe ser inicializado.

```
int a=50;  
int &refa=a; //correcto  
int &ref2a; //incorrecto: no inicializado
```

Todas las operaciones efectuadas sobre la referencia se realizan sobre el propio objeto:  
refa+=5; equivale a sumar 5 a a, que vale ahora 55  
int \*p=&refa; inicializa p con la dirección de a

### Operadores especiales de C++

:: Resolución de ámbito (o alcance)  
.\* Indirección (eliminación de referencia directa) un puntero a un miembro de una clase  
->\* Indirección (eliminación de referencia directa) un puntero a un miembro de una clase  
**new** Asigna (inicializa) almacenamiento dinámico  
**delete** Libera almacenamiento asignado por new

### Declaraciones y definiciones

Los términos declaración y definición tienen un significado distinto aunque con frecuencia se intercambian.

Las declaraciones se utilizan para introducir un nombre al compilador, pero no se asigna memoria. Las definiciones asignan memoria.

En **C++** cuando se declara una estructura se proporciona su nombre al compilador pero no se asigna memoria. Cuando se crea una instancia de la estructura es cuando se asigna memoria.

En C todas las declaraciones dentro de un programa o función deben hacerse al principio del programa o función; en otras palabras las declaraciones dentro de un ámbito dado deben ocurrir al principio de ese ámbito.

Todas las declaraciones globales deben aparecer antes de cualquier función y cualquier declaración local debe hacerse antes de cualquier sentencia ejecutable.

**C++**, por el contrario, permite mezclar datos con funciones y código ejecutable: trata una declaración como un tipo de sentencia y permite situarla en cualquier parte como tal. Esta característica de **C++** es muy cómoda ya que permite declarar una variable cuando se necesite e inicializarla inmediatamente.

El ámbito de una variable es el bloque actual y todos los bloques subordinados a él. Su ámbito comienza donde aparece la declaración. Las sentencias **C++** que aparecen antes de la declaración no pueden referirse a la variable incluso aunque aparezcan en el mismo bloque que la declaración de la variable.

### Moldes (cast)

**C++** soporta dos formas diferentes de conversiones forzosas de tipo explícitas:

```
int f=0;
long l= (long) f; //molde tradicional, tipo C
long n = long (f); //molde nuevo, tipo C++
```

La sintaxis: **nombre\_tipo (expresión)** se conoce como **notación funcional** y es preferible por su legibilidad.

### El especificador constante (const)

Una constante es una entidad cuyo valor no se puede modificar, y en **C++**, la palabra reservada **const** se utiliza para declarar una constante.

```
const int longitud = 20;
char array[longitud]; // válido en C++ pero no en C
```

Una vez que una constante se declara no se puede modificar dentro del programa.

En **C++** una constante debe ser inicializada cuando se declara, mientras que en ANSI C, una constante no inicializada se pone por defecto a 0.

La diferencia más importante entre las constantes en C++ y C es el modo como se declaran fuera de las funciones. En ANSI C se tratan como constantes globales y se pueden ver por cualquier archivo que es parte del programa. En C++, las constantes que se declaran fuera de una función tienen ámbito de archivo por defecto y no se pueden ver fuera del archivo en que están declaradas. Si se desea que una constante se vea en más de un archivo, se debe declarar como **extern**.

En C++ las constantes se pueden utilizar para sustituir a **#define**.

```
En C:
#define PI 3.141592
#define long 128
```

```
En C++:
const PI = 3.141592
const long = 128
```

### Punteros y direcciones de constantes simbólicas

No se puede asignar la dirección de una constante no simbólica a un puntero no constante, pero sí se puede desreferenciar el puntero y cambiar el valor de la constante.

```
const int x=6;
int *ip;
```

```
ip = &x; //error
*ip=7;
```

### Punteros a un tipo de dato constante

Se puede declarar un puntero a un tipo de dato constante pero no se puede desreferenciar.

```
const int x=6;
const int *ip; //puntero a una constante entera
int z;
ip=&x;
ip=&z;
*ip=10; //error: no se puede desreferenciar este tipo de puntero y modificar z
z=7; //válido: z no es una constante y se puede cambiar.
```

El puntero a un tipo constante es muy útil cuando se desea pasar una variable puntero como un argumento a una función pero no se desea que la función cambie el valor de la variable a la que está apuntada.

```
struct ejemplo
{
int x;
int y;
};
void funcion(const ejemplo *);
void main()
{
ejemplo e={4,5};
funcion(&e); //convertirá un puntero no constante a un puntero const
}
void funcion(const ejemplo *pe)
{ ejemplo i;
i.x=pe->x;
i.y=pe->y;
pe->x=10; } // error: no se puede desreferenciar pe.
```

### Punteros constantes

Se puede definir una constante puntero en **C++**. Las constantes puntero se deben inicializar en el momento en que se declaran. Se leen de derecha a izquierda.

```
void main()
{
int x=1,y;
int *const xp= &x; // xp es un puntero constante a un int
xp=&y; // error: el puntero es una constante y no se puede asignar un nuevo valor a punteros
constantes
*xp=4;
}
```

### El especificador de tipo volatile

La palabra reservada **volatile** se utiliza sintácticamente igual que **const**, aunque en cierto sentido tiene sentido opuesto.

La declaración de una variable volátil indica al compilador que el valor de esa variable puede cambiar en cualquier momento por sucesos externos al control del programa. En principio, es posible que las

variables volátiles se puedan modificar no sólo por el programador sino también por hardware o software del sistema (rutina de interrupción).volatile int puerto;

### Sizeof (char)

En **C**, todas las constantes **char** se almacenan como enteros. Esto significa que en **C** un carácter ocupa lo mismo que un entero. En **C++** un **char** se trata como un **byte**, y no como el tamaño de un int. Por ejemplo en una máquina con una palabra de 4 bytes sizeof('a') se evalúa a 1 en C++ y a 4 en C.

### Punteros a void

En C, una variable puntero siempre apunta a un tipo de dato específico, de modo que el tipo de dato es importante para aritmética de punteros. Si se intenta inicializar una variable puntero con el tipo de dato incorrecto, el compilador genera un mensaje de error. En C se pueden moldear los datos al tipo de dato correcto:

```
void main()
{
int a=1,*p;
double x=2.4;
p=&a; //válido: puntero y variable son del mismo tipo
p=&x; //errorp=(int *)&x; //válido
}
```

En C++ se puede crear un puntero genérico que puede recibir la dirección de cualquier tipo de dato.

```
void main()
{
void *p;
int a=1;
double x=2.4;
p=&a;
p=&x;
}
```

No se puede desreferenciar un puntero

```
void main()
{ void *p;
double x=2.5;
p=&x;
*p=3.6; // error: se desreferencia a un puntero void
}
```

### Salidas y entradas

Las operaciones de salida y entrada se realizan en C++, al igual que en C, mediante flujos (streams) o secuencias de datos. Los flujos estándar son cout (flujo de salida) y cin (flujo de entrada). La salida fluye normalmente a la pantalla y la entrada representa los datos que proceden de teclado. Ambos se pueden redireccionar.

#### Salida

El flujo de **salida** se representa por el identificador cout, que es en realidad un objeto. El operador << se denomina operador de inserción y dirige el contenido de la variable situada a su derecha al objeto situado a su izquierda. El equivalente en C de **cout** es **printf**.

El archivo de cabecera **iostream.h** contiene las facilidades **standard** de entrada y salida de **C++**.

En C++, los dispositivos de salida **estandar** no requieren la cadena de formato.

Se pueden utilizar también diferentes tipos de datos, enviando cada uno de ellos a la vez al flujo de salida. El flujo **cout** discierne el formato del tipo de dato, ya que el compilador C++ lo descifra en el momento de la compilación.

El operador de inserción se puede utilizar repetidamente junto con **cout**.

```
include <iostream.h>
void main()
{
int a=4;
float b=3.4;
char *texto="hola\n";
cout<< "entero " << a << " real " << b << " mensaje " << texto;
}
```

### Salida con formato

C++ asocia un conjunto de manipuladores con el flujo de salida, que modifican el formato por defecto de argumentos enteros. Por ejemplo, valores simbólicos de manipuladores son `dec`, `oct` y `hex` que visualizan representaciones decimales, octales y hexadecimales de variable.

### Entrada

C++ permite la entrada de datos a través del flujo de entrada **cin**. El objeto **cin** es un objeto predefinido que corresponde al flujo de entrada **estandar**. Este flujo representa los datos que proceden del teclado. El operador `>>` se denomina de extracción o de lectura de. Toma el valor del objeto flujo de su izquierda y lo sitúa en la variable situada a su derecha.

### El operador de resolución de ámbito ::

C es un lenguaje estructurado por bloques. C++ hereda la misma noción de bloque y ámbito. En ambos lenguajes, el mismo identificador se puede usar para referenciar a objetos diferentes. Un uso en un bloque interno oculta el uso externo del mismo nombre. C++ introduce el operador de resolución de ámbito o de alcance.

El operador `::` se utiliza para acceder a un elemento oculto en el ámbito actual. Su sintaxis es `::`

#### variable

Ejemplo:

```
#include <iostream.h>
int a;
void main()
{
float a;
a=1.5;
::a=2;
cout << "a local " << a << "\n";
cout << "a global " << ::a << "\n";
}
```

Este programa visualiza:

```
a local 1.5
a global 2
```

Este operador se utilizará también en la gestión de clases.

## Estructuras y uniones

Los tipos definidos por el usuario se definen como estructuras, uniones, enumeraciones o clases. Las estructuras y uniones son tipos de clase (clase, es el tipo de dato fundamental en la programación orientada a objetos).

El tipo **struct** permite agregar componentes de diferentes tipos y con un solo nombre. Las estructuras en C++ se declaran como en C:

```
struct datos
{
int num;
char nombre[20];
};
```

El nuevo tipo de dato definido puede tener instancias que se declaran: `datos persona; //declaración C++`  
`+struct datos persona; // declaración C`

En C++ debe evitarse el uso de **typedef**.

El mismo convenio aplicado a las estructuras, se aplica a las uniones.

Un tipo especial de unión se ha incorporado a C++ :unión anónima. Una unión anónima declara simplemente un conjunto de elementos que comparten la misma dirección de memoria; no tiene nombre identificador y se puede acceder directamente a los elementos por su nombre.

Un ejemplo:

```
union
{
int i;
float f;
};
```

Tanto **i** como **f** comparten la misma posición de memoria. A los miembros de esta unión se puede acceder directamente en el ámbito en que está declarada. Por tanto, en el ejemplo, la sentencia **i=3** sería aceptable.

Las uniones anónimas son interesantes en el caso en que se defina una unión en el interior de una estructura:

```
struct registro
{
union
{
int num;
float salario;
};
char *telefono;
};
registro empleado;
```

Para acceder al nombre de un campo dentro de esta estructura:

```
empleado.sueldo;
```

## Asignación dinámica de memoria: new y delete

En C la asignación dinámica de memoria se manipula con las funciones **malloc()** y **free()**. En C++ se define un método de hacer asignación dinámica utilizando los operadores **new** y **delete**.

```
En C:
void main()
{
int *z;
z= (int*) malloc(sizeof(int));
*z=342;
printf("%d\n",*z);
free(z);
}
```

```
En C++:
void main()
{
int *z=new int;
*z=342;
cout << z;
delete z;
}
```

El operador new está disponible directamente en C++, de modo que no se necesita utilizar ningún archivo de cabecera; new se puede utilizar con dos formatos: new tipo // asigna un único elemento

new tipo[num\_eltos] // asigna un array

Si la cantidad de memoria solicitada no está disponible, el operador new proporciona el valor 0. El operador **delete** libera la memoria signada con new.

**delete variable**  
**delete [n] variable**

**new** es superior a **malloc** por tres razones:

- 1.- **new** conoce cuánta memoria se asigna a cada tipo de variable.
- 2.- **malloc()** debe indicar cuánta memoria asignar.
- 3.- **new** hace que se llame a un constructor para el objeto asignado y **malloc** no puede.

**delete** produce una llamada al destructor en este orden:

1. Se llama al destructor
2. Se libera memoria

**delete** es más seguro que **free()** ya que protege de intentar liberar memoria apuntada por un puntero nulo.

## La biblioteca iostream

C++ proporciona una nueva biblioteca de funciones que realizan operaciones de **E/S**: la biblioteca **iostream**. Esta biblioteca es una implementación orientada a objetos y está basada, al igual que **stdio**, en el concepto de flujos. Cuando se introducen caracteres desde el teclado, puede pensarse en caracteres que fluyen desde el teclado a las estructuras de datos del programa. Cuando se escribe en un archivo, se piensa en un flujo de bytes que van del programa al disco.

Para acceder a la biblioteca **iostream** se debe incluir el archivo **iostream.h**. Este archivo contiene información de diferentes funciones de **E/S**. Define también los objetos **cin** y **cout**.

### Manipuladores de salida

La biblioteca **iostream** define varios operadores particulares, llamados manipuladores, que le permiten controlar precisamente, el formato de los datos visualizados. Situando un manipulador en la cadena de operadores **<<**, se puede modificar el estado del flujo.

Una característica importante de un flujo que debe tratar con valores numéricos es la base de los números. Hay tres manipuladores (**dec**, **hex** y **oct**) para controlar la situación. La base por omisión es **10** y por tanto sólo será necesario indicar **dec** cuando se haya fijado la base a otro valor:

```
cout <<oct<<x<<endl;
cout <<dec<<x<<endl;
```

Los manipuladores que toman argumentos se declaran en **iomanip.h**, el resto en **iostream.h**.

A continuación mostraremos un listado con los manipuladores, su aplicación y la descripción. Cada uno de ellos lo separaremos mediante --

```
dec -- cout<<dec<<x; -- Conversión a decimal
dec -- cin>>dec>>x; -- Conversión a decimal
Hex -- out<<hex<<x; -- conversión a hexadecimal
Hex -- cin>>hex>>x; -- conversión a hexadecimal
oct -- cout<<oct<<x; -- Conversión a octal
oct -- cin>>oct>>x; -- conversión a octal
ws -- cin>>ws; Salta espacios en la entrada
ende -- cout<<endl; -- Envía carácter fin de línea
flush -- cout<<flush; -- Limpia el buffer
setfill(int) -- cout<<setfill('*'); -- Fija el carácter de relleno
setprecision(int) -- cout<<setprecision(6); -- Fija la conversión en coma flotante al nº de dígitos especificado
setw(int) -- cout<<setw(6)<<x; cin>>setw(10)>>x; -- Fija la anchura
```

Con **setw()** los valores numéricos se justifican a derechas y los datos carácter a izquierdas.

La información de la justificación de la salida se almacena en un modelo o patrón de bits de una clase llamada **ios**, que constituye la base de todas las clases de flujos. Puede establecer o reinicializar bits específicos utilizando los manipuladores **setiosflags()** y **resetiosflags()** respectivamente.

Para utilizar cualquiera de los indicadores de formato hay que insertar el manipulador **setiosflags()** con el nombre del indicador como argumento. Hay que utilizar **resetiosflags()** con el mismo argumento para invertir el estado del formato antes de utilizar el manipulador **setiosflags()**.

```
Indicador -- Significado del indicador activado
ios::left -- Justifica la salida a la izquierda
ios::right -- Justifica la salida a la derecha
ios::scientific -- Utiliza notación científica para números de coma flotante
ios::fixed -- Utiliza notación decimal para números de coma flotante
ios::dec -- Utiliza notación decimal para enteros
ios::hex -- Utiliza notación hexadecimal para enteros
ios::oct -- Utiliza notación octal para enteros
ios::showpos -- Muestra un signo positivo cuando se visualizan valores positivos
```

Al igual que en C, un programa en C++ consta de una función principal: **main** y un número indeterminado de otras funciones. C++ requiere que todas las funciones tengan prototipos.

## Punteros a void en funciones

El uso más importante de punteros void en C++ es pasar la dirección de tipos de datos diferentes en una llamada a función cuando no se conoce por anticipado que tipo de dato se pasa.

```
#include <iostream.h>
enum dato{caracter,real,entero,cadena};
void ver(void *,dato);
```

```
void main()
{
char a='b';
int x=3;
double y=4.5;
char *cad="hola";
ver(&a,caracter);
ver(&x,entero);
ver(&y,real);
ver(cad,cadena);
}
```

```
void ver( void *p, dato d)
{
switch(d)
{
case caracter: printf("%c\n",*(char *)p);
break;
case entero: printf("%d\n",*(int *)p);
break;
case real: printf("%ld\n",*(double *)p);
break;
case cadena: printf("%s\n",*(char *)p);
}
}
```

En este ejemplo no se conoce por anticipado cuál es el tipo del valor que se pasará como argumento a la función ver.

## Compilación separada

Un programa C++ consta de uno o más archivos fuente que se compilan y enlazan juntos para formar un programa ejecutable.

La mayoría de las bibliotecas contienen un número significativo de funciones y variables. Para ahorrar trabajo y asegurar la consistencia cuando se hacen declaraciones externas de estos elementos, C++ utiliza un dispositivo denominado archivo de cabecera. Un archivo de cabecera es un archivo que contiene las declaraciones externas de una biblioteca. Estos archivos tienen extensión .h.

Ejemplo:  
archivo max.h  
int max(int,int); //prototipo de la función  
archivo maximo.cpp  
int max(int x, int y) //definición de la función  
{  
if (x>y) return(x);  
return(y);  
}

```

archivo principal.cpp
#include <iostream.h>
#include "max.h"
void main()
{
int a=5,b=6;
cout<<"mayor "<<max(a,b);
}

```

### Variable referencia

Una referencia o variable referencia en C++ es simplemente otro nombre o alias de una variable. En esencia una referencia actúa igual que un puntero (contiene la dirección de un objeto), pero funciona de diferente modo, ya que no se puede modificar la variable a la que está asociada la referencia, pero sí se puede modificar el valor de la variable asociada.

```

Usando variable referencia
int i;
int &x=i; // x es un alias de i
x=40; // i vale 40

```

```

Usando punteros
int i;
int *p=&i;
*p=40; //i vale 40

```

### Parámetros por valor y por referencia

En C++ el paso por valor significa que al compilar la función y el código que llama a la función, ésta recibe una copia de los valores de los parámetros que se le pasan como argumentos. Las variables reales no se pasan a la función, sólo copias de su valor.

Cuando una función debe modificar el valor de la variable pasada como parámetro y que esta modificación retorne a la función llamadora, se debe pasar el parámetro por referencia. En este método, el compilador no pasa una copia del valor del argumento; en su lugar, pasa una referencia, que indica a la función dónde existe la variable en memoria.

La referencia que una función recibe es la dirección de la variable. Es decir, pasar un argumento por referencia es, simplemente, indicarle al compilador que pase la dirección del argumento.

```

Ejemplo:
void demo(int &valor)
{
valor=5;
cout<<valor<<endl;
}

```

```

void main()
{int n=10;
cout<<n<<endl;
demo(n);
cout<<n<<endl;
}

```

La salida de este programa será: **10 5 5**

Una limitación del método de paso por referencia es que se pueden pasar sólo variables a la función. No se pueden utilizar constantes ni expresiones en la línea de llamada a la misma.

### Los modificadores **const** y **volatile**

El número y tipo de argumentos de una definición de función debe coincidir con su correspondiente prototipo de función. Si no existe correspondencia exacta, el compilador C++ intentará convertir el tipo de los argumentos reales de modo que se puedan concordar con los argumentos formales en las funciones llamadas. Los modificadores **const** y **volatile** pueden preceder a los tipos de los argumentos formales para indicar instrucciones específicas al compilador.

Se puede preceder a un tipo de argumento con el modificador **const** para indicar que este argumento no puede ser modificado. Al objeto al cual se aplica no se le puede asignar un valor o modificar de ningún modo.

El **modificador volatile** se puede utilizar para indicar que los argumentos formales pueden ser modificados en la ejecución del programa, bien por el propio programador o por el propio sistema, es decir, cualquier suceso externo al programa. Típicamente, las variables volátiles se necesitan sólo para valores que pueden ser modificados por una rutina de interrupción que opera independientemente del programa. La declaración de una variable volátil es **volatile int k;**

**Funciones con número de argumentos no especificado** C++ permite declarar una función cuyo número y tipo de argumentos no son conocidos en el momento de la compilación. Esta característica se indica con la ayuda del operador especial puntos suspensivos (...) en la declaración de la función. void f1(...);

Esta declaración indica que f1 es una función que no devuelve ningún valor y que tomará un número variable de argumentos en cada llamada. Si algunos argumentos se conocen, deben situarse al principio de la lista: void f2(int a, float b, ...);

Los puntos suspensivos indican que la función se puede llamar con diferentes conjuntos de argumentos en distintas ocasiones. Este formato de prototipo reduce la cantidad de verificación que el compilador realiza. Así, la función predefinida **printf()**, que tiene el prototipo: **int printf(char \*formato, ...);**

Esto significa que la función devuelve un entero y acepta un puntero a un parámetro fijo de caracteres y cualquier número de parámetros adicionales de tipo desconocido. Con este prototipo, el compilador verifica en tiempo de compilación los parámetros fijos, y los parámetros variables se pasan sin verificación de tipos.

El archivo **stdarg.h** contiene macros que se pueden utilizar en funciones definidas por el usuario con número variable de parámetros.

En **C++** hay otro método para proporcionar argumentos a funciones que tienen listas de argumentos en número variable. Este método facilita al compilador toda la información necesaria en una lista con un número variable de argumentos, proporcionándole un conjunto de variables preescritas que le indican cuántos argumentos contiene la lista y qué tipo de datos son.

Las macros que se pueden utilizar para este propósito están definidas en un archivo de cabecera denominado **stdarg.h** y son **va\_start()**, **va\_arg()** y **va\_end()**.

Las macros del archivo **stdarg.h** tienen dos características útiles:  
- Han sido escritas para uso del programador y son bastantes versátiles, de modo que se pueden utilizar de diversas formas.

- Las macros proporcionan un medio transportables para pasar argumentos variables. Esto se debe a que están disponibles dos versiones de macros: las definidas en **stdarg.h**, que siguen el **standard ANSI C**, y las definidas en **varargs.h**, que son compatibles con UNIX **System V**.

La macro **va\_list** es un tipo de dato que es equivalente a una lista de variables. Una vez que se define una variable **va\_list**, se puede utilizar como un parámetro de las macros **va\_start()** y **va\_end()**.

La sintaxis de la macro **va\_start()** es: **void va\_start(va\_list arg\_ptr, prev\_param);**

**arg\_ptr** apunta al primer argumento opcional en una lista variable de argumentos pasados a una función. Si la lista de argumentos de una función contiene parámetros que se han especificado en la declaración de la función, el argumento **prev\_param** de **va\_start()** proporciona el nombre del argumento especificado de la función que precede inmediatamente al primer argumento opcional de la lista de argumentos.

Cuando la macro **va\_start()** se ejecuta, hace que el parámetro **arg\_ptr** apunte al argumento especificado por **prev\_param**.

La sintaxis de la macro **va\_arg()** es: **void va\_arg(va\_list arg\_ptr, tipo);**

La macro **va\_arg()** tiene un propósito doble:

- Primero, **va\_arg()** devuelve el valor del objeto apuntado por el argumento **arg\_ptr**.
- Segundo, **va\_arg()** incrementa **arg\_ptr** para apuntar al siguiente elemento de la lista variable de argumentos de la función que se está llamando, utilizando el tamaño tipo para determinar dónde comienza el siguiente argumento.

La sintaxis de la macro **va\_end()** es: **void va\_end(va\_list arg\_ptr);**

La macro **va\_end()** realiza las tareas auxiliares que son necesarias para que la función llamada retorne correctamente. Cuando todos los argumentos se leen, **va\_end()** reinicializa **arg\_ptr** a **NULL**.

Ejemplo:

```
#include <iostream.h>
#include <stdarg.h>
int calcular(int primero,...);
```

```
void main()
{
cout<<calcular(2,15,-1)<<endl;
cout<<calcular(6,6,6,-1)<<endl;
cout<<calcular(8,10,1946,47,-1)<<endl;
}
```

```
int calcular(int primero,...)
{
int cuenta=0,suma=0,i=primero;
va_list marcador;
va_start(marcador primero);
while (i!=-1){ suma+=i;
cuenta++;
i=va_arg(marcador,int);
}
va_end(marcador);
return suma;
}
```

## Argumentos por defecto (omisión)

Una mejora de las funciones en C++ es que se pueden especificar los valores por defecto para los argumentos cuando se proporciona un prototipo de una función. Cuando se llama a una función se pueden omitir argumentos e inicializarlos a un valor por defecto.

Un argumento por defecto u omisión es un parámetro que un llamador a una función no ha de proporcionar. Los argumentos por defecto también pueden ser parámetros opcionales. Si se pasa un valor a uno de ellos, se utiliza ese valor. Si no se pasa un valor a un parámetro opcional, se utiliza un valor por defecto como argumento.

```
void f(int ,int =2);
void main()
{
f(4,5);
f(6);
}
```

```
void f(int i, int j)
{
cout<<i<<" "<<j<<endl;
}
```

Al ejecutar el programa, se visualizará: **4,5 6,2**

Los argumentos por defecto se pasan por valor.

Todos los argumentos por defecto debe estar situados al final del prototipo de la función. Después del primer argumento por defecto, todos los argumentos posteriores deben incluir también valores por defecto.

## Introducción

Los tipos definidos por el usuario o tipos abstractos de datos (**TAD**) empaquetan elementos dato con las operaciones que se realizan sobre esos datos. C++ soporta los **TAD** con el tipo clase que puede ser implementado con estructuras, uniones y clases.

## Abstracción de datos

Una característica importante de cualquier lenguaje de programación es la capacidad de crear tipos de datos definidos por el usuario. Aunque se pueden crear en C sus propios tipos, utilizando las palabras reservadas typedef y struct, los tipos resultantes no se pueden integrar fácilmente en el resto del programa. Además, en C, sólo se pueden definir los tipos en términos de datos; es decir, las funciones utilizadas para manipular esos tipos no son parte de la definición del tipo.

Una definición de un tipo que incluye datos y funciones y el modo para encapsular los detalles, se conoce como tipo abstracto de dato. En C++ se implementa mediante el uso de tipos de datos definidos por el usuario, llamados clases. **clase = datos + funciones**

Una diferencia importante entre C y C++, es que en C++ se pueden declarar funciones dentro de una estructura (no se requiere declarar punteros a funciones). Las estructuras pueden tener también especificadas regiones de acceso (medios en que se puede controlar el acceso a los datos).

La abstracción de datos en C++ se obtiene con los tipos de datos estructura (struct) y clase (class).

## Concepto de clase

Una clase es un tipo de dato que contiene uno o más elementos dato llamados miembros dato, y cero, una o más funciones que manipulan esos datos (llamadas funciones miembro). Una clase se puede definir con struct, union o class. La sintaxis de una clase es:

```
class nombre_clase
{
miembro1;
miembro2;
...
funcion_miembro1();
funcion_miembro2();
...
};
```

Una clase es sintácticamente igual a una estructura, con la única diferencia de que en el tipo class todos los miembros son por defecto privados mientras que en el tipo struct son por defecto públicos.

En C se utiliza el término variable estructura para referirse a una variable de tipo estructura. En C++ no se utiliza el término variable de clase, sino instancia de la clase.

El término objeto es muy importante y no es más que una variable, que a su vez no es más que una instancia de una clase. Por consiguiente una clase es:

```
class cliente
{
char nom[20];
char num;
};
```

y un objeto de esta clase se declara: **cliente cli;**

Una definición de una clase consta de dos partes: una declaración y una implementación. La declaración lista los miembros de la clase. La implementación o cuerpo define las funciones de la clase.

Una de las características fundamentales de una clase es ocultar tanta información como sea posible. Por consiguiente, es necesario imponer ciertas restricciones en el modo en que se puede manipular una clase y de cómo se pueden utilizar los datos y el código dentro de una clase.

Una clase puede contener partes públicas y partes privadas. Por defecto, todos los miembros definidos en la clase son privados. Para hacer las partes de una clase públicas (esto es, accesibles desde cualquier parte del programa) deben declararse después de la palabra reservada public. Todas las variables o funciones definidas después de public son accesibles a las restantes funciones del programa. Dado que una característica clave de la POO es la ocultación de datos, debe tenerse presente que aunque se pueden tener variables públicas, desde un punto de vista conceptual se debe tratar de limitar o eliminar su uso. En su lugar, deben hacerse todos los datos privados y controlar el acceso a ellos a través de funciones públicas.

```
class articulo
{
private:float precio;
char nombre[];
public:void indicar();
};
```

Por defecto u omisión todo lo declarado dentro de una clase es privado y sólo se puede acceder a ello con las funciones miembro declaradas en el interior de la clase o con funciones amigas.

Los miembros que se declaran en la sección protegida de una clase sólo pueden ser accedidos por funciones miembro declaradas dentro de la clase, por funciones amigas o por funciones miembro de clases derivadas.

A los miembros que se declaran en la región pública de una clase se puede acceder a través de cualquier objeto de la clase de igual modo que se accede a los miembros de una estructura en C.

```
class alfa
{
int x; //miembros dato privados
float y;
char z;
public:double k; //miembro dato público
void fijar(int,float,char); //funciones miembro públicas
void visualizar();
};

void main()
{
alfa obj; //declaración de un objeto
obj.fijar(3,2.1,'a'); //invocar a una función miembro
obj.visualizar(); //invocar a una función miembro
obj.x=4; //error: no se puede acceder a datos privados
obj.k=3.2; //válido: k está en la región pública
}
```

La definición de funciones miembro es muy similar a la definición ordinaria de función. Tienen una cabecera y un cuerpo y pueden tener tipos y argumentos. Sin embargo, tienen dos características especiales:

a) Cuando se define una función miembro se utiliza el operador de resolución de ámbito (::) para identificar la clase a la que pertenece la función.

b) Las funciones miembro (métodos) de las clases pueden acceder a las componentes privadas de la clase.

Opción 1

```
class ejemplo
{
int x,y;
public:
void f()
{
cout<<"x= "<<x<<" y= "<<y<<endl;
}
};
```

Opción 2

```
class ejemplo
{
int x,y;
public: void f();
};
void ejemplo::f()
```

```

{
cout<<"x= "<<x<<" y= "<<y<<endl;
}

```

En la primera opción la función está en línea (inline). Por cada llamada a esta función, el compilador genera (vuelve a copiar) las diferentes instrucciones de la función. En la segunda opción (la más deseable) la función f se llamará con una llamada verdadera de función.

La declaración anterior significa que la función f es miembro de la clase ejemplo. El nombre de la clase a la cual está asociada la función miembro se añade como prefijo al nombre de la función. El operador :: separa el nombre de la clase del nombre de la función. Diferentes clases pueden tener funciones del mismo nombre y la sintaxis indica la clase asociada con la definición de la función.

## Objetos

En C++ un objeto es un elemento declarado de un tipo clase. Se conoce también como una instancia de una clase.

Los objetos se pueden tratar como cualquier variable C. La principal diferencia es que se puede llamar a cualquiera de las funciones que pertenecen a un objeto, esto es, se puede enviar un mensaje a ella.

```

class rectangulo
{
int base,altura;
public:
void dimensiones(int,int);
int area();
};

void rectangulo::dimensiones(int b,int h)
{
base=b;
altura=h;
}
int rectangulo::area()
{
return base*altura;
}
void main()
{rectangulo r; //declarar el objeto
r.dimensiones(3,5); //definir el tamaño
cout<<"area "<<r.area();
}

```

## Acceso a los miembros de una clase

A los miembros de una clase se accede de igual forma que a los miembros de una estructura. Existen dos métodos para acceder a un miembro de una clase: el operador punto (.) y el operador flecha (->) que actúan de modo similar.

```

class contador
{
public:leer() {return 1;}
};
void main()
{
contador c;
contador *p=new(contador);
}

```

```
leer()); //error: función desconocida, no en ámbito
cout<<c.leer(); //correcto
cout<<p->leer(); //correcto
}
```

### **Clases vacías**

Aunque el propósito de una clase es encapsular código y datos, una clase puede tener también una declaración vacía. `class vacia{}`;

Naturalmente, no se pueden realizar grandes cosas con esta clase, pero se pueden crear objetos de ella: `vacía obj;`

Con frecuencia, en el desarrollo de un proyecto grande se necesitan comprobar implementaciones de primeras versiones en las que algunas clases no están totalmente identificadas o implementadas todavía. Se suelen denominar resguardos (stubs) y se diseñan para obtener códigos que se compilen sin errores, permitiendo comprobar alguna parte de ellos.

### **Clases anidadas**

La potencia de abstracción de una clase se puede incrementar incluyendo otras declaraciones de clase. Una clase declarada en el interior de otra se denomina clase anidada, y se puede considerar como una clase miembro.

El identificador de una clase anidada está sujeto a las mismas reglas de acceso que los restantes miembros. Si una clase anidada se declara en la sección `private` de la clase circundante, la clase anidada será utilizable sólo por los miembros datos de la clase que la circunde. La clase que encierra puede acceder al nombre de la clase anidada sin resolución de ámbito. Si un nombre de una clase anidada es accesible a una clase o función que no la circunda, se debe aplicar el operador `::`.

```
class externa
{
public:
class interna
{
public:int x;
};
};
```

```
void main()
{
externa::interna valor;
int v=valor.x;
}
```

### **Los miembros dato**

La lista de miembros de una clase puede comprender cualquier tipo válido en C++. Puede contener tipos primarios, estructuras e incluso punteros a cualquier tipo válido. Los miembros dato pueden ser incluso clases. En cualquier caso sólo las instancias de clases definidas o declaradas previamente pueden ser miembros.

Los miembros dato declarados en la clase se deben considerar equivalentes a campos de una estructura, no a variables. Tal como las estructuras, se debe declarar un objeto de un tipo clase y a continuación se inicializan sus miembros dato.

Un miembro de una clase se puede declarar estático (static). Para un miembro dato, la designación static significa que existe sólo una instancia de ese miembro. Un miembro dato estático es compartido por todos los objetos de una clase.

A un miembro dato static se le asigna una zona fija de almacenamiento en tiempo de compilación, al igual que una variable global, pero el identificador de la variable está dentro de ámbito utilizando solamente el operador :: con el nombre de la clase.

Los miembros datos se asignan generalmente con la misma clase de almacenamiento. Para declarar o inicializar un miembro static se utiliza la misma notación que una variable global.

```
class ejemplo
{
public:
static int valor; //declarar miembro estático
};
int ejemplo::valor; //definir miembro estático
void main()
{
ejemplo e1,e2;
e1.valor=1;
e2.valor=10;
}
```

A los miembros dato static se puede acceder:

1. Utilizando el operador punto
2. Utilizando el operador flecha, si el lado izquierdo es un puntero a objeto
3. Utilizando el identificador de la clase sin referenciar un objeto real: ejemplo::valor=3;

Los miembros datos static no siempre tienen que ser public.

```
class ejemplo
{
private:static int valor; //declarar miembro estático
};
int ejemplo::valor=5; //definir miembro estático
void main()
{
ejemplo e1;
e1.valor=1; //error: acceso no válido
}
```

Para acceder a un miembro dato private static se necesita utilizar el operador ::. Otros medios son:

- 1.- A través de una función miembro de la clase
- 2.- A través de una función declarada amiga de la clase

### **Ámbito de una clase**

Una clase actúa como cualquier otro tipo de dato con respecto al ámbito. Todos los miembros de una clase se dice que están en el ámbito de esa clase; cualquier miembro de una clase puede referenciar a cualquier otro miembro de la misma clase.

Las funciones miembro de una clase tienen acceso no restringido a los miembros dato de esa clase. El acceso a los miembros dato y funciones de una clase fuera del ámbito de la clase está controlado por el programador. La idea es encapsular la estructura de datos y funcionalidad de una clase, de modo que el acceso a la estructura de datos de la clase desde fuera de las funciones miembro de la clase, sea limitada o innecesaria.

El nombre de la clase tiene que ser único dentro de su ámbito.

### **Especificadores de acceso a los miembros de una clase**

En una definición de clase, un especificador de acceso se utiliza para controlar la visibilidad de los miembros de una clase fuera del ámbito de la clase.

Los miembros de una clase pueden ser públicos, privados o protegidos. Las palabras reservadas **public**, **private** y **protected** se utilizan para controlar el modo de acceso a la clase.

Dentro de una declaración de clase, cada una de estas palabras se puede utilizar para preceder a una o más declaraciones de los miembros de una clase:

- **Acceso protegido**. Los miembros protegidos significan que sólo se puede acceder a ellos por funciones miembro dentro de la misma clase y por funciones miembro de clases derivadas de esta clase.
- **Acceso público**. Los miembros públicos son accesibles por cualquier parte del programa.
- **Acceso privado**. Los miembros privados sólo pueden ser utilizados por las funciones miembro de la clase y las funciones amigas de la clase.

### **Funciones miembro**

Las funciones miembro son miembros de una clase y son funciones diseñadas para implementar las operaciones permitidas sobre los tipos de datos de una clase. Para declarar una función miembro hay que situar su prototipo en el cuerpo de la clase. No hay que definir la función dentro de la clase; dicha definición puede estar fuera de la clase e incluso en un archivo independiente, aunque también pueden ser definidas en línea dentro de la clase.

Las funciones miembro son necesarias para acceder a los datos privados. En general, son públicas; si no lo fueran, ninguna otra función podría llamarlas. Se pueden declarar para devolver valores con tipos incluyendo objetos de clases, punteros o referencias. Pueden ser declaradas también para aceptar cualquier número y tipo de argumentos. Los argumentos por defecto están permitidos, así como la notación de puntos suspensivos.

### **El puntero this**

Nunca se puede llamar a una función miembro privada de una clase a menos que se asocie con un objeto. En **C** cada vez que se utiliza un puntero para acceder a los miembros de una estructura, debe utilizarse el operador de puntero (->) para acceder a los datos. ¿Cómo sabe una función miembro cuál es la instancia de una clase asociada con ella?

El método utilizado por **C++** es añadir un argumento extra oculto a las funciones miembro. Este argumento es un puntero al objeto de la clase que lo enlaza con la función asociada y recibe un nombre especial denominado **this**.

Dentro de una función miembro, **this** apunta al objeto asociado con la invocación de la función miembro. El tipo de **this** en una función miembro de una clase **T** es **T\*const** es decir, un puntero constante a un objeto de tipo **T**.

Normalmente, el programador no necesita preocuparse por este puntero, ya que **C++** realiza la operación automáticamente, haciendo el uso de **this** transparente a las funciones miembro que la utilizan. Dentro de una función miembro, se pueden hacer referencias a los miembros del objeto asociado a ella con el prefijo **this** y el operador de acceso **->**. Sin embargo, este proceso no es necesario, ya que es redundante. Consideremos como ejemplo el constructor de una clase que manipula números complejos:

```

complejo::complejo (float a,float b)
{
real=a;
imag=b;
}

```

Este constructor se puede escribir:

```

complejo::complejo (float a,float b)
{
this->real=a;
this->imag=b;
}
this->nombre_miembro : apunta a un miembro
*this : es el objeto total. Es un valor constante
this : es la dirección del objeto apuntado

```

### Funciones miembro estáticas

Las funciones miembro **static** sólo pueden acceder a otras funciones y datos declarados en una clase, pero no pueden manipular funciones ni datos no estáticos. La razón de esta característica es que una función miembro **static** no tiene asignado un puntero **this** y por ello no puede acceder a miembros dato de la clase a menos que se pase explícitamente este puntero **this**.

### Constructores

Un constructor es una función especial que sirve para construir o inicializar objetos. En C++ la inicialización de objetos no se puede realizar en el momento en que son declarados; sin embargo, tiene una característica muy importante y es disponer de una función llamada constructor que permite inicializar objetos en el momento en que se crean.

Un constructor es una función que sirve para construir un nuevo objeto y asignar valores a sus miembros dato. Se caracteriza por:

- Tener el mismo nombre de la clase que inicializa
- Puede definirse **inline** o fuera de la declaración de la clase
- No devuelve valores
- Puede admitir parámetros como cualquier otra función
- Puede existir más de un constructor, e incluso no existir

Si no se define ningún constructor de una clase, el compilador generará un constructor por defecto. El constructor por defecto no tiene argumentos y simplemente sitúa ceros en cada byte de las variables instancia de un objeto. Si se definen constructores para una clase, el constructor por defecto no se genera.

Un constructor del objeto se llama cuando se crea el objeto implícitamente: nunca se llama explícitamente a las funciones constructoras. Esto significa que se llama cuando se ejecuta la declaración del objeto. También, para objetos locales, el constructor se llama cada vez que la declaración del objeto se encuentra. En objetos globales, el constructor se llama cuando se arranca el programa.

El constructor por defecto es un constructor que no acepta argumentos. Se llama cuando se define una instancia pero no se especifica un valor inicial.

Se pueden declarar en una clase constructores múltiples, mientras tomen parte diferentes tipos o número de argumentos. El compilador es entonces capaz de determinar automáticamente a qué constructor llamar en cada caso, examinando los argumentos.

Los argumentos por defecto se pueden especificar en la declaración del constructor. Los miembros dato se inicializarán a esos valores por defecto, si ningún otro se especifica.

C++ ofrece un mecanismo alternativo para pasar valores de parámetros a miembros dato. Este mecanismo consiste en inicializar miembros dato con parámetros.

```
class prueba
{
tipo1 d1;
tipo2 d2;
tipo3 d3;
public:
prueba(tipo1 p1, tipo2 p2, tipo3 p3):d1(p1),d2(p2),d3(p3)
{ }
};
```

Un constructor que crea un nuevo objeto a partir de uno existente se llama constructor copiadore o de copias. El constructor de copias tiene sólo un argumento: una referencia constante a un objeto de la misma clase. Un constructor copiadore de una clase complejo es:

```
complejo::complejo(const complejo &fuente)
{
real=fuente.real;imag=fuente.imag;
}
```

Si no se incluye un constructor de copia, el compilador creará un constructor de copia por defecto. Este sistema funciona de un modo perfectamente satisfactorio en la mayoría de los casos, aunque en ocasiones puede producir dificultades. El constructor de copia por defecto inicializa cada elemento de datos del objeto a la izquierda del operador = al valor del elemento dato equivalente del objeto de la derecha del operador =. Cuando no hay punteros involucrados, eso funciona bien. Sin embargo, cuando se utilizan punteros, el constructor de copia por defecto inicializará el valor de un elemento puntero de la izquierda del operador = al del elemento equivalente de la derecha del operador; es decir que los dos punteros apuntan en la misma dirección. Si ésta no es la situación que se desea, hay que escribir un constructor de copia.

## **Destructores**

Un destructor es una función miembro con igual nombre que la clase, pero precedido por el carácter ~. Una clase sólo tiene una función destructor que, no tiene argumentos y no devuelve ningún tipo. Un destructor realiza la operación opuesta de un constructor, limpiando el almacenamiento asignado a los objetos cuando se crean. C++ permite sólo un destructor por clase. El compilador llama automáticamente a un destructor del objeto cuando el objeto sale fuera del ámbito. Si un destructor no se define en una clase, se creará por defecto un destructor que no hace nada.

Normalmente los destructores se declaran public.

## **Creación y supresión dinámica de objetos**

Los operadores new y delete se pueden utilizar para crear y destruir objetos de una clase, así como dentro de funciones constructoras y destructoras.

Un objeto de una determinada clase se crea cuando la ejecución del programa entra en el ámbito en que está definida y se destruye cuando se llega al final del ámbito. Esto es válido tanto para objetos globales como locales, ya que los objetos globales se crean al comenzar el programa y se destruyen al salir de él. Sin embargo, se puede crear un objeto también mediante el operador new, que asigna la

memoria necesaria para alojar el objeto y devuelve su dirección, en forma de puntero, al objeto en cuestión.

Los constructores normalmente implican la aplicación de new.  
p =new int(9) //p es un puntero a int inicializado a 9  
cadena cad1("hola");  
cadena \*cad2=new cadena;

Un objeto creado con new no tiene ámbito, es decir, no se destruye automáticamente al salir fuera del ámbito, sino que existe hasta que se destruye explícitamente mediante el operador delete.

```
class cadena
{
char *datos;
public:cadena(int);
~cadena();
};
cadena::cadena(int lon)
{
datos=new char[lon];
}
cadena::~~cadena()
{
delete datos;
}
```

### **Funciones amigas**

Una función amiga es una función no miembro de una clase que puede tener acceso a las partes privadas de una clase; se debe declarar como amiga de la clase mediante la palabra reservada friend.

Las funciones amigas se declaran situando su prototipo de función en la clase de la que son amiga precediéndola con la palabra reservada friend. Por ejemplo:

```
class cosa
{
int datos;
public:
friend void cargar (cosa& t, int x);
};
void cargar(cosa& t, int x)
{
t.datos=x;
}
```

Como la función cargar se declara amiga de la clase cosa puede acceder al miembro privado datos.

Las razones fundamentales para utilizar funciones amigas es que algunas funciones necesitan acceso privilegiado a más de una clase. Una segunda razón es que las funciones amigas pasan todos sus argumentos a través de la lista de argumentos y cada valor de argumento se somete a la conversión de asignación.

### **Clases amigas**

No sólo puede ser una función, amiga de una clase, también una clase completa puede ser amiga de otra clase. En este caso todas las funciones de la clase amiga pueden acceder a las partes privadas de la otra clase.

Una clase amiga puede ser declarada antes de que pueda ser designada como amiga.

```
class animales;
class hombre
{
public:
friend class animales;
};
class animales
{//..
};
```

Cada función miembro de animales es amiga de la clase hombre. La designación de un amigo puede estar en una sección private o public de una clase.

## **Sobrecarga de funciones y operadores**

C++ proporciona las herramientas necesarias que permiten definir funciones y operadores que utilizan el mismo nombre o símbolo que una función u operador incorporado. Estas funciones y operadores se dicen que están sobrecargados y se pueden utilizar para redefinir una función u operador definido.

### **Sobrecarga de funciones**

En C++ dos o más funciones pueden tener el mismo nombre representando cada una de ellas un código diferente. Esta característica se conoce como sobrecarga de funciones.

Una función sobrecargada es una función que tiene más de una definición.

Estas funciones se diferencian en el número y tipo de argumentos, pero también hay funciones sobrecargadas que devuelven tipos distintos.

Sobrecarga se refiere al uso de un mismo nombre para múltiples significados de un operador o una función.

Dado el énfasis del concepto de clase, el uso principal de la sobrecarga es con las funciones miembro de una clase. Cuando más de una función miembro con igual nombre se declara en una clase, se dice que el nombre de la función está sobrecargado en esa clase. El ámbito del nombre sobrecargado es el de la clase.

La sobrecarga de funciones amigas es similar a la de funciones miembro, con la única diferencia de que es necesaria la palabra reservada friend al igual que en la declaración de cualquier función amiga.

### **Sobrecarga de operadores**

De modo análogo a la sobrecarga de funciones, la sobrecarga de operadores permite al programador dar nuevos significados a los símbolos de los operadores existentes en C++.

C++ permite a los programadores sobrecargar a los operadores para tipos abstractos de datos.

Operadores que se pueden sobrecargar: +, -, \*, /, %, ^, &, |, ~, ', =, <, >, <=, >=, ++, --, <<, >>, ==, 0, %%, ||, +=, -=, \*=, /=, %=, &=, |=, <<=, >>=, [ ], ( ), ->, ->\*, new, delete

Si un operador tiene formatos unitarios y binarios (tales como + y &) ambos formatos pueden ser sobrecargados. Un operador unitario tiene un parámetro, mientras que un operador binario tiene dos. El único operador ternario, ?:, no se puede sobrecargar.

Existen una serie de operadores que no se pueden sobrecargar: ., ::, ?:, sizeof

La sobrecarga de operadores en C++ tiene una serie de restricciones que es necesario tener presente a la hora de manejar operadores sobrecargados:

- \* Se pueden sobrecargar sólo los operadores definidos en C++
- \* La sobrecarga de operadores funciona sólo cuando se aplica a objetos de una clase
- \* No se puede cambiar la preferencia o asociatividad de los operadores en C++
- \* No se puede cambiar un operador binario para funcionar con un único objeto
- \* No se puede cambiar un operador unitario para funcionar con dos objetos
- \* No se puede sobrecargar un operador que funcione exclusivamente con punteros

La clave para la sobrecarga de un operador es una función incorporada a C++ que permite al programador sustituir una función definida por el usuario para uno de los operadores existentes.

Para sobrecargar un operador, se debe definir lo que significa la operación relativa a la clase a la cual se aplica. Para hacer esta operación, se crea una función operador, que define su acción. El formato general de una función operador es el siguiente: **tipo nombre\_clase::operator op (lista\_argumentos) {...}**

tipo es el tipo del valor devuelto por la operación especificada. Los operadores sobrecargados, con frecuencia tienen un valor de retorno que es del mismo tipo que la clase para la cual el operador se sobrecarga.

Las funciones operador deben ser miembro o amigas de la clase que se está utilizando.

Las funciones operador tienen la misma sintaxis que cualquier otra, excepto que el nombre de la función es operator op, donde op es el operador que se sobrecarga.

### **Declaración de funciones operador**

Las funciones operador u operadores sobrecargados pueden ser o no miembros de una clase. De este modo, se pueden sobrecargar funciones miembro de una clase, así como funciones amigas.

Una función amiga tiene un argumento para un operador unitario y dos para uno binario. Una función miembro tiene cero argumentos para un operador unitario y uno para un operador binario.

### **Sobrecarga de operadores unitarios**

Consideramos una clase t y un objeto x de tipo t y definimos un operador unitario sobrecargado: ++, que se interpreta como: **operator++(x)** o bien como **x.operator()**

Ejemplo:

```
class vector
{
double x,y;
public:
void iniciar(double a, double b){x=a; y=b;}
```

```

void visualizar()
{
cout<<"vector "<<x<<" "<<y<<endl;
}
double operator++(){x++;y++;}
};
void main()
{
vector v;
v.iniciar(2.5,7.1);
v++;
v.visualizar(); // visualiza 3.5 8.1
}

```

### Versiones prefija y postfija de los operadores ++ y --

La versión prefija del operador de incremento se sobrecarga definiendo una versión de ++ de un parámetro; la versión postfija se sobrecarga definiendo una versión de ++ de dos parámetros, siendo el segundo de tipo int (será un parámetro mudo).

Sobrecargar un operador unitario como función miembro.

```

class c
{ int x;
public:
c() {x=0;}
c(int a) {x=a;}
c& operator--() {--x;return *this;}
void visualizar() {cout<<x<<endl;}
};
void main()
{ c ejemplo(6);
--ejemplo; //ejemplo.operator--();
}

```

La función --() está declarada; ya que es una función miembro, el sistema pasa el puntero this implícitamente. Por consiguiente, el objeto que llama a la función miembro se convierte en el operando de este operador.

Sobrecarga de un operador unitario como una función amiga.

```

class c
{ int x;
public:
c() {x=0;}
c(int a) {x=a;}
friend c& operator--(c y) {--y.x;return y;}
void visualizar() {cout<<x<<endl;}
};
void main()
{ c ejemplo(6);
--ejemplo; //operator--(ejemplo);
}

```

La función --() está definida; ya que es una función amiga, el sistema no pasa el puntero this implícitamente. Por consiguiente, se debe pasar explícitamente el objeto.

## Sobrecarga de operadores binarios

Los operadores binarios se pueden sobrecargar pasando a la función dos argumentos. El primer argumento es el operando izquierdo del operador sobrecargado y el segundo argumento es el operando derecho. Suponiendo dos objetos x e y de una clase c, se define un operador binario + sobrecargado. Entonces x + y se puede interpretar como operator+(x,y) o como x.operator+(y)

Un operador binario puede, por consiguiente, ser definido:

- como un amigo de dos argumentos
- como una función miembro de un argumento (caso más frecuente)
- nunca los dos a la vez

Sobrecarga de un operador binario como función miembro

El siguiente ejemplo muestra cómo sobrecargar un operador binario como una función miembro:

```
class binario
{
int x;
public:binario() {x=0;}
binario(int a) {x=a;}
binario operator + (binario &);
void visualizar() {cout<<x<<endl;
};
binario binario::operator +(binario &a)
{
binario aux;
aux.x=x+a.x;
return aux;
}
void main()
{
binario primero(2),segundo(4),tercero;
tercero = primero + segundo;
tercero.visualizar();
}
```

La salida del programas es **6**.

Sobrecarga de un operador binario como una función amiga

```
class binario
{
int x;
public:
binario() {x=0;}
binario(int a) {x=a;}
friend binario operator + (binario &,binario &);
void visualizar() {cout<<x<<endl;
};
binario binario::operator +(binario &a,binario &b)
{ binario aux;
aux.x=a.x+b.x;
return aux; }
void main()
{ binario primero(2),segundo(4),tercero;
tercero = primero + segundo;
tercero.visualizar();
}
```

La salida del programa será **6**.

La función operador binario `+` está declarada; debido a que es una función amiga, el sistema no pasa el puntero `this` implícitamente y, por consiguiente, se debe pasar el objeto binario explícitamente con ambos argumentos. Como consecuencia, el primer argumento de la función miembro se convierte en el operando izquierdo de este operador y el segundo argumento se pasa como operando derecho.

### **Sobrecargando el operador de llamada a funciones ( )**

Uno de los operadores más usuales es el operador de llamada a función y puede ser sobrecargado. La llamada a función se considera como un operador binario **expresión principal (lista de expresiones)** donde expresión principal es un operando y lista de expresiones (que puede ser vacía) es el otro operando.

La función operador correspondiente es `operator()` y puede ser definida por el usuario para una clase sólo mediante una función miembro no estática.

`x(i)` equivale a `x.operator()` (`i`)

`x(i,j)` equivale a `x.operator()(x,y)`

### **Sobrecargando el operador subíndice [ ]**

El operador `[]` se utiliza normalmente como índice de arrays. En realidad este operador realiza una función útil; ocultar la aritmética de punteros. Por ejemplo, si se tiene el siguiente array: `char nombre[30]`; y se ejecuta una sentencia tal como `car = nombre[15]`; el operador `[]` dirige la sentencia de asignación para sumar 15 a la dirección base del array `nombre` para localizar los datos almacenados en esta posición de memoria.

En C++ se puede sobrecargar este operador y proporciona muchas extensiones útiles al concepto de subíndices de arrays. `[]` se considera un operador binario porque tiene dos argumentos. En el ejemplo `p=x[i]`

Los argumentos son `x` e `i`. La función operador correspondiente es `operator []`; ésta puede ser definida para una clase `x` sólo mediante un función miembro. La expresión `x[i]` donde `x` es un objeto de una determinada clase, se interpreta como `x.operator[](y)`.

### **Sobrecarga de operadores de flujo**

Las sentencias de flujos se utilizan para entradas y salidas. Los flujos no son parte de C++, pero se implementan como clases en la biblioteca de C++. Las declaraciones para estas clases se almacenan en el archivo de cabecera `iostream.h`. En C++ es posible sobrecargar los operadores de flujo de entrada y salida de modo que pueda manipular cualquier sentencia de flujo que incluya cualquier tipo de clase. Como se definen en el archivo de cabecera `iostream.h`, estos operadores trabajan con todos los tipos predefinidos, tales como `int`, `long`, `double`, `char`. Sobrecargando estos los operadores de flujo de entrada y salida, estos pueden además manipular cualquier tipo de objetos de clases.

Ejemplo de sobrecarga de los flujos `<<` y `>>`

```
class punto
{
int x,y;
public:punto() {x=y=0;}
punto (int xx,int yy) { x=xx;y=yy;}
void fijarx(int xx) { x=xx;}
void fijary(int yy) {y=yy;}
int leerx() {return x;}
int leery() {return y;}
```

```

friend ostream& operator << (ostream& os,const punto &p);
friend istream& operator >> (istream& is,const punto &p);
};
void main()
{
punto p;
cout<<p<<endl;
p.fijarx(50);
p.fijary(100);
cout<<p<<endl;
cout <<"introducir los valores de x e y\n";
cin>>p;
cout<<"se ha introducido"<<p;
}
ostream& operator<<(ostream &os,punto &p)
{
os <<"x= " <<p.x<<" ,y= " <<p.y;
return os;
}
istream& operator>>(istream &is,punto &p)
{
is >>p.x>>p.y;
return is;
}

```

Es posible poner en cascada múltiples objetos en una sentencia de flujo de salida:

```

punto p1,p2,p3;
cout<<p1<<":"<<p2<<":"<<p3;

```