C++ y Programación Orientada a Objetos (POO)

C++ es un potente lenguaje de programación que apareció en 1980, continuando con las ventajas, flexibilidad y eficacia del C. Es un lenguaje de programación que permite programar desde sistemas operativos, compiladores, aplicaciones de bases de datos, procesadores de texto, juegos, etc.
C++ ha ido evolucionando y ha contribuido con la aparición de Java y C#, simplificando instrucciones de C++ y añadiendo nuevas funcionalidades para realizar aplicaciones utilizables desde Internet.

Los programas **estructurados** se basan en estructuras de control bien definidas, bloques de código, subrutinas independientes que soportan recursividad y variables locales. La esencia de la programación estructurada es la reducción de un programa a sus elementos constituidos.

La programación orientada a objetos (POO), permite descomponer un problema en subgrupos relacionados. Cada subgrupo pasa a ser un objeto auto contenido que contiene sus propias instrucciones y datos que le relacionan con ese objeto. Todos los lenguajes Orientados a Objetos (OO) comparten tres características: **Encapsulación**, **Polimorfismo** y **Herencia**.

ENCAPSULACIÓN: es el mecanismo que agrupa el código y los datos que maneja. Los mantiene protegidos frente a cualquier interferencia y mal uso. Cuando el código y los datos están enlazados de esta manera se ha creado un objeto. Ese código y datos pueden ser privados para ese objeto o públicos para otras partes del programa.

POLIMORFISMO: es la cualidad que permite que un nombre se utilice para dos o más propósitos relacionados pero técnicamente diferentes. El propósito es poder usar un nombre para especificar una clase general de acciones. Por ejemplo en C tenemos tres funciones distintas para devolver el valor absoluto. Sin embargo en C++ incorpora Polimorfismo y a cada función se puede llamar abs(). El Polimorfismo se puede aplicar tanto a funciones como a operadores.

HERENCIA: proceso mediante el cual un objeto puede adquirir las propiedades de otro objeto. La información se hace manejable gracias a la clasificación jerárquica.

OBJETO: conjunto de variables y funciones pertenecientes a una clase encapsulados. A este encapsulamiento es al que se denomina objeto. Por tanto la clase es quien define las características y funcionamiento del objeto.

La base del encapsulamiento es la clase, a partir de ellas se le dan las características y comportamiento a los objetos. Lo primero es crear la clase y después en la función main que sigue siendo la principal crearemos los objetos de cada una de las clases. Las variables y funciones de una clase pueden ser **públicas**, **privadas** o **protegidas**. Por defecto, si no se indica nada, son **privadas**.

Estos modificadores nos indican en que partes de un programa podemos utilizar las funciones y variables.

private: solo tendrán acceso los de la misma clase donde estén definidos.
public: se pude hacer referencia desde cualquier parte del programa.
protected: se puede hacer referencia desde la misma clase y las subclases.

Creación de una clase:

```
class NombreClase {
	private:
	// funciones y variables privadas;
	public:
	// funciones y variables públicas;
}
```

Creación de objetos:

```
NombreClase nombreObjeto1;
NombreClase nombreObjeto2;
```

Llamadas a las funciones de una clase:

```
nombreObjeto.NombreFuncion(parámetros);
```

Desarrollo de funciones miembro:

```
ValorDevuelto NombreClase :: NombreFuncion(parametros) {
	// cuerpo;
}
```

Ejemplo: declaramos una clase "*MiClase*" con una variable privada y dos funciones públicas.

```
class MiClase
{
	private:
		int valor;
	public:
		void SetValor(int numero);
		int GetValor();
}

void MiClase :: SetValor(int numero) {
   valor = numero;
}

int miclase :: GetValor() {
   return a;
}

void main() {
   clrscr();
   MiClase obj1, obj2;
   obj1.SetValor(10);
   obj2.SetValor(99);
   printf("%d\n",obj1.GetValor()); // 10
   printf("%d\n",obj2.GetValor()); // 99
   getch();
}
```

Salida por Pantalla

En C++ se pueden seguir utilizando las mismas sentencias para mostrar información por pantalla o pedirla mediante teclado. Pero a estas antiguas se añaden 2 nuevas de la misma potencia y mayor facilidad de uso. La cabecera que utilizan estas dos sentencias es iostream.h.

Mostrar por pantalla:

```
cout << expresión;
```

Pedir por teclado:

```
cin >> variable;  // la variable pude ser de cualquier tipo.
```

Ejemplo:

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

void main()
{
   int i, j;
   double d;
   i=10;
   j=15;
   clrscr();
   cout << "Introducir valor: ";
   cin >> d;
   cout << "Estos son los valores:  ";
   cout << i << "  "<< j << "  "<< d;
   getch();
}
```

Indicadores de Formato: tres funciones miembro (width, precision y fill) que fijan formato de anchura, precisión y carácter de relleno. Es necesario fijar la anchura, precisión y carácter de relleno antes de cada sentencia de escritura.

Anchura: cout.width(ancho);
Decimales: cout.precision(nº digitos);
Relleno: cout.fill('carácter');

Funciones Constructoras y Destructoras

En los programas hay partes que requieren inicialización. Esta necesidad de inicialización es incluso más común cuando se está trabajando con objetos. Para tratar esta situación, C++ permite incluir una función constructora. A estas funciones se las llama automáticamente cada vez que se crea un objeto de esa clase.

La función constructora debe tener el mismo nombre que la clase de la que es parte, no tienen tipo devuelto, es ilegal que un constructor tenga un tipo devuelto. Pero si es posible pasarle valores a modo de parámetros.

Prototipo de la función:
```
NombreFuncion(parámetros);
```

Desarrollo de la función:
```
NombreCalse :: NombreFuncion(parámetros){
       // cuerpo;
}
```

El complemento de un constructor es la ***función destructora***. A esta función se la llama automáticamente cuando se destruye el objeto. El nombre de las *funciones destructoras* debe ser el mismo que el de la clase a la que pertenece precedido del carácter ~ (*alt*+126). Los objetos de destruyen cuando se salen de ámbito cuando son locales y al salir del programa si son globales. Las funciones destructoras no devuelven tipo y tampoco pueden recibir parámetros. Su objetivo es liberar cualquier recurso que el objeto haya solicitado.
Técnicamente un constructor y un destructor se utilizan para inicializar y destruir los objetos, pero también se pueden utilizar para realizar cualquier otra operación. Sin embargo esto se considera un estilo de programación pobre.

Prototipo de la función:
```
~NombreFuncion(parámetros);
```

Desarrollo de la función:
```
NombreCalse :: ~NombreFuncion(parámetros){
       // cuerpo;
}
```

Constructores con Parámetros

Es posible pasar argumentos a una función constructora. Para permitir esto, simplemente añada los parámetros a la declaración y definición de la función constructora. Después, cuando declare un objeto, especifique los parámetros como argumentos.

```
class MiClase {
       private:
          int valor;
       public:
          MiClase();
          MiClase(int numero);
          ~MiClase();
          void Mostrar();
};
```

```
MiClase :: MiClase() {
       cout << "Constructor Vacío";
       valor = 0;
}

MiClase :: MiClase(int numero) {
       cout << "Constructor con parámetro";
       valor = numero;
}

MiClase :: ~MiClase() {
     cout << "Destruyendo...\n";
     getch();
}

void MiClase :: Mostrar() {
      cout << "El valor es: " << valor;
}

void main() {
       MiClase objeto(4);
       objeto.Mostrar();
       getch();
}
```

Funciones Inline y Automáticas

La ventaja de las ***funciones insertadas*** es que se pueden ejecutar más rápidamente que las funciones normales. La llamada y vuelta de una función normal tardan tiempo y si tienen parámetros incluso más. Para declarar este tipo de funciones simplemente hay que preceder la definición de la función con el especificador inline.

```
inline valor_devuelto NombreFunción(parámetros) {
       // cuerpo;
}
```

Las llamadas a las funciones insertadas se realizan de la misma manera que cualquier función. Uno de los requisitos es que se tiene que definir antes de llamarla, es decir definir y desarrollar antes de la función main.
Si el compilador no es capaz de cumplir la petición, la función se compila como una función normal y la solicitud inline se ignora. Las restricciones son cuatro, no puede contener variables de tipo static, una sentencia de *bucle*, un *switch* o un *goto*.

```
inline int EsPar(int x) { return !(X%2); }

void main() {
  int a;
  cout << "Ingrese un valor entero: ";
  cin >> a;
  if (EsPar(a))
       cout << "Es par ";
  else
       cout << "Es impar";
}
```

La característica principal de las ***funciones automáticas*** es que su definición es lo suficientemente corta y puede incluirse dentro de la declaración de la clase. La palabra inline no es necesaria. Las restricciones que se aplican a las funciones inline se aplican también para este tipo. El uso más común de las funciones automáticas es para funciones constructoras.

```
class Ejemplo {
       public:
              int EsPar(int x) { return !(X%2); }
};

void main() {
  int a;
  cout << " Ingrese un valor entero: ";
  cin >> a;
  if (EsPar(a))
       cout << "Es par ";
  else
       cout << "Es impar";
}
```

Utilización de Estructuras como Clases

En C++, la definición de una estructura se ha ampliado para que pueda también incluir funciones miembro, incluyendo funciones constructoras y destructoras como una clase. De hecho, la única diferencia entre una estructura y una clase es que, por omisión, los miembros de una clase son privados y los miembros de una estructura son públicos.

```
struct Nombre {
              // variables y funciones públicas;
       private:
              // variables y funciones privadas;
};
```

Aunque las estructuras tienen las mismas capacidades que las clases, se reserva el uso de struct para objetos que no tienen funciones miembro. Una de las razones de la existencia de las estructuras es mantener compatibilidad con los programas hechos C.

Operaciones con Objetos

Asignación de Objetos: se puede asignar un objeto a otro a condición de que ambos objetos sean del mismo tipo (misma clase). Cuando un objeto se asigna a otro se hace una copia a nivel de bits de todos los miembros, es decir se copian los contenidos de todos los datos. Los objetos continúan siendo independientes.

Array de Objetos: los objetos son variables y tienen las mismas capacidades y atributos que cualquier tipo de variables, por tanto es posible disponer objetos en un array. La sintaxis es exactamente igual a la utilizada para declarar y acceder al array. También disponemos de arrays bidimensionales.

Declaración:
```
NombreClase nombreObjeto[Nro. de elementos];
NombreClase nombreObjeto[Nro. de elementos] = {elementos};
```

Inicialización:
```
nombreObjeto[índice].Función(valores);
```

Paso de Objetos a Funciones: los objetos se pueden pasar a funciones como argumentos de la misma manera que se pasan otros tipos de datos. Hay que declarar el parámetro como un tipo de clase y después usar un objeto de esa clase como argumento cuando se llama a la función. Cuando se pasa un objeto a una función se hace una copia de ese objeto.
Cuando se crea una copia de un objeto porque se usa como argumento para una función, no se llama a la función constructora. Sin embargo, cuando la copia se destruye (al salir de ámbito), se llama a la función destructora.

Prototipo de Función:
```
TipoDevuelto NombreFuncion(NombreClase nombreObjeto){
      // cuerpo;
}
```

Llamada a la Función:
```
NombreFuncion(objeto);
```

Objetos devueltos por Funciones: al igual que se pueden pasar objetos, las funciones pueden devolver objetos. Primero hay que declarar la función para que devuelva un tipo de clase. Segundo hay que devolver un objeto de ese tipo usando la sentencia return.

Cuando un objeto es devuelto por una función, se crea automáticamente un objeto temporal que guarda el valor devuelto. Este es el objeto que realmente devuelve la función. Después el objeto se destruye, esto puede causar efectos colaterales inesperados.

Punteros a Objetos: hasta ahora se ha accedido a miembros de un objeto usando el operador punto. Es posible acceder a un miembro de un objeto a través de un puntero a ese objeto. Cuando sea este el caso, se emplea el operador de flecha (**->**) en vez del operador punto. Para obtener la dirección de un objeto, se precede al objeto con el operador **&**. Se trabaja de igual forma que los punteros a otros tipos.

```
void main() {
     clrscr();
     MiClase objeto(200);
     MiClase *p;
     p = &objeto;
     cout << "El valor del Objeto es " << objeto.GetValor();
     cout << "El valor del Puntero es" << p->GetValor();
     getch();
}
```

Funciones Amigas

Habrá momentos en los que se quiera que una función tenga acceso a los miembros privados de una clase sin que esa función sea realmente un miembro de esa clase. De cara a esto están *las **funciones amigas***. Son útiles para la sobrecarga de operadores y la creación de ciertos tipos de funciones E/S.
El prototipo de esta funciones viene precedido por la palabra clave ***friend***, cuando se desarrolla la función no es necesario incluir *friend*. Una ***función amiga*** no es miembro y no se puede calificar mediante un nombre de objeto. Estas funciones no se heredan y pueden ser amigas de más de una clase.

Protitpo:
```
friend TipoDevuelto NombreFuncion(parametros);
```

Desarrollo:
```
TipoDevuelto NombreFuncion(parametros){
        // cuerpo;
}
```

This, New y Delete

This es un puntero que se pasa automáticamente a cualquier miembro cuando se invoca. Es un puntero al objeto que genera la llamada, por tanto la función recibe automáticamente un puntero al objeto. A este puntero se referencia como *this* y solo se pasa a los miembros punteros *this*.

```
void MiClase :: Mostrar() {
      cout << "El valor es: " << this->valor;
}
```

Hasta ahora si se necesitaba asignar memoria dinámica, se hacía con ***malloc*** y para liberar se utilizaba ***free***. En C++ se puede asignar memoria utilizando ***new*** y liberarse mediante ***delete***. Estos operadores no se pueden combinar unas con otras, es decir debe llamarse a ***delete*** solo con un puntero obtenido mediante ***new***. Los objetos también se les puede pasar un valor inicial con la sentencia ***new***.

Sintaxis:
```
puntero = new MiClase;
delete puntero;
puntero = new MiClase(valor_inicial);
```

También se pueden crear arrays asignados dinámicamente. La sintaxis general es:
```
puntero = new MiClase[tamaño];
```

Herencia

Para empezar, es necesario definir dos términos normalmente usados al tratar la herencia. Cuando una clase hereda otra, la clase que se hereda se llama ***clase base***. La clase que hereda se llama ***clase derivada***. La *clase base* define todas las cualidades que serán comunes a cualquier *clase derivada*. Otro punto importante es el acceso a la clase base. El acceso a la clase base pude tomar 3 valores, **public**, **private** y **protected**.

Si el acceso es **public**, todos los atributos de la clase base son públicos para la derivada.
Si el acceso es **private**, los datos son privados para la clase base la derivada no tiene acceso.
Si el acceso es **protected**, datos privados para la base y derivada tiene acceso, el resto sin acceso.

La herencia puede ser simple o múltiple. En el caso simple solo se heredan los atributos y el comportamiento de una sola clase base. En el segundo caso se puede heredar de múltiples cases padre.

```
class MiClase {
       private:
              int a;
       protected:
              int b;
       public:
              int c;
              MiClase(int n, int m) { a=n; b=m; }
              int GetA() { return a; }
              int GetB() { return b; }
};

void main() {
     MiClase objeto(10, 20);
     clrscr();
     objeto.c = 30;
     // objeto.b = 30; error, sin acceso.
     // objeto.a = 30; error, sin acceso.
     Cout << objeto.GetA() << "\n";
     cout << objeto.GetB() << "\n";
     cout << objeto.c;
     getch();
}
```

Formato de la Clase Derivada:

```
class NombreClaseDerivada:tipo_acceso NombreClaseBase {
       // cuerpo;
};
```

Ejemplo: Herencia pública.

```
class ClaseBase {
            int x;
       public:
            void SetX(int a){ x = a; }
            void MostrarX() { cout << x; }
};
```

```
class ClaseDerivada:public ClaseBase {
            int y;
       public:
            void SetY(int b) { y = b; }
            void MostrarY()  { cout << y; }
};

void main() {
     clrscr();
     ClaseDerivada obj;
     obj.SetX(10);
     obj.SetY(20);
     obj.MostrarX();
     cout << "\n";
     obj.MostrarY();
     getch();
}
```

Ejemplo: Herencia con acceso privado.

```
class ClaseBase {
            int x;
       public:
            void SetX(int a) { x = a; }
            void MostrarX()  { cout << x << "\n"; }
};
class ClaseDerivada:private ClaseBase {
            int y;
       public:
            void SetXY(int a, int b) { SetX(a); y = b; }
            void MostrarXY() { MostrarX(); cout << y << "\n"; }
};

void main() {
     clrscr();
     ClaseDerivada ob;
     ob.SetXY(10, 20);
     ob.MostrarXY();
     // ob.SetX(10); error, sin acceso.
     // ob.MostrarX(); error, sin acceso.
     getch();
}
```

Herencia Múltiple: existen dos métodos en los que una clase derivada puede heredar más de una clase base. El primero, en el que una clase derivada puede ser usada como la clase base de otra clase derivada, creándose una jerarquía de clases. El segundo, es que una clase derivada puede heredar directamente más de una clase base. En esta situación se combinan dos o más clases base para facilitar la creación de la clase derivada.

Sintaxis*:* para construir la derivada mediante varias clases base.

```
class ClaseDerivada:acceso nombreClaseBase1, NombreClaseBase2, nombreClaseBaseN {
       // cuerpo;
};
```

Sintaxis: para crear herencia múltiple de modo jerárquico.

```
class ClaseDerivada1:acceso ClaseBase {
       // cuerpo;
};
class ClaseDerivada2:acceso ClaseDerivada1 {
       // cuerpo;
};
class ClaseDerivadaN:acceso ClaseDerivada2 {
       // cuerpo;
};
```

Ejemplo: Herencia de tipo jerárquica.

```
class ClaseBaseA {
            int a;
      public:
            ClaseBaseA(int x) { a = x;}
            int GetA() { return a; }
};

class ClaseDerivadaB:public ClaseBaseA {
            int b;
      public:
            ClaseDerivadaB(int x, int y):ClaseBaseA(x) { b = y; }
            int GetB() { return b; }
};

class ClaseDerivadaC:public ClaseDerivdaB {
            int c;
      public:
            ClaseDerivadaC(int x, int y, int z):ClaseDerivdaB(x, y) { c = z;}
            void MostrarTodo() {
                cout << GetA() << " " << GetB() << " " << c;
            }
};

void main() {
     clrscr();
     ClaseDerivadaC objeto(1,2,3);
     objeto.MostrarTodo();
     cout << "\n" << objeto.GetA() << " " << objeto.GetB();
     getch();
}
```

El caso de los ***constructores*** es un poco especial. Se ejecutan en orden descendente, es decir primero se realiza el constructor de la clase base y luego el de las derivadas. En las *destructoras* ocurre en orden inverso, primero el de las derivadas y luego el de la base.

Ejemplo: Herencia Múltiple de varias clases base.

```
class B1 {
            int a;
      public:
            B1(int x) { a = x; }
            int GetA(){ return a; }
};

class B2 {
            int b;
      public:
            B2(int x) { b = x; }
            int GetB(){ return b; }
};

class C1:public B1, public B2 {
            int c;
      public:
            C1(int x, int y, int z):B1(z),B2(y) {
                C = x;
            }
            void MostrarTodo() {
                cout << GetA() << " " << GetB() << " ";
                cout << c << "\n";
            }
};

void main() {
     C1 objeto(1,2,3);
     objeto.MostrarTodo();
     getch();
}
```

Funciones Virtuales

Una función virtual es miembro de una clase que se declara dentro de una clase base y se redefine en una clase derivada. Para crear una ***función virutal*** hay que preceder a la declaración de la función la palabra clave ***virtual***. Debe tener el mismo tipo y numero de parámetros y devolver el mismo tipo.

Cada redefinición de la *función virtual* en una clase derivada expresa el funcionamiento especifico de la misma con respecto a esa clase derivada. Cuando se redefine una *función virtual* en una clase derivada NO es necesaria la palabra virtual.

```
class ClaseBase {
       public:
              int i;
              base(int x) { i = x; }
              virtual void Mostrar() { cout << i << "\n";}
};

class ClaseDerivada1:public ClaseBase {
       public:
              ClaseDerivada1(int x):base(x) {};
              void Mostrar() { cout << i*I << "\n"; }
};

class ClaseDerivada2:public ClaseBase {
       public:
              ClaseDerivada2(int x):base(x) {};
              void Mostrar() { cout << i + i; }
};

void main() {
     ClaseBase obj1(10);
     ClaseDerivada1 obj2(10);
     ClaseDerivada2 obj3(10);
     obj1.Mostrar();
     obj2.Mostrar();
     obj3.Mostrar();
}
```

Sobrecarga de Funciones y Operadores

En C++ dos o más funciones pueden compartir el mismo nombre en tanto en cuanto difiera el tipo de sus argumentos o el número de sus argumentos o ambos. Cuando comparten el mismo nombre y realizan operaciones distintas se dice que están *sobrecargadas*. Para conseguir la sobrecarga simplemente hay que declarar y definir todas las versiones requeridas.

```
// Tres posibles sobrecargas de la función Valor Absoluto
int Abs(int numero);
long Abs(long numero);
double Abs(double numero);

// Dos posibles alternativas para una función que muestra una Fecha.
void MostrarFecha(char *fecha);
void MostrarFecha(int anio, int mes, int dia);
```

También es posible y es muy común sobrecargar las funciones constructoras. Hay 3 razones por las que sobrecargar las funciones constructoras. Primero ganar flexibilidad, permitir arrays y construir copias de constructores.

Argumentos Implícitos: otra característica relacionada con la sobrecarga es la utilización de argumentos implícitos que permite dar un valor a un parámetro cuando **no se especifica** el argumento correspondiente en la llamada a la función.

Protoipo:

```
TipoDevuelto(var1=valor, var2=valor, varN=valor);

void Funcion(int a=0, int b=0) {
     cout << "a: " << a << ", b: " << b << "\n";
}
void main() {
     funcion();           // a:  0, b: 0
     funcion(10);         // a: 10, b: 0
     funcion(20,30);      // a: 20, b: 30
}
```

Operadores

Es muy similar a la sobrecarga de funciones, un operador siempre se sobrecarga con relación a una clase. Cuando se sobrecarga un operador no pierde su contenido original, gana un contenido relacionado con la clase. Para sobrecargar un operador se crea una función operadora que normalmente será una función amiga a la clase.

Prototipo:

```
TipoDevuelto NombreClase::operator operador(parametros) {
      // cuerpo;
}
```

Se puede realizar cualquier actividad al sobrecargar los operadores pero es mejor que las acciones de un operador sobrecargado se ajusten al uso normal de ese operador. La sobrecarga tiene dos restricciones, no puede cambiar la precedencia del operador y que el número de operadores no puede modificarse. También hay operadores que no pueden sobrecargarse (Ejemplo: ., ::, ¿??, .*).

Existen 3 tipos de sobrecarga de operadores. Operadores binarios, operadores lógicos-relacionales y operadores unarios. Cada uno de ellos debe tratarse de una manera específica para cada uno de ellos.

Binarios: la función solo tendrá un parámetro. Este parámetro contendrá al objeto que este en el lado derecho del operador. El objeto del lado izquierdo es el que genera la llamada a la función operadora y se pasa implícitamente a través de *this*.

```
class Operador {
            int x, y;
       public:
            Operador() { x=0; y=0; }
            Operador(int i, int j) { x=i; y=j; }
            void SetParametrosXY(int &i, int &j) { i=x; j=y; }
            Operador operator+(Operador objeto);
};

Operador Operador::operator+(Operador objeto) {
     Operador temp;
     temp.x = x + objeto.x;
     temp.y = y + objeto.y;
     return temp;
}

void main() {
     Operador obj1(10,10), obj2(5,3), obj3;
     int x, y;
     obj3 = obj1 + obj2;
     obj3.SetParametrosXY(x, y);
     cout << "Suma de obj1 mas obj2\n ";
     cout << "Valor de x: "<< x << "  Valor de y: " << y;
     getch();
}
```

Lógicos y Relacionales: cuando se sobrecargan dichos operadores no se deseará que las funciones operadoras devuelvan un objeto, en lugar de ello, devolverán un entero que indique verdadero o falso. Esto permite que los operadores se integren en expresiones lógicas y relacionales más extensas que admitan otros tipos de datos.

```
class Operador {
            int x, y;
       public:
            Operador() { x=0; y=0; }
            Operador(int i, int j) { x=i; y=j; }
            void SetParametrosXY(int &i, int &j) { i=x; j=y; }
            int operator==(Operador objeto);
};
```

```
int Operador::operator==(opera obj) {
     if(x==objeto.x && y==objeto.y)
         return 1;
     else
         return 0;
}

void main() {
     clrscr();
     Operador obj1(10,10), obj2(5,3);
     if (obj1==obj2)
         cout << "Objeto 1 y Objeto 2 son iguales";
     else
         cout << "Objeto 1 y objeto 2 son diferentes";
     getch();
}
```

Unarios: el miembro no tiene parámetro. Es el operando el que genera la llamada a la función operadora. Los operadores unarios pueden preceder o seguir a su operando, con lo que hay que tener en cuenta como se realiza la llamada.

```
class Operador {
           int x, y;
       public:
           Operador() { x=0; y=0; }
           Operador(int i, int j) { x=i; y=j; }
           void SetParametrosXY(int &i, int &j) { i=x; j=y; }
           Operador operator++();
};

Operador Operador::operator++() {
     x++;
     y++;
}

void main() {
     clrscr();
     Operador objeto(10,7);
     int x, y;
     objeto++;
     objeto.SetParametrosXY(x,y);
     cout << "Valor de x: " << x << " Valor de y: "<< y << "\n";
     getch();
}
```

Archivos

Para realizar E/S en archivos debe incluirse en el programa el archivo cabecera **fstream.h**. Un archivo se abre mediante el enlace a un flujo (*stream*). Tenemos 3 tipos de flujo: de *entrada*, de *salida* o de *entrada-salida*. Antes de abrir un fichero debe obtenerse el flujo. Los 3 flujos tienen funciones constructoras que abren el archivo automáticamente. Una vez realizadas las operaciones con los ficheros debemos cerrar el fichero mediante la función close( ).

Flujo	***Descripción***
ofstream out	De salida.
ofstream in	De entrada.
fstream io	De salida-entrada.

En C++ podemos trabajar con 3 tipos de ficheros: secuencial, binario sin formato y acceso aleatorio. Todos comparten el método de apertura, pero cada uno de ellos tiene métodos propios para ir escribiendo y leyendo.

Sintaxis:
```
flujo("nombre_fichero.extension");
```

Ejemplo: Fichero **secuencial**.

```
void main() {
     clrscr();
     ofstream out("fichero.txt");
     if (!out) {
         cout << "El archivo no puede abrirse";
         exit(1);
     }
     char cad[80];
     cout << "Escritura de cadenas. Para salir dejar en blanco\n";
     do
     {
         cout << ": ";
         gets(cad);
         out << cad << endl;
     }
     while(*cad);

     out.close();
}
```

Binarios Sin Formato: las funciones E/S son read()y write(). La función read() lee *num* bytes del flujo asociado y los coloca en la variable. La función write() escribe *num* bytes de la variable en el flujo asociado.

Prototipos:
```
in.read(variable,num_bytes);
out.write(variable,longitud_cadena);
```

Ejemplo: Fichero **binario**. **Escritura**.

```
void main() {
     ofstream out("texto.txt");
     if (!out) {
         cout << "El archivo no puede abrirse";
         exit(1);
     }
     char cad[80];
     cout << "Escritura de cadenas. Para salir dejar en blanco\n";
     do {
         cout << ": ";
         gets(cad);
         out.write(cad, strlen(cad));
     }
     while(strlen(cad));

     out.close();
}
```

Ejemplo: fichero **binario**. **Lectura**.

```
void main() {
     clrscr();
     ifstream in("texto.tex");
     if (!in) {
         cout << "El archivo no puede abrirse";
         exit(1);
     }
     char cad[80];
     cout << "Lectura de cadenas\n";
     in.read(cad, 80);
     cout << cad;
     in.close();
}
```

Aleatorios: también podemos realizar el acceso aleatorio. Las funciones que se utilizan son seekg() y seekp() para posicionarnos y las funciones get() y put() para leer y escribir en el fichero. Las funciones de posicionamiento y leer-escribir van emparejadas.

Prototipos:

```
out.seekp(posicion,lugar_de_comienzo);
out.put('char');
in.seekg(posicion,lugar_de_comienzo);
in.get(var_char);
```

Lugar	***Descripción***
ios::beg	Desde el principio.
ios::end	Desde el final.
ios::cur	Posición actual.

Ejemplo: Fichero **aleatorio**. **Escritura**.

```
void main() {
     clrscr();
     fstream out("texto1.txt", ios::in|ios::out);
     if (!out) {
         cout << "El archivo no se puede abrir";
         exit(1);
     }
     out.seekp(4, ios::beg);
     out.put('z');
     out.close();
     getch();
}
```

Excepciones

Es un mecanismo de gestión de errores incorporado. Permite gestionar y responder a los errores en tiempo de ejecución. Las excepciones están construidas a partir de tres palabras clave: **try**, **catch** y **throw**. Cualquier sentencia que provoque una excepción debe haber sido ejecutada desde un bloque try o desde una función que este dentro del bloque try.

Cualquier excepción debe ser capturada por una sentencia **catch** que sigue a la sentencia **try**, causante de la excepción. Es posible lanzar una excepción mediante la sentencia **throw.**

Sintaxis:

```
try {
      // cuerpo;
}
catch(tipo1 arg){
      // bloque catch 1;
}
catch(tipo2 arg){
      // bloque catch 2;
}
catch(tipoN arg){
      // bloque catch N;
}
```

Ejemplo:

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

void main()
{
     try{
         cout << "Dentro del bloque try\n";
         throw 10;
         cout << "Esto se ejecuta si no hay problemas";
     }
     catch(int i) {
         cout << "Capturado el error " << i;
         cout << "\n";
     }
     cout << "fin";
     getch();
}
```