PROGRAMACIÓN ORIENTADA A OBJETOS CON C++

¡o cómo aprender C++ en 1 hora!

sin que se note que realmente está escribiendo en C

Enrique Alba Torres
Andrés Rubio del Río

PROGRAMACIÓN ORIENTADA A OBJETOS (POO) CON C++

1. TÉRMINOS Y CONCEPTOS DE POO

El esquema tradicional de un programa, independientemente del lenguaje que se utilice, está compuesto por una secuencia de sentencias, más o menos agrupadas en rutinas o funciones, que van operando sobre una información contenida en campos o variables. El problema de esta estructura estriba en que ni las sentencias tienen un control de las variables con las que trabajan, ni estas variables están relacionadas en forma alguna con las sentencias que habrán de tratarlas. ¿Cómo puede la POO ayudarnos?

1.1. ¿Qué Significa POO?

La filosofía de la POO (*Object Oriented Programming, Programación Orientada a Objetos)* rompe con este esquema, dando lugar a una nueva idea, *el objeto*. El objeto es una abstracción en la que se unen sentencias y datos, de tal forma que a un objeto sólo lo van a poder tratar los métodos definidos para él, y estos métodos están preparados para trabajar específicamente con él. Este grado de compenetración evita que un método pueda tratar datos no apropiados, o bien que unos datos puedan ser tratados por un método no adecuado, ya que la llamada a cualquier método ha de ir siempre precedida del objeto sobre el que se quiere actuar, y éste sabe si ese método se ha definido o no para él. C++ es un lenguaje que contiene estos y otros conceptos de POO.

En terminología POO, cuando se quiere ejecutar un método (función) sobre un objeto, se utiliza un mensaje que se envía al objeto, de tal forma que el objeto llame al método y éste sepa qué objeto lo ha llamado.

1.2. Encapsulación

Este concepto permite tener un control de acceso selectivo tanto a los miembros como a los métodos, de tal forma que desde fuera del objeto sólo se pueda acceder a los métodos e identificadores que permita el creador del objeto.

1.3. Herencia

Permite la reutilización y la extensión del código. Permite diseñar nuevas clases a partir de otras ya existentes, pudiendose además extender sus métodos (cambiar su semántica en la clase que hereda).

- *Ej. Pilas y Colas.*

1.4. Polimorfismo

Permite tratar de forma genérica objetos de distintas clases, ahorrando así código y proporcionando simplicidad. Por tanto, trata de forma genérica objetos de distintos tipos derivados de una misma clase de objetos.

1.5. Constructores y Destructores

Un constructor es un método que se ejecuta automáticamente cuando se define o se crea un objeto, y su función es inicializar el objeto y prepararlo para realizar las operaciones necesarias.

Un destructor es un método que se ejecuta automáticamente cuando un objeto queda fuera del ámbito en el que se creó, y su función es destruir el objeto, liberando cualquier asignación dinámica de memoria que se haga.

El objeto puede tener o no definidos explícitamente el constructor y el destructor, o tener varios constructores.

2. POO CON C++

2.1. Clases

Una clase es una definición de operaciones que se define una vez en algún punto del programa, pero normalmente se define en un archivo cabecera, asignándole un nombre que se podrá utilizar más tarde para definir objetos de dicha clase. Las clases se crean usando la palabra clave ***class***, y su sintaxis es la siguiente:

```
class Nombre_de_la_Clase
{
      Definición_de_Datos;
      Prototipos_y_métodos;
};
```

Nota: *Muy importante no olvidar el punto y coma final.*

Cuando deseemos crear un objeto de una clase definida lo que haremos será lo siguiente:

```
Nombre de la Clase Nombre del Objeto;

Punto p1,p2;   // Creación de dos objetos
               // de la clase Punto
```

También se pueden definir apuntadores a objetos de la siguiente forma:

```
Nombre de la Clase *Nombre del Objeto;

Punto p1,p2;   // Creación de dos objetos
               // de la clase Punto
```

Se puede acceder a los datos y métodos de una clase de la misma forma que se accede a un campo de una estructura, es decir, con `.` o con `->`, seguido del nombre del elemento a usar.

2.2. Miembros Públicos, Protegidos y Privados

Un miembro *público* es accesible en cualquier lugar en el que exista un objeto de la clase.

Un miembro *protegido* sólo es accesible desde las clases que se hereden de la clase que lo contiene.

Un miembro *privado* sólo es accesible por los métodos de la clase a la que pertenece.

También se puede crear una clase usando la palabra clave **`struct`**, la diferencia es que con **`struct`** los miembros se hacen públicos por defecto, mientras que con **`class`** se hacen privados por defecto.

Para hacer un miembro *público, privado o protegido* usamos respectivamente:

`public`, **`private`** o **`protected`**

2.3. Definición de una Clase

El cuerpo de un método, puede o bien incluirse en la definición de una clase, en cuyo caso lo llamaremos método `inline`, o bien en un archivo fuera de la clase.

Para que ***C++*** sepa cuándo una función es simplemente una función, y cuándo es un método, en éste último caso siempre debemos poner al método como prefijo el nombre de la clase a la que pertenece seguido de `::` (operador de ámbito).

❑ Ej. Ver figuras 1 y 2.

```
class Celdilla
{
                    char Caracter, Atributo; //Miembros privados
      public:
                    void FijaCeldilla(char C, char A)
                    {
                          Caracter=C; Atributo=A;
                    }
                    void ObtenCeldilla(char &C, char &A)
                    {
                          C=Caracter; A=Atributo;
                    }
                    //&C es una referencia a la variable C
};
```

Figura 1. *Ejemplo de la definición de una clase con métodos definidos dentro de la clase.*

Esto mismo se podría haber hecho como se muestra en la *figura 2*.

2.4. Métodos `Inline`

En la definición de la clase anterior pueden verse dos variantes, en la primera los métodos se definen dentro de la misma clase, mientras que en la segunda sólo se incluye en la clase un prototipo de los métodos, su implementación está fuera. Los métodos cuya definición se incluye en la clase se denominan Métodos `inline`.

En el caso de emplear métodos `inline`, vamos a tener como beneficio que la ejecución va a ser más rápida, pero sin embargo, el código objeto que genera el compilador va a ser mayor.

En algunos compiladores puede que no suceda esto, pero normalmente suele ocurrir. Así, a la conclución a la que podemos llegar es que, para métodos pequeños, el *método Inline* si es óptimo, pero en caso contrario no lo será.

```
class Celdilla
{
                    char Caracter, Atributo;
      public:
                    void FijaCeldilla(char C, char A);
                    void ObtenCeldilla(char &C, char &A);
};
void Celdilla::FijaCeldilla(char C, char A)
{
      Caracter=C;Atributo=A;
}
void Celdilla::ObtenCeldilla(char &C, char &A)
{
      C=Caracter;A=Atributo;
}
```

Figura 2. *Ejemplo de la definición de una clase con métodos no Inline.*

2.5. Entrada y Salida Básica

2.5.1. `cin` y `cout`

`cin` y `cout` son dos flujos (entrada y salida) de datos pertenecientes a la biblioteca de **C++** llamada `iostream`, y se utilizan para lo siguiente:

`cin>>`*dato* *: Almacena el carácter introducido por teclado en* ***dato.***

`cout<<`*dato* : *Lee el contenido de* ***dato*** *y lo muestra por pantalla.*

3. SOBRECARGA

Consiste en la redefinición de un método. Por tanto, un *método sobrecargado* es un método que consta de varias definiciones distintas, aunque todas ellas llamadas de igual forma. Lo que va a permitir distinguir un método de otro que se llame de igual forma van a ser el tipo y el número de parámetros que recibe.

```
#include <iostream.h>
class Celdilla
{                  char Caracter, Atributo;
      public:
                   void FijaCeldilla(char C, char A)
                   {
                         Caracter=C;
                         Atributo=A;
                   }
                   void FijaCeldilla(unsigned CA)
                   {
                         Caracter=CA & 0xff;
                         Atributo=CA >> 8;
                   }
                   void ObtenCeldilla(char &C, char &A)
                   {
                         C=Caracter;
                         A=Atributo;
                   }
                   unsigned ObtenCeldilla()
                   {
                         return Atributo<<8 | Caracter;
                   }
};
void main()
{  Celdilla        X, Y;
   unsigned     Casilla;
   char            C, A;

   X.FijaCeldilla('A',112);      // Fija los valores del objeto X
   Casilla = X.ObtenCeldilla(); //Obtiene el valor de X en forma
                                // de un entero
   Y.FijaCeldilla(Casilla);     //Toma Y en dos caracteres
   cout << "Caracter= " << C << ", Atributo= " << (int) A;
}
```

Figura 3. *Ejemplo de sobrecarga de métodos de una clase.*

4. CONSTRUCTORES Y DESTRUCTORES

Cada vez que se define una variable de un tipo básico, el programa ejecuta un procedimiento que se encarga de asignar la memoria necesaria, y si es necesario, también realizará las inicializaciones pertinentes.

De forma complementaria cuando una variable queda fuera de ámbito se llama a un procedimiento que libera el espacio que estaba ocupando dicha variable.

El método al que se llama cuando creamos una variable es el ***constructor***, y al que se llama cuando se destruye una variable es el ***destructor***.

Una clase puede tener *uno o varios constructores*, pero *un sólo destructor*. El *constructor* debe tener el mismo nombre que la clase a la que pertenece, mientras que el *destructor* también debe llamarse de igual forma pero precedido por el carácter **~**.

Estos métodos no van a tener parámetros de salida, ni siquiera puede usarse ***void***. Sin embargo, el *constructor* sí podría tomar parámetros de entrada, cosa que no puede hacer el *destructor*.

```
#include <string.h>
class Cadena
{                 char           *pc;
                  unsigned  longitud;
      public:
                   Cadena( char * Str);
                   ~Cadena() { delete pc; }
            Cadena Operator+(Cadena c);
            int    Operator==(Cadena c)
            {
                  return ! strcmp(pc, c.pc);
            }
            unsigned strlen()
            {
                  return longitud;
            }
};
Cadena::Cadena(char * Str)
{     longitud= ::strlen( Str);
      pc=new char[longitud+1];
      strcpy(pc, Str);
}
Cadena Cadena::operator+(Cadena c)
{     char Resultado[ 1024 ];

      strcpy(Resultado, pc);
      strcat(Resultado, c.pc);
      Cadena Temp(Resultado);
      return Temp;
}
void main()
{     Cadena C1("Pis Pas"), C2("Pas Pis");     }
```

Figura 4. *Ejemplo de una clase con constructor y destructor.*

Al haber creado un constructor que toma un parámetro - la cadena a asignar a la cadena que se está declarando-, ya no es posible crear un objeto de clase **Cadena** sin ese parámetro. Puesto que es posible tener varios constructores, estamos usando sobrecarga. También se puede tener un constructor que por ejemplo inicialice a nulo todos los datos miembros de la clase.

```
include <string.h>
class Cadena
{                 char          *pc;
                  unsigned longitud;
      public:
                  Cadena  (){ longitud=0; pc=0;}
                  Cadena  (char * Str);
                  Cadena  (Cadena &c);
                  ~Cadena (){ if (longitud) delete pc;}
                  Cadena  operator+ (Cadena c);
                  void    operator= (Cadena c);
                  int     operator==(Cadena c)
                  {
                        return ! strcmp(pc, c.pc);
                  }
                  unsigned strlen()
                  {
                        return longitud;
                  }
};
Cadena::Cadena(char * Str)
{
      longitud= ::strlen( Str);
      pc=new char[longitud+1];
      strcpy(pc, Str);
}
Cadena Cadena::operator+(Cadena c)
{
      char Resultado[ 1024 ];

      strcpy(Resultado, pc);
      strcat(Resultado, c.pc);
      Cadena Temp(Resultado);
      return Temp;
}
Cadena Cadena::operator=(Cadena &c)
{
      longitud=c.longitud;
      pc=new char[longitud+1];
      strcpy(pc,c.pc);
}
void main()
{
      Cadena C1(“Pis Pas”), C2(“Pas Pis”);
      Cadena C3(C1), C4= C1+C2;
      C3=C2;
}
```

Figura 5. *Ejemplo de una clase con constructores de copia y asignación, y con destructor.*

4.1. Constructor de Copia y Asignación

En el ejemplo anterior, se observa que el operador de concatenación devuelve un objeto de la clase `Cadena` a fin de que ese valor de retorno se asigne a algún objeto de la forma *C3=C1+C2*. Sin embargo, no se ha definido el operador `=` para esta clase, y esto va a funcionar a pesar de ello porque por defecto el operador `=` hace una copia bit a bit. Por el contrario sí va a haber problemas cuando ocurra que el operando de destino *(C2=C1)* apunte a alguna dirección de memoria y con esta asignación se pierda. Para evitar este problema, se usa el *constructor de copia y asignación*, definiendo el método *operator=()*. Sin embargo, hay a veces un problema con este operador que nos lleva a que tengamos que sobrecargarlo.

Si se ejecuta el programa de la figura 5 paso a paso, se verá que para crear e inicializar los objetos *C3 y C4* se usa el constructor de copia, mientras que en la asignación *C3=C2* se llama al método *operator=()*. También observará que el programa llama a veces al constructor de copia sin causa aparente, esto es porque dicho constructor también se utiliza para crear objetos temporales por parte del programa, por ejemplo para pasar un objeto de una función a otra.

4.2. Constructor de Conversión

A veces es necesario usar un objeto de una determinada clase en una expresión que no está preparada para él, y habrá que realizar conversiones (*casting*).

Pueden ocurrir dos casos:

a) *Que se asigne a un objeto de nuestra clase otro que no lo es.*

b) Q*ue se asigne a un objeto de otra clase otro de la nuestra.*

- **Ej.:** Tenemos una definición del siguiente tipo: `Cadena C1;`
  `C1="Hola, mundo"`,

Esto sí es correcto, porque estamos asignando a `C1` un `char *`, que sería la cadena `"Hola, mundo"`, que será lo que reciba como parámetro el constructor de la clase `Cadena`:

```
Cadena(char *Str);
```

Sin embargo no sería correcto:

```
Cadena    C1;
Fecha     Hoy;
C1=Hoy;
```

Esto no sería correcto hasta que no se cree un *constructor* que recibiendo un objeto de la clase `Fecha` cree un objeto de la clase `Cadena`. En el otro caso, es decir, convertir un dato de nuestra clase en otro tipo, hay que sobrecargar los operadores de moldeado o *casts*.

Por ejemplo, si queremos mostrar por pantalla la cadena de caracteres que tiene un objeto, así como su longitud, no podríamos hacer `cout<<C1`, sino que tendríamos que específicar el tipo básico de datos que queremos mostrar, es decir, debemos hacer lo siguiente: `cout<<(char *) C1;` `cout<<(int) C1;`

4.3. Destructores

El *destructor*, como se mencionó antes, se distingue del resto de miembros de una clase porque va precedido por el carácter `~`. Cada vez que un objeto deja de ser útil, se llamará al destructor del objeto, que realizará sobre éste las últimas operaciones que sean necesarias. Si no se provee de un destructor se creará uno por defecto, que destruirá el objeto, sin tener en cuenta posibles liberaciones de memoria.

Cualquier destructor no puede aceptar parámetros ni de entrada ni de salida, y además sólo puede aparecer a lo sumo uno en cada clase.

4.4. Asignación Dinámica de Memoria

Se llama *Asignación Dinámica de Memoria* al proceso mediante el cual, se va reservando memoria a medida que vaya haciendo falta, y se va liberando cuando ya no sea útil. Algunos ejemplos son los vistos antes con `new y delete`.

Un objeto también se puede crear explícitamente por medio del operador `new`, que asignará la memoria necesaria para contener al objeto y devolverá un puntero al objeto en cuestión.

- Ej.: `Cadena *C=new Cadena;`

El operador `new` siempre va seguido de un parámetro que indica el tipo de objeto que se quiere crear. De este parámetro, `new` obtiene el número de bytes necesarios para almacenar el objeto, y a continuación buscará un bloque de memoria apropiado, lo asigna al programa y devuelve su dirección. En este caso, accederemos a los miembros de la clase mediante el operador `->`, y por medio de `*` accedemos al

contenido. Lo que ocurre es que un objeto creado con `new` no se destruirá automáticamente cuando quede fuera de su ámbito, sino que ha de usarse para liberar memoria el comando `delete`. Al crear un objeto con `new` detrás del nombre de la clase se le puede hacer una asignación poniéndola entre paréntesis.

4.5. Errores en Asignación de Memoria

El operador `new` devuelve la dirección de memoria reservada para el objeto que se crea, pero esto lo hará siempre y cuando no haya problemas. Si no hay memoria suficiente devolverá un valor `NULL`. Por tanto, para controlar tales tipos de problemas lo que podemos hacer es crearnos una función que los trate.

Esto puede automatizarse en *C++* usando la función `set_new_handler()` a la que se le pasará como parámetro el nombre de la función que maneja el error. Es necesario para poder usarlo incluir el fichero cabecera `new.h`.

```
void  ControlErroresdeMemoria()
{
      cout<<"Error en asignación dinámica \n";
      exit(-1);
}
void main()
{
      set_new_handler(ControlErroresdeMemoria);
      Cadena *Q=new Cadena[32000];
}
```

4.6. Sobrecarga de `New` y `Delete`

El operador `new`, así como el `delete`, se pueden sobrecargar y así particularizar su funcionamiento dentro de una clase. `new` toma como parámetro `size_t`, indicando el tamaño del objeto a crear, y devuelve un puntero de tipo `void`. `Delete` por su parte, recibe un puntero `void` y no devuelve nada.

Además de `new` y `delete` también podemos utilizar `malloc() o free()`.

5. Herencia

La herencia es una de las características fundamentales de la *Programación Orientada a Objetos* por la que, tomando como base una clase ya existente, es posible derivar una nueva, que heredará todos sus miembros, teniendo posibilidad de sobrecargarlos, crear unos nuevos o utilizarlos. La idea básica por tanto, es reutilizar las clases existentes y extender o modificar su semántica.

Para crear una clase derivada, tras el nombre de la clase, se pondrán dos puntos y el nombre de la clase que se quiere tomar como base. Además, deberemos especificar el tipo de herencia, lo cual se especificará delante del nombre de la clase base.

Por ejemplo, supongamos que a partir de la `clase Ventana`, que permite gestionar ventanas en modo texto, queremos crear una `clase Ventanagrafica` que las gestiona en modo gráfico. Para ello empleamos el siguiente código:

```
class Ventanagrafica: public Ventana
{
      .
      .
      .
      .
};
```

5.1. Herencia Pública o Privada

¿Qué estado tendrán en una clase derivada los miembros y métodos heredados de su clase base?.

Todo va a depender del cualificador que se utilice delante del nombre de la clase base: `public` o `private`.

Los miembros de la clase base que sean `private` no serán accesibles en la clase derivada en ningún momento.

Los miembros de la clase base que sean declarados `protected` serán accesibles en la clase derivada, aunque no fuera de ella, es decir, que si una clase deriva de forma privada miembros protegidos de la base, en la derivada van a ser accesibles, pero en una futura derivada de la actual derivada van a ser privados.

Si por el contrario, se derivan como `public`, en la clase derivada van a seguir siendo públicos, pero si se deriva de forma privada, serán privados.

❑ Ej. Ver Figuras 6 y 7.

BASE			**DERIVADA public**		**DERIVADA private**
Miembro	*private* *protected* *public*	Pasa a	*private* *protected* *public*	Pasa a	*private* *private* *private*

Figura 6. *Distintos tipos de herencia que se pueden realizar en C++.*

```
#include <iostream>
class Base
{
                    int Entero;     // Miembro privado por defecto
      protected:
                    void FijaEntero( int N)
                    {
                          Entero=N;
                    }
      public:
                    void ObtenEntero(void)
                    {
                          return Entero;
                    }
};
class Derivada : public Base
{
      public:
                    void ImprimeEntero(void)
                    {
                          cout<<Entero;  // Error: Entero no está
                                   // accesible aquí
                    }
                    void ActivaBase(int N)
                    {
                          FijaEntero(N);  // Correcto: acceso a
                                    // miembro protegido
                    }
                    void MuestraEntero(void)
                    {
                          cout << ObtenEntero();
                          // Correcto: acceso a miembro público
                    }
};


void main(void)
{
      Derivada A;
      A.ActivaBase(5);          // Se accede a un miembro público
      cout << A.ObtenEntero();  // Correcto, al ser un miembro
                                //  público
      A.FijaEntero(10);         //Error, FijaEntero está protegido,
}                               //y por tanto, no accesible desde
                                //fuera
```

Figura 7. *Ejemplo de clase derivada de una clase base.*

5.2. Problemas de Accesibilidad

Vamos a ver lo que ocurre cuando tenemos una clase derivada de otra, en la cual (en la derivada), hay miembros que coinciden en nombre con miembros de la clase base.

Como regla general, una clase siempre utilizará por defecto sus miembros antes que cualquier otro que se llame igual, aunque esté en la clase de donde se ha derivado.

Para hacer uso de la clase base es necesario usar el operador **::** precedido del nombre de dicha clase.

```
#include <iostream.h>
int Entero; // Variable global

class Base
{
      protected:
                  int Entero;  // Miembro protegido
};
class Derivada : public Base
{
                  int Entero;
      public:
                  void ImprimeValores(void);
};
void Derivada::ImprimeValores(void)
{
      cout << Entero;  // Imprime el Entero de la clase derivada
      cout << Base::Entero; // Imprime el Entero de la clase Base
      cout << ::Entero;  // Imprime el Entero varible global
}
```

Figura 8. *Ejemplo de la complejidad de la accesibilidad.*

5.3. Conversión de la Clase Derivada a Base

Un objeto de una clase derivada tendrá más información que uno de la clase base. Así, esto nos va a permitir hacer asignaciones de un objeto de la clase derivada a otro de la clase base, aunque no al revés, es decir, BASE=DERIVADA, pero no al revés.

❑ Ej. Ver Figura 9.

```
class Base
{
            int Entero1;
};
class Derivada : public Base
{
            int Entero2;  // Se hereda Entero1
};
void main(void)
{
      Base        A;
      Derivada    B;

      A = B;  // Ningún problema, Entero1 se asigna de B a A
      B = A; // Error, en A existe Entero1 para asignar a B, pero
             // no Entero2
}
```

Figura 9. *Ejemplo de problemas que pueden surgir con la conversión de una clase derivada a base.*

En este ejemplo, el objeto A está compuesto de un miembro -un entero-, mientras que B está compuesto de dos miembros -el entero heredado y el suyo propio-. Por tanto, B tiene información suficiente para ”llenar” el objeto A, pero no así en sentido contrario.

5.4. Herencia, Constructores y Destructores

Si se tiene una clase derivada se comprobará que parte de los miembros pertenecen a la *clase base*, y será ella quien los cree, y parte pertenecen a *la derivada*. Por tanto, en la creación de una clase derivada intervendrán el constructor de la clase base y el de la derivada. Por tanto, el constructor de la clase derivada, normalmente, llamará al de la clase base.

En el caso de que el constructor de la clase base no tenga parámetros, no va haber problemas, puesto que se llamará de forma automática, pero si por el contrario, tiene parámetros, la clase derivada tendrá que llamar al constructor de la clase base. Por tanto, el constructor de la clase derivada deberá recibir dos conjuntos de parámetros, el suyo y el de la clase base, al cual deberá de llamar con el conjunto de parámetros correspondiente.

❑ Ej. Ver Figura 10.

```
#include <iostream.h>
class Base
{
                    int Entero1;
      public:
                    Base()
                    {
                          Entero1=0;
                          cout <<(“Constructor Base \n”);
                    }
                    Base(int N)
                    {
                          Entero1=N;
                    }
};
class Derivada : public Base
{
                    // Se hereda Entero1
                    int Entero2;
      public:
                    Derivada()
                    {
                          Entero2=0;
                          cout <<(“Constructor Derivado\n”);
                    }
                    Derivada(int N1, int N2) : Base (N1)
                    {
                          Entero2=N2;
                    }
};
void main(void)
{
      Base A(5);
      Derivada B(3,6);
}
```

Figura 10. *Ejemplo de herencia en constructores.*

En cambio, con los destructores ocurre todo lo contrario, puesto que se destruirá primero el objeto de la clase heredada y después se irán llamando a los destructores de objetos de las clases bases. Además, una ventaja que tiene es que como los destructores no tienen parámetros, no es necesario llamar al **destructor** de la clase base, sino que la llamada se hará de forma automática.

❑ Ej. Ver Figura 11.

```
#include <iostream>
class Base
{
      protected:
                  int Entero1;
      public:
                  Base()
                  {
                        Entero1=0;
                        cout <<("Constructor Base\n");
                  }
                  Base(int N)
                  {
                        Entero1=N;
                  }
                  ~Base()
                  {
                        cout <<("Destructor Base\n");
                  }
};
class Derivada : public Base
{
                  int Entero2;  // Se hereda Entero1
      public:
                  Derivada()
                  {
                        Entero2=0;
                        cout <<("Constructor Derivado\n");
                  }
                  Derivada(int N1, int N2) : Base(N1)
                  {
                        Entero2=N2;
                  }
                  ~Derivada()
                  {
                        cout << "Destructor Derivado\n";
                  }
}
void main(void)
{
      Base A(5);
      Derivada B(3,6);
}
```

Figura 11. *Ejemplo de herencia en destructores.*

5.5. Herencia Múltiple

Una misma clase puede heredar de varias clases base, e incorporar por tanto los distintos miembros de éstas, uniéndolos todos en una sola clase.

Para construir una clase derivada tomando varias como base, se pondrán los nombres de cada una de las clases base separadas por coma, antecedidos, cada una de la palabra `public` o `private`. Siempre que sea posible evite usar herencia múltiple.

```
___________:private......,public......,public.....
```

6. POLIMORFISMO

Esta característica de C++ permite que un objeto tome distintas formas, gracias al enlace en tiempo de ejecución (vinculación dinámica) o ***late binding***.

6.1. Funciones Virtuales

Estas funciones nos van a ayudar a usar un método de una clase o de otra, llamado igual en las dos clases, dependiendo de la clase a la que estemos referiéndonos. Para ello existe la palabra reservada virtual, que se pone en la clase base y se antepone al prototipo. A partir de este momento, todas las funciones llamadas igual con los mismos parámetros en las clases derivadas serán virtuales, sin que sea necesario indicarlo. Esto hará necesario que en el momento de ejecución se distinga cual es la función que hay que llamar.

Esto permite que un objeto unas veces tome una forma y otras veces tome otra.

6.2. Clases Abstractas

A veces es posible que nos interese que una clase sea abstracta. Con esto lo que queremos decir es que a veces no tiene finalidad alguna crear un objeto de una clase que no tenga funcionalidad pero represente un concepto de clase útil.

7. AMISTAD

Esta característica permite que una función que no pertenezca a una clase pueda hacer uso de todos sus miembros.

7.1. Funciones Amigas

Para hacer que una función sea amiga de una clase y tenga acceso a todos sus miembros, habrá que definir un prototipo de esa función dentro de la clase, precediendo a la definición con la palabra `friend`.

❑ Ej. Ver Figura 13.

```
class  Clase
{
      int EnteroPrivado;

      public:
            void FijaEntero(int N);
            friend void FuncionAmiga(Clase &X,int N);
};
void FuncionAmiga(Clase &X,int N)
{
      X.EnteroPrivado=N;//Acceso al miembro privado
}
// Si no fuese friend EnteroPrivado no estaría accesible desde
// fuera de la clase
```

Figura 13. *Ejemplo de Función Amiga.*

Es necesario, como se ve en el ejemplo, que a la función amiga se le pase como parámetro un objeto de la clase de la cual es amiga.

7.2. Clases Amigas

De forma análoga, puede ocurrir que sea necesario que todos los miembros de una clase tengan que hacer uso de los miembros de otra. Se podrían declarar como funciones amigas todos los métodos de la primera. Sin embargo, es más fácil declarar como amiga la primera clase. Para ello es necesario que antes de declarar la clase de la cual es amiga la otra, se ponga como aviso de que hay una clase amiga.

```
class ClaseAmiga;
class Clase
{
      int EnteroPrivado;
      friend ClaseAmiga;
};
```

8. PLANTILLAS

Un problema con el que nos podemos encontrar es que tengamos dos clases que hacen uso de los mismos métodos pero estos están aplicados a distintos tipos de datos; Por ejemplo, listas con enteros, listas con cadenas, etc...

Para solucionar esto hay varias opciones:

a.- *Que la clase almacene todos sus datos como punteros a* **`void`** *realizando moldeados (castings) en las operaciones.*

b.- *Que todos los objetos que vayan a usar esa clase se creen a partir de una misma base.*

c.- *Usar plantillas.*

8.1. Plantillas de Funciones

Con esto lo que vamos a hacer es permitir al compilador ir creando las funciones apropiadas dependiendo de los parámetros utilizados.

La sintaxis es:

```
template <class X,class Y,.....>
tipo_resultado función(parámetros)
.
.
.
.
```

Con la palabra `template` estamos indicando al compilador que estamos creando una función general, (una plantilla) a partir de la cual deberá posteriormente ir creando las funciones adecuadas.

Los identificadores `X` e `Y`, representan dos clases con las que al usar la función se debe instanciar la plantilla, y pueden usarse para indicar el tipo del valor devuelto, el tipo de los parámetros que recibe la función o para declarar variables dentro de ella.

8.2. Plantillas de Clases

Lo que diferencia una plantilla de clase de una plantilla de función es que en lugar de tener la definición de una función detrás de `template` y los argumentos, vamos a tener la definición de una clase.

A la hora de implementar los métodos de la clase es obligatorio encabezar cada método de la misma forma que la clase, con la palabra `template` y los parámetros adecuados.

9. OTROS CONCEPTOS

9.1. Miembros `Static`

Los miembros y métodos de una clase pueden declararse estáticos anteponiendo la palabra `static`.

Un dato declarado estático en una clase es un dato de una sóla instancia, se crea al definir la clase y sólo existirá uno, independientemente del número de objetos que se creen. Esto es bueno cuando todos los objetos van a compartir una misma variable. Para inicializar la variable, lo que haremos será preceder a la variable del nombre de la clase.

```
class Clase
{
      static int Dato;
      .....
};

int Clase::Dato=ValorInicial;      //Inicialización de la
                                   // variable estática
```

En cuanto a los métodos declarados como estáticos, debemos decir que no han de estar relacionados con objeto alguno de dicha clase. Un método estático no puede ser nunca virtual.

9.2. El modificador `const`

Un miembro declarado con `const`, no puede sufrir modificaciones, tan sólo una inicialización en el momento en que se crea. También un miembro de una clase puede declararse constante, impidiendo que cualquier otro método que no sea el constructor y el destructor lo trate. Ni siquiera el constructor puede asignar un valor al campo directamente, sino que tendrá que llamar al constructor del miembro que se va a inicializar.

Sin embargo, si queremos que algunos métodos puedan hacer uso de estos miembros, lo que hacemos es poner detrás el modificador `const`.

❑ Ej. Ver Figura 14.

```
#include <iostream.h>
class Clase
{                 const int Objeto;   // Un miembro const privado
      public:
                  Clase(int N=0) : Objeto(N){ }// Inicializa el
                                               // objeto
            // El método Imprime es capaz de manejar objeto const
                  void Imprime(void)
                  const { const cout << Objeto;}
};
void main(void)
{     const Clase X(15);   // Un objeto const de valor inicial 15
      X.Imprime();
}
```

Figura 14. *Ejemplo del modificador* `const`.

9.3. El Objeto `this`

Permite acceder al objeto actual cuyo comportamiento se está definiendo. Es un apuntador a una clase usado dentro de cualquier método de dicha clase, por ejemplo: `this->Imprime()`.

9.4. Problemas de Ámbito

Es bastante normal crear dentro de una clase tipos enumerados o estructuras. En el caso de los tipos enumerados podemos acceder a sus elementos por medio del nombre de la clase y el operador de resolución de ámbito. Aunque también por medio de un objeto perteneciente a la clase se puede hacer, esta notación no se suele usar.

En cambio, en el caso de las estructuras el acceso a los datos siempre hay que hacerlo a través de un objeto de la clase. Si tenemos privados los elementos de una enumeración entonces sólo van a poder ser accedidos desde el interior de la clase.

```
class Clase
{                 enum Enumeracion1 {Negro, Blanco};
      public:
                  enum Enumeracion2 {Abierto, Cerrado};

                  struct {
                              char  Caracter;
                              int     Numero;
                        } Datos;
};
void main(void)
{     int        N;
      Clase Objeto;

      N=Clase::Abierto;  // Dos formas de acceder a lo mismo
      N=Objeto.Abierto;
      N=Objeto.Datos.Caracter;
}
```

Figura 15. *Ejemplo de clase con problemaas de ámbito.*

9.5. Un Truco Útil

En algunos compiladores como *Borland C++* no existen *plantillas* o `templates`, con el consiguiente problema de que hay que crearse una clase para cada tipo. Sin embargo, para solucionar este problema y no tener que hacer algo así, podemos hacer uso de los llamados **toren-pasting.**

10. CREACIÓN DE BIBLIOTECAS CON CLASES

10.1. Archivos de Cabecera

Normalmente, las definiciones de las clases están incluidas en ficheros cabecera, existiendo un archivo de cabecera por cada clase que definamos. En este archivo incluiremos la definición de la clase, cualquier constante, enumeración, etc..., que se pueda usar con un objeto de la clase y los métodos `inline`, ya que estos necesitan estar definidos antes de usarse.

```
#define Une(x,y)
                  // Macro para crear clase de manejo de matrices
#define ClaseMatriz(tipo)

class Une(Matriz, tipo)
{                 tipo *Dato;
      public:
                  Une(Matriz, tipo)(int Nelementos)
                  {
                        Dato=new tipo[NElementos];
                  }
                  Une(~Matriz, tipo)()
                  {
                        delete Dato;
                  }
                  tipo & operator[](int Elemento)
                  {
                        return Dato[Elemento];
                  }
};

ClaseMatriz(int); // Se crea una clase para manejar matrices int
ClaseMatriz(double); // Se crea una clase para manejar matrices
                     // double

#include <iostream.h>
void main(void)
{     Matrizint Objetoint(10);
      Matrizdouble Objetodouble(10);
      Objetoint[0]=10;
      Objetodouble[5]=24.5;
      cout << Objetoint[0] << '\n' << Objetodouble[5];
}
```

Figura 17. *Ejemplo de una macro.*

10.2. Realización

En cada uno de los archivos que necesiten hacer uso de una clase será necesario incluir los archivos cabecera en los cuales están definidas las clases que van a ser usadas. Sin embargo, normalmente estas bibliotecas son compiladas obteniendo ficheros `.OBJ` que serán enlazados junto a las aplicaciones.

En el caso de que tengamos varios ficheros se puede, o bien generar un código objeto por cada uno de los archivos fuente, o bien varios archivos `.OBJ`, o bien unirlos todos en una biblioteca `.LIB`.

En definitiva, el proceso de creación y utilización de una biblioteca de funciones podría simplificarse a:

1.- *Definición de las clases en los archivos cabeceras.*

2.- *Codificación de los métodos en uno o varios archivos fuentes, incluyendo archivos de cabecera.*

3.- *Compilación de cada uno de los archivos fuente, con la consiguiente generación del código objeto.*

4.- *Unión de los códigos objetos en los distintos archivos en una sóla biblioteca con cada objeto.*

5.- *Incluir los archivos de cabecera en la aplicación que use la biblioteca y enlazarla con ella.*

10.3. Enlace Seguro

Con el enlace seguro podemos dar la característica de sobrecarga a funciones implementadas en `C` que queremos usar en `C++.` Para ello, ponemos en la declaración del prototipo `extern "c"`.

Esto también se puede hacer con bloques de código y con archivos cabecera.

```
extern "c" void Represent(void);

extern"c"{
            #include "cabecera.h"
         }
```

11. *STREAMS*

Un *Stream* es un flujo de datos, y para que exista es necesario que haya un origen abierto conteniendo datos, y un destino abierto y preparado para recibir datos. Entre el origen y el destino debe haber una conexión por donde puedan fluir los datos. Estas conexiones se llevan a cabo a través de operadores y métodos de C++.

Los *streams* se usan de forma genérica, independientemente de cuales sean el origen y el destino.

11.1. Tipos de *Stream*

Los más normales son los que se establecen entre un programa y la consola: el teclado y la pantalla. También entre un programa y un fichero o entre dos zonas de memoria. En algunos casos es necesario abrir el *stream* antes de usarlo, como es el caso de los ficheros, y otras veces no, como es el caso de las consolas.

11.2. Operadores de Inserción y Extracción

Todos los *streams* van a disponer de estos dos operadores. El operador de inserción, **<<** , es el operador de desplazamiento de bits a la izquierda, que está definido en la clase `ostream` perteneciente a `iostream.h`, y **>>**, es el operador de desplazamiento a la derecha y está definido en la clase `istream`, definida en el mismo archivo cabecera.

Estos operadores van a disponer a su izquierda un objeto *stream* que será origen o destino, dependiendo del operador usado, y a la derecha lo que se va a escribir o donde se va a almacenar lo leído. Ya se han usado *streams*: `cout y cin` que tienen por defecto la salida estándar (pantalla) y la entrada estándar (teclado). También existe `cerr` para la salida estándar de errores del programa.

11.3. Entrada/Salida con Formato

La E/S de datos se realiza por medio de los operadores de inserción y extracción en *streams* y puede formatearse por ejemplo para ajustar a la derecha o a la izquierda con una longitud mínima o máxima. Para esto se definen métodos y manipuladores.

Los métodos son aquellos a los que se les llama de forma habitual (*con el nombre del objeto a la izquierda*). En cambio, los manipuladores pueden actuar en cualquier punto a través de los operadores de inserción y extracción.

11.4. Anchura

Por defecto, los datos que entran y salen tienen una anchura que se corresponde con el espacio necesario para almacenar ese dato.

Para fijar la anchura, tanto se puede usar el método `width()` como el manipulador `setw()`. Si se usa `width()` sin parámetro devolverá la anchura fijada, que por defecto es 0. Esto significa que se tomen los carácteres necesarios para la representación del dato en el *stream* de salida. Si le pasamos un parámetro se fijará la anchura a ese número de carácteres.

11.5. Carácter de Relleno

Cuando se ajusta una salida, y quedan espacios libres, estos se rellenan con espacios en blanco. Este carácter de relleno se puede modificar con el método `fill()` o el manipulador `setfill()`.

Llamando a `fill()` nos devuelve el carácter de relleno que se está usando en ese momento.

11.6. Precisión

Para números en punto flotante, a la hora de la salida se representa con todos sus dígitos, pero esto es alterable, o bien mediante el método `precision()`, o bien mediante el manipulador `setprecision()`, poniendo el número de dígitos a presentar en la salida como parámetro.

El método sin parámetros devolverá el valor fijado anteriormente.

11.7. Bases de Numeración

Normalmente, la entrada y salida de datos se hace en decimal, lo cual es modificable mediante `dec()`, `hex()`, `oct()`, y `setbase()`. Los tres primeros no necesitan parámetro alguno, pasando a base *decimal, hexadecimal, u octal*, mientras que la última necesitará recibir un entero indicando la base en la que se desea trabajar. No es necesario usar paréntesis en los tres primeros.

- Ej. Ver Figura 18.

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

void main(void)
{
      unsigned long Dato;

      cout << "Dame la dirección hexadecimal : ";
      cin  >> hex >> Dato;
      cout << hex << Dato << "," << dec << Dato << "," << oct <<
              Dato << '\n';
      cout << setbase(10);
      cin  >> setbase(10);
}
```

Figura 18. *Ejemplo de datos en distintas bases de numeración.*

11.8. Otros Manipuladores

Para el objeto `cout` existen además:

`endl()`
`ends()` ➔ sin parámetros
`flush()`

`endl():` Inserta un retorno de carro.
`ends():` Inserta un terminador de cadena.
`flush():` Fuerza la salida de todos los carácteres del *buffer* al *stream*.

Todos ellos se pueden llamar con **`cout.`**_____ o bien con **`cout<<`**_____.

`Cin` además tiene el manipulador `ws`, sin parámetros, que permite ignorar los espacios en blanco en las operaciones de entrada o lectura. Está activado por defecto.

11.9. Indicadores de Formato

Método `setf()`	Activa indicadores de formato
Método `unsetf()`	Desactiva indicadores de formato
Método `flags`	Asigna el estado de todos los indicadores

Los dos primeros equivalen a los modificadores `setioflags()` y `resetioflags()`. Todos toman un entero largo que indica los indicadores a activar o desctivar. Pertenecen a la `clase ios`, de la cual se derivan las clases a las que pertenecen los objetos que se manejan.

Los métodos `flags(), setf(), unsetf()` devuelven un entero con el estado de los indicadores antes de alterarlos, siendo interesante almacenarlos para luego restituirlos.

11.10. Otros Métodos de E/S

Los operadores de inserción y extracción permiten realizar entradas y salidas con formato. Sin embargo, a veces puede ser necesario leer o escribir información sin formato alguno. Para ello también se dispone de algunos operadores en las clases `istream y ostream`.

Métodos	***Acción***
ios::Skipws	Cuando está activo ignora los espacios en blanco iniciales
ios::left	Activa la justificación izquierda en la salida de datos
ios::right	Activa la justificación derecha en la salida de datos
ios::internal	Permite que en las salidas el signo o indicador de base, se muestre a la izquierda del relleno como primer carácter
ios::dec	Activa conversión decimal
ios::oct	Activa conversión octal
ios::hex	Activa conversión hexadecimal
ios::showbase	Mostrará el indicador de base en las salidas
ios::showpoint	Muestra el punto decimal y los dígitos decimales necesarios para completar la salida
ios::uppercase	Se usan letras mayúsculas en la salida
ios::showpos	Está desactivado por defecto y lo que hace es mostrar el signo en las salidas numéricas
ios::scientific	Realiza las salidas de punto flotante con notación científica
ios::unitbuf	Vuelca los caracteres de E/S en el dispositivo origen o destino
ios::fixed	Realiza la notación decimal

11.11. El Polifacético Método `get`

`get()` es un método para realizar lecturas o entradas de datos desde un objeto. Sus posibles formas son:

`get()`: Toma el siguiente carácter y lo devuelve. En caso de que haya encontrado final del *stream*, devuelve `EOF`.

`get(char *, int, char)`: El primer parámetro es la dirección donde se almacenará la cadena de caracteres, el segundo la longitud y el tercero el carácter delimitador, para que cuando lo encuentre se pare. Se va a detener cuando:

a) Encuentre el delimitador.

b) Encuentre el final del *stream*.

Si no se da el último carácter se tomará por defecto '\n'.

`get(char &)`: Toma un carácter del *stream* y lo almacena en la variable cuya referencia se toma como parámetro.

11.12. Lectura de Líneas

El método `getline()` toma los mismos parámetros que `get()` y el mismo valor por defecto para el delimitador. Se almacena el delimitador antes de añadir el carácter nulo.

11.13. Lectura Absoluta

Mediante el método `read()` se puede leer un número de caracteres y almacenarlo en el espacio de memoria indicado. Sólo se detiene si llega al final del *stream* y no ha leído la cantidad indicada, pero no lo hace cuando se encuentre un delimitador o el carácter nulo.

Como primer parámetro pasamos el puntero a donde se almacenará la cadena, y como segundo la longitud. Mediante el método `gcount()` es posible saber el número de bytes leídos (nos lo devuelve).

11.14. Movimiento en un *Stream*

Es posible desplazar el puntero de lectura en un *stream* mediante los métodos:

Métodos	***Parámetros y Acción***
Seekg()	Pasándole un entero largo se sitúa en una posición absoluta
ios::beg	Se sitúa en una posición relativa moviéndose desde el principio del *stream* e indicándole un desplazamiento en forma de entero largo
ios::cur	Lo mismo que antes pero parte de la posición actual
ios::end	Lo mismo que antes pero parte desde el final
tellg()	Nos devuelve la posición actual en el stream en forma de entero largo

11.15. Otras Operaciones de Lectura

Métodos	***Acción***
peek()	Método que devolverá en forma de entero el código del siguiente carácter disponible sin llegar a extraerlo
putback()	Método que toma como parámetro un carácter que se va a devolver al *stream*. Sólo se usa cuando se toma un carácter que se quiera devolver
ignore()	Método que ignora un número específico de caracteres saltándolos hasta encontrar el delimitador o hasta completar el número especificado. Toma dos parámetros: Número de caracteres a saltar y el delimitador

11.16. Escritura sin Formato

Existen dos métodos principales para la escritura sin formato que son:

`Put()`: Que toma como parámetro un carácter al que da salida por el *stream.*

`Write()`: Toma los mismos parámetros que `read()` permitiendo escribir un bloque de bytes del tamaño indicado.

También existen los métodos para desplazar el puntero de escritura en el *stream*:

```
seekp()=Seekg()
tellp()=tellg()
```

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

void main(void)
{
      char Cadena[20];
      int           N;

      cout << "Introduzca una cadena :";
      cin.getline(Cadena,20);  // Se toma una cadena de 20
                               // caracteres como máximo.
      cout << endl << endl;    // Dos líneas en blanco
      cout.write(Cadena,20); // Se da salida a los 20 caracteres
      cout.flush(); // Vuelca el stream al correspondiente
                    // dispositivo.
      cout << endl << "Introduzca un número :";
      cin >> N;
      cout << endl << "Su representación es :";
      cout.write(char *) &N, sizeof(N);  // Escribe su
                                         // representación en memoria
      cout.flush();
}
```

Figura 19. *Ejemplo del uso de Streams.*

11.17. Los *Streams* y los *Buffers*

A cada *stream* se le asocia un *buffer*, para mejorar los tiempos de acceso. Mediante el método `rdbuf()` obtenemos un puntero al *buffer* correspondiente a un *stream*. Los métodos que nos van a permitir acceder a estos *buffers* podemos dividirlos en tres grupos:

Grupo 1: Lectura del *buffer*

Métodos	***Acción***
in_avail()	Devuelve un entero indicando el número de caracteres que hay actualmente en el *buffer* de entrada, tomados desde el *stream* de lectura
sbumpc()	Lee un carácter del *buffer* de entrada, lo devuelve y avanza una posición
snextc()	Avanza una posción, lee un carácter del *buffer* y lo devuelve
sgetc()	Lee el carácter actual apuntado en el *buffer* de entrada y lo devuelve, pero no avanza
stossc()	Avanza al siguiente carácter
sgetn()	Este método necesita dos parámetros, un puntero a carácter y un entero. Lee el número de caracteres indicado del *buffer* de entrada y lo almacena en la dirección indicada por el puntero
sputbackc()	Devuelve un carácter al *buffer* de entrada, especificándoselo como parámetro

Grupo 2: Escritura en el *buffer*

Métodos	***Acción***
out_waiting()	Devuelve un entero indicando el número de caracteres que hay en el *buffer* de salida esperando a escribirse en el *stream*
sputc()	Permite escribir un carácter al *buffer* de salida, carácter que habrá que pasar como parámetro
sputn()	Permite escribir en el *buffer* de salida un determinado número de caracteres. Toma dos parámetros: Puntero a carácter y un entero

Grupo 3: <u>Posicionamiento en el *buffer*</u>

Métodos	***Acción***
seekpos()	Posiciona el puntero en una posición absoluta dada por un entero largo, el cual se pasa como parámetro. El segundo parámetro indica el puntero que se va a fijar, el de entrada o el de salida
seekoff()	Toma tres parámetros y la posición del puntero de forma relativa según el segundo parámetro que puede ser: `ios::beg`, `ios::cur`, `ios::end`, según se desee desplazar desde el principio, desde la posición actual o desde el final. El tercer parámetro es el indicador del puntero a desplazar `ios::in`, `ios::out`. Por defecto, `ios::in/ios::out`

🕮 Bibliografía

[1] ***Programación Orientada a Objetos con Borland C++***

Francisco Charte Ojeda

Anaya Multimedia, 1993

[2] ***El Lenguaje de Programación C++ (2ª Edición)***

Bjarne Strouptrup

Addison-Wesley, 1993