Metodología de la Programación II

Clases en C++

Índice

3.1 Introducción

Las clases permiten la construcción de tipos definidos por el usuario, extendiendo el conjunto de tipos elementales.

Las clases son una extensión y evolución natural de los `struct`.

Cada clase contiene:

- datos (los **datos miembro** o **campos**) y
- las funciones que los manipulan (las **funciones miembro** o los **métodos**).

Mediante los campos especificamos las *propiedades* de los objetos y mediante los métodos se modela su *comportamiento* y las acciones que pueden realizar.

Un representante de una clase es un **objeto**. *Un objeto es, a una clase, lo que una variable a un tipo.*

```
class MiClase{
   // Declaracion de datos miembro
   // Declaracion de metodos
};

MiClase objeto1, objeto2;
```

En la declaración de una clase, para cada dato miembro (campo o método), debe especificarse mediante los **modificadores de acceso** el ámbito desde el cual puede accederse a dicho miembro. Son:

- `private`

  Sólo se permite su acceso desde las funciones miembro (métodos) de la clase.

- `public`

  Se permite su acceso desde cualquier punto que pueda usar la clase. Un dato público es accesible desde cualquier objeto de la clase.

- `protected`

  Se permite su uso en los métodos de la clase y en los de las clases derivadas mediante herencia.

Cada objeto tendrá sus propios datos miembros (excepto si se declaran `static`) y sólo existe una copia de los métodos aplicable a todos los objetos de la clase.

Ejercicio: compruébese con la función `sizeof`.

Ventajas de las clases

- Ocultación de la información. Permite:
  - Que los usuarios no puedan hacer mal uso de los datos propios de un objeto, proporcionando la **integridad de los datos**, por tanto promoviendo un **software más robusto**.
  - **Separación efectiva representación-implementación**. Ocultar la implementación de las funciones, propiciando disponer de los derechos propios en la elaboración de software, y lo que es más importante, poder modificar la implementación sin que afecte al usuario.
- Posibilidad de utilizar ciertos operadores "tradicionales" (+, - +=, =, , =, ...) mediante la *sobrecarga* de los mismos.

Modificador por defecto: Si un miembro no contienen ningún modificador de acceso se considera *privado* para una clase y *público* para un `struct`.

Declaración de datos miembro:

```
class Polinomio {
   private:
      float * Coef;
      int Grado;
      int MaxGrado;

   // Declaracion de metodos
};
```

En una declaración no se asignan valores a los campos de la clase: no hay reserva de memoria. Esta tarea se reserva a los *métodos constructores*.

Declaración de métodos.

- Como cualquier función en C++.
- Se trata de especificar los *prototipos* de los métodos de la clase. En algunos casos, incluso su definición (funciones `inline`).

```
class Polinomio {
   private:
      float * Coef;
      int Grado;
      int MaxGrado;
   public:
      // Algunos métodos:
      void  PonCoeficiente (int i, float c);
```

```
      int   LeeGrado (void) const;
      float LeeCoeficiente (int i) const;
      float Evalua (float x) const;
      void  PintaPolinomio (const char * const msg) const;
}
```

Un método `const` impide la modificación de los datos miembro del objeto al que se aplican.

- Es usual disponer de funciones miembro públicas que modifican y/o leen el valor de los datos privados.
- Pueden definirse funciones privadas que pueden ser de utilidad (funciones auxiliares) para las funciones públicas.
- "Métodos *especiales*": Constructores y destructor.

Toda clase dispone, por defecto, de un constructor, de un destructor y del operador de asignación:

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

// Una clase minimalista
class Test {};
Test global;

void funcion (void){
  cout << "Ejecutando: funcion\n";
  Test local_f;
  cout << "Saliendo de: funcion\n";
}
```



```
int main (void){
  cout << "Ejecutando : main\n";
  Test local_main_1, local_main_2;

  funcion ();

  local_main_2 = local_main_1;

  cout << "Saliendo de: main\n";
  return (0);
}
```


3.2 Encapsulamiento

En un lenguaje orientado a objetos, éstos son el centro del lenguaje:

Los objetos contienen *datos* y *funciones* (*métodos*) que permiten manipular esos datos.

Para proteger esos datos de modificaciones indeseadas se *encapsulan* en objetos y se clasifican:

- *privados*: solo puede accederse a ellos por los métodos de la clase.
- *públicos*: puede accederse a ellos libremente desde fuera de la clase.

Es habitual que las clases ofrezcan un conjunto de funciones públicas a través de las cuales se puede actuar sobre los datos miembro, que serán privados.

Estas funciones forman la ***interfaz*** *de la clase.*

3.3 Constructores

- Cuando se crea un objeto de una clase *siempre* se llama *automáticamente* a un constructor.

```
Polinomio pol1;
Polinomio * pol2 = new (nothrow) Polinomio;
```

- Se emplea para iniciar los objetos de una clase.

```
Polinomio pol3 (15); // Establece MaxGrado=15
```

- Es particularmente útil para reservar, si es necesario, memoria para ciertos campos del objeto. Su liberación se realiza, en ese caso, con el *destructor*.

  Cuando el objeto haya pedido memoria dinámicamente, debe implementarse el operador de asignación (se verá más adelante).

- Pueden haber varios constructores para una clase. Es un buen ejemplo de *sobrecarga*.
- Un constructor tiene el mismo nombre que la clase en la que está declarado y no devuelve nada.

  CUIDADO: **no** es una función `void`.

¿Qué ocurre si en una clase no se define ningún constructor?

El compilador crea un ***constructor de oficio***, sin argumentos, que inicia los datos miembros a cero.

En cualquier caso:

Siempre debería implementarse, al menos, el constructor básico (sin parámetros), ***el constructor por defecto***, de manera que el programador pueda controlar y personalizar la creación e iniciación.

Importante:

El compilador sólo crea un constructor de oficio si no se ha definido ningún constructor.

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

class Test{
  private:
    string msg;
  public:
    Test (void);    // Constructor basico
    Test (string);  // Constructor con parámetros
};
Test :: Test (void) {
  msg = "VACIO";
  cout << "  Constructor 1 de Test " << "--> " << msg << "\n";
}
```



```
Test :: Test (string cad){
  msg = cad;
  cout << "  Constructor 2 de Test " << "--> " << msg << "\n";
}
Test global ("global"); // Objeto global

void funcion (void){
  cout << "Ejecutando : funcion\n";
  Test local_f ("local a funcion()");
  cout << "Saliendo de: funcion\n";
}

int main(void){
  cout << "Ejecutando : main\n";
  Test local_m ("local a main()");
  funcion ();
```



```
  local_m = global;
  Test local_vacia;

  cout << "Saliendo de: main\n";
  return (0);
}
```

Su ejecución produce:

```
  Constructor 2 de Test --> global
Ejecutando : main
  Constructor 2 de Test --> local a main()
Ejecutando : funcion
  Constructor 2 de Test --> local a funcion()
Saliendo de: funcion
  Constructor 1 de Test --> VACIO
Saliendo de: main
```


En el caso de la clase `Polinomio`:

```
class Polinomio{
   ......
   public:
      // Constructores
      Polinomio (void);
      Polinomio (int grado);
      ......
};
```

Implementación de los constructores declarados:

```
// Constructor por defecto
Polinomio :: Polinomio (void){
  MaxGrado = 10;
  Grado = 0;
  Coef = new (nothrow) float [MaxGrado+1];
```

```
  if (!Coef) {
     cerr << "\nError: no hay memoria\n\n";
     exit (1);
  }
  for (int i=0; i<= MaxGrado; i++) Coef[i] = 0.0;
}
// Constructor con parametros
Polinomio :: Polinomio (int grado){
  MaxGrado = grado;
  Grado = 0;
  Coef = new (nothrow) float [MaxGrado+1];
  if (!Coef) {
     cerr << "\nError: no hay memoria\n\n";
     exit (1);
  }
  for (int i=0; i<= MaxGrado; i++) Coef[i] = 0.0;
}
```


Los datos miembro pueden iniciarse usando la *sección de iniciación* del constructor. Esta implementación realiza las mismas tareas que la anterior y su prototipo (declaración) es idéntica:

```
// Constructor por defecto (alternativa)
Polinomio::Polinomio(void): MaxGrado(10),Grado(0){
  Coef = new (nothrow) float [MaxGrado+1];
  if (!Coef) {
    cerr << "\nError: no hay memoria\n\n";
    exit (1);
  }
  for (int i=0; i<= MaxGrado; i++) Coef[i] = 0.0;
}
```


Los dos constructores presentados pueden resumirse en uno solo, haciendo uso de la posibilidad de emplear *valores por defecto* para los parámetros:

```
class Polinomio{
  ......
  public:
    // Constructor
    Polinomio (int GradoMaximo = 10);
    ......
};
Polinomio :: Polinomio (int GradoMaximo): MaxGrado(GradoMaximo){
  Grado = 0;
  Coef = new (nothrow) float [MaxGrado+1];
  if (!Coef) {
    cerr << "\nError: no hay memoria\n\n";
    exit (1);
  }
  for (int i=0; i<= MaxGrado; i++) Coef[i] = 0.0;
}
```


Ejemplo: Polinomio (1)

DemoPolinomio.cpp

```
#include <iostream>
#include <iomanip>
#include "ModuloPolinomio.h"
using namespace std;

int main (void){
   cout.setf (ios::showpoint);
   cout.setf (ios::fixed);

   Polinomio p1;
   p1.PintaPolinomio("p1 inic-def");
   p1.PonCoeficiente(2, 3.0);
   p1.PintaPolinomio("p1 con coef 2=3.0");
   p1.PonCoeficiente(4,-10.5);
   p1.PintaPolinomio("p1 con coef 2=3.0, 4=-10.5");
```

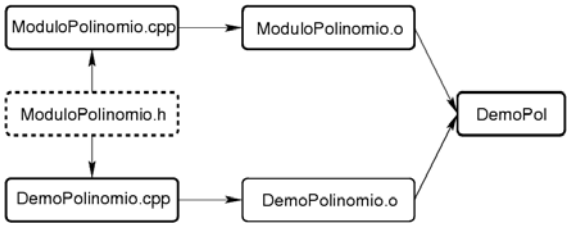
```
   cout << "p1(1) = " << setw(6) << setprecision(2)
        << p1.Evalua(1) << endl << endl;

   Polinomio p2 (5);
   p2.PintaPolinomio("p2 inic-MaxGrado=5");

   return (0);
}
```

El resultado de la ejecución de este código será:

```
p1 inic-def
   MaxGrado = 10   Grado =  0

p1 con coef 2=3.0
   MaxGrado = 10   Grado =  2
     3.00 X^2
```



```
p1 con coef 2=3.0, 4=-10.5
   MaxGrado = 10   Grado =  4
     3.00 X^2  -10.50 X^4

p1(1) =  -7.50

p2 inic-MaxGrado=5
   MaxGrado =  5   Grado =  0
```



ModuloPolinomio.h

```
#ifndef ModuloPolinomioH
#define ModuloPolinomioH
class Polinomio{
  private:
   float * Coef;
   int Grado;
   int MaxGrado;
  public:
   Polinomio (int GradoMaximo = 10); // Constructores
   void  PonCoeficiente (int i, float c); // Otros metodos
   int   LeeGrado (void) const;
   float LeeCoeficiente (int i) const;
   float Evalua (float x) const;
   void PintaPolinomio (const char * const msg) const;
};
#endif
```


ModuloPolinomio.cpp

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstdlib>
#include "ModuloPolinomio.h"
#using namespace std;
// Constructor por defecto y con parametros
Polinomio :: Polinomio (int GradoMaximo): MaxGrado(GradoMaximo){
   Grado = 0;
   Coef = new (nothrow) float [MaxGrado+1];
   if (!Coef) {
      cerr << "\nError: no hay memoria\n\n";
      exit (1);
   }
   for (int i=0; i<= MaxGrado; i++) Coef[i] = 0.0;
}
```

```
// Devuelve el grado del polinomio
int Polinomio ::  LeeGrado (void) const{
   return (Grado);
}

// Establece el valor del coeficiente de grado i
void Polinomio :: PonCoeficiente (int i, float valor){
   if ((i<0) && (i>MaxGrado)) {
      cerr << "\nError: grado incorrecto\n\n";
      exit (1);
   }
   Coef[i] = valor;
   if (i > Grado) Grado = i;
}
```



```
// Devuelve el valor del coeficiente de grado i
float Polinomio :: LeeCoeficiente (int i) const{
   if ((i<0) && (i>MaxGrado)) {
      cerr << "\nError: grado incorrecto\n\n";
      exit (1);
   }
   return (Coef[i]);
}

// Muestra en cout un polinomio
void Polinomio::PintaPolinomio (const char * const msg) const{
   cout.setf (ios::showpoint);
   cout.setf (ios::fixed);
   cout.setf (ios::right);
   cout << msg << endl;
   cout << "   MaxGrado = " << setw(2) << MaxGrado;
```



```
   cout << "   Grado = " << setw(2) << Grado << endl;
   cout << "   ";
   if (Coef[0]!=0)
      cout << setw(6) << setprecision(2) << Coef[0] << "  ";
   for (int i=1; i<=Grado; i++)
      if (Coef[i]!=0)
         cout << setw(6) << setprecision(2) << Coef[i] << " X^" << i << "  ";
   cout << endl << endl  ;
}
// Evalua el polinomio en el valor x
float Polinomio :: Evalua (float x) const{
   float res=0.0;
   for (int i=1; i<=Grado; i++)
      if (Coef[i]!=0) res += (Coef[i]*pow(x,Grado));
   return (res);
}
```


3.4 Destructor

- Sólo hay un destructor para una clase.
- Cuando un objeto deja de existir *siempre* se llama *automáticamente* al destructor.
- Un constructor tiene el mismo nombre que la clase, precedido por el carácter `~`

  No admite parámetros ni devuelve ningún valor.
- Si no se especifica, el compilador proporciona un ***destructor de oficio***. Su implementación tiene sentido sólo cuando el constructor ha reservado memoria dinámicamente.
- Basta con añadir la declaración (en el fichero `.h` y la definición (en el fichero `.cpp`) y no hay que modificar nada más.

```
class Polinomio{
   ......
   public:
      ......
      // Destructor
      ~Polinomio (void);
      ......
};

Polinomio :: ~Polinomio (void){
   delete [] Coef;
}
```

Importante: no hay que hacer ninguna llamada al destructor.

3.5 Prioridad de ejecución

En cuanto a los constructores:

- Los objetos globales se crean primero, antes de la ejecución de cualquier función (incluso `main()`).
- A continuación se ejecuta la función `main()` y se crean los objetos locales a ésta.
- Cuando se llama a cualquier función y ésta empieza su ejecución se crean los objetos locales a ella.

En cuanto a los destructores, actúan cuando un objeto deja de existir:

- Si es una variable local, al finalizar la ejecución de la función.

- Si es una variable global al finalizar la ejecución del programa.

Cuando la ejecución llega a la declaración de un objeto `static` se llama al constructor. El destructor actúa al finalizar la ejecución del programa.

Regla: Los destructores se ejecutan en orden inverso a la ejecución de los constructores respectivos.

```
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

class Test{
   private:
      string msg;
```



```
   public:
       Test (void);
       Test (string cad);
      ~Test (void);
};
Test :: Test (void){
   msg = "VACIO";
   cout << "  Constructor 1 de Test " << "--> " << msg << "\n";
}
Test :: Test (string cad){
   msg = cad;
   cout << "  Constructor 2 de Test " << "--> " << msg << "\n";
}
```

```
Test :: ~Test (void){
   cout << "  Destructor de Test  " << "----> " << msg << "\n";
}

Test global ("global"); // Objeto global

void funcion (void){
   cout << "Ejecutando : funcion\n";
   Test local_f ("local a funcion()");
   cout << "Saliendo de: funcion\n";
}
```



```
int main(int argc, char* argv[]){
   cout << "Ejecutando : main\n";
   Test local_m ("local a main()");

   funcion ();

   Test local_vacia;

   cout << "Saliendo de: main\n";
   return (0);
}
```


Su ejecución produce este resultado:

```
  Constructor 2 de Test --> global
Ejecutando : main
  Constructor 2 de Test --> local a main()
Ejecutando : funcion
  Constructor 2 de Test --> local a funcion()
Saliendo de: funcion
  Destructor de Test  ----> local a funcion()
  Constructor 1 de Test --> VACIO
Saliendo de: main
  Destructor de Test  ----> VACIO
  Destructor de Test  ----> local a main()
  Destructor de Test  ----> global
```


3.6 El constructor de copia

- Para iniciar un objeto a partir de otro existente.

```
int orig; // iniciacion por defecto
...
orig = 5;
...
int copia1 = orig; // iniciacion y copia
int copia2 (orig); // iniciacion y copia
```

- Si no se especifica un constructor de copia, el compilador proporciona un ***constructor de copia de oficio***.
- Este constructor de oficio realiza la copia bit a bit de los datos miembro.

Esto es válido en muchas ocasiones.

No es válido cuando el objeto requiere memoria dinámicamente: se necesita implementar un constructor de copia.

Declaración y definición de un constructor de copia:

```
class Polinomio{
  ......
  public:
  ......
  Polinomio (const Polinomio & otro);
  ......
};
```



```
// Constructor de copia
Polinomio::Polinomio (const Polinomio & otro){
   // Copia de datos miembro
   MaxGrado = otro.MaxGrado;
   Grado = otro.Grado;

   // Reserva de la propia memoria para coeficientes
   Coef = new (nothrow) float [MaxGrado+1];
   if (!Coef) {
      cerr << "\nError: no hay memoria\n\n";
      exit (1);
   }

   // Copia de los valores de los coeficientes
   for (int i=0; i<= MaxGrado; i++) Coef[i] = otro.Coef[i];
}
```


¿ Cómo se usa?

```
   ......
   Polinomio p1;
   p1.PintaPolinomio("p1 inic-def");
   p1.PonCoeficiente(2, 3.0);
   p1.PintaPolinomio("p1 con coef 2=3.0");
   p1.PonCoeficiente(4,-10.5);
   p1.PintaPolinomio("p1 con coef 2=3.0, 4=-10.5");
   cout << "p1(1) = " << setw(6) << setprecision(2)
        << p1.Evalua(1) << endl << endl;
   ......
   Polinomio p3 (p1);
   p3.PintaPolinomio("p3 copia de p1");
   Polinomio p4 = p1;
   p4.PintaPolinomio("p4 copia de p1");
   ......
```

La ejecución de este código produce este resultado:


```
p1 inic-def
   MaxGrado = 10   Grado =  0

p1 con coef 2=3.0
   MaxGrado = 10   Grado =  2
     3.00 X^2

p1 con coef 2=3.0, 4=-10.5
   MaxGrado = 10   Grado =  4
     3.00 X^2  -10.50 X^4

p1(1) =  -7.50

p3 copia de p1
   MaxGrado = 10   Grado =  4
     3.00 X^2  -10.50 X^4

p4 copia de p1
   MaxGrado = 10   Grado =  4
     3.00 X^2  -10.50 X^4
```

¿Cuándo se usa el constructor de copia?

1. Cuando se declara un objeto y se inicia a partir de otro de la misma clase.

   Este caso ya se ha tratado.

2. Cuando una función recibe un objeto por valor.

   Cada parámetro real es una variable local y se crea e inicia con el constructor de copia.

3. Cuando una función devuelve un objeto.

   El valor (el objeto) devuelto se creó localmente en la función. Para poder devolverse a la función que hace la llamada debe copiarse temporalmente a un objeto.

Nuestro consejo:

Implementar *siempre* el constructor de copia.

3.7 Funciones `inline`

La etiqueta `inline` indica ("sugiere" o "recomienda") al compilador que ponga una copia del código de esa función en el lugar donde se haga una llamada en lugar de realizar la llamada.

Ventaja: Tiempo de ejecución menor.

Desventaja: Programas de tamaño mayor ya que se crean varias copias del código.


```
class Polinomio{
  ......
  public:
    ......
    int LeeGrado (void) const;
    ......
};

......

inline int Polinomio:: LeeGrado (void) const{
   return (Grado);
}
```


En el siguiente caso el compilador considera a la función `inline`, aunque no se haya indicado explícitamente esa etiqueta:

```
class Polinomio{
  ......
  public:
    ......
    int LeeGrado (void) const {return (Grado);};
    ......
};
```

La función `LeeGrado()` es una función `inline` porque se declara y define en la misma clase.

3.8 El puntero `this`

El puntero `this` *es una variable predefinida en todas las funciones u operadores miembro de una clase. Contiene la dirección del objeto concreto de la clase sobre el cual se está aplicando la función u operador miembro.*

- Una función miembro (un método) de un objeto puede usar los datos miembro empleando solamente su nombre, y no se refiere al objeto sobre el que se aplica (éste es un *argumento implícito*).
- Si en una función u operador miembro queremos referirnos al objeto sobre el cual se están aplicando podemos emplear el puntero `this`.

Puede decirse que `*this` es un *alias* del objeto. Al ser un puntero, el acceso a los miembros del objeto se realizará con el operador ->


```
// Constructor de copia
Polinomio :: Polinomio (const Polinomio & otro){
   this->MaxGrado = otro.MaxGrado;
   this->Grado = otro.Grado;

   this->Coef = new (nothrow) float [(this->MaxGrado)+1];

   if (!(this->Coef)) {
      cerr << "\nError: no hay memoria\n\n";
      exit (1);
   }
   for (int i=0; i<= this->MaxGrado; i++)
      (this->Coef)[i] = otro.Coef[i];
}
```


3.9 Sobrecarga de operadores

C++ permite sobrecargar la mayoría de los operadores disponibles. El objetivo es que el programador de una clase emplee los operadores predefinidos de C++ en el contexto de la clase (operador + para la suma de polinomios, operador >> para la lectura de polinomios, etc.).

Limitaciones:

1. Puede modificarse la definición de un operador pero no su gramática (número de operandos, precedencia y asociatividad).
2. Un operando, al menos, debe ser un objeto de la clase en la que se define el operador.

Un operador puede estar, a su vez, sobrecargado, de manera que actúe según sea el tipo de los objetos que tiene como operandos.

La sobrecarga no es automática. Puede hacerse mediante:

- **Funciones miembro** de la clase:
  - Operadores unarios.
  - Operadores no unarios que modifican el primer operando (p.e., la asignación con =).

  En los operadores miembro de la clase, el primer operando debe ser un objeto de la clase: *el argumento implícito*.
- **Funciones amigas** (`friend`) de la clase.

  Operadores que actúan sobre varios objetos sin modificarlos.

  El número de argumentos debe ser el esperado por el operador.

Cualquier operador se sobrecarga escribiendo la definición de una función de la manera habitual salvo que el nombre de la función es la palabra `operator` seguida del símbolo del operador que se sobrecarga.

Operadores que pueden sobrecargarse:

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

Operadores que **no** pueden sobrecargarse:

```
.     .*     ::     ?:     sizeof
```


Es un error suponer que al sobrecargar un operador automáticamente se sobrecargan operadores relacionados.

Por ejemplo, al sobrecargar + no se sobrecarga automáticamente +=, ni al sobrecargar == lo hace automáticamente !=.

Los operadores sólo se sobrecargan explícitamente: ***no hay sobrecarga implícita***.

3.9.1 Sobrecarga del operador =

- Si no se sobrecarga, funciona de la manera habitual: se realiza una copia exacta (bit a bit) de un objeto en otro.
- **Problema crítico**: Si no se ha sobrecargado = y el objeto asignado ha reservado memoria dinámicamente, tras una asignación los dos objetos referencian a la misma zona de memoria.

Conclusión: Si los objetos de la clase reservan memoria dinámicamente se debería sobrecargar el operador de asignación.

Se trata de un operador binario en el que se modifica el primer operando: se implementa como una función miembro.

TIPO `&` `operator = (const` *TIPO* `&` orig`)`

En `ModuloPolinomio.h`:

```
class Polinomio{
 ......
 public:
  ......
  // Operador asignacion
  Polinomio & operator = (const Polinomio & otro);
};
```

En `ModuloPolinomio.cpp`:

```
Polinomio & Polinomio :: operator = (const Polinomio & otro){
   if (this != &otro) {
      MaxGrado = otro.MaxGrado;
      Grado = otro.Grado;
      Coef = new (nothrow) float [MaxGrado+1];
```

```
      if (!Coef) {
         cerr << "\nError: no hay memoria\n\n";
         exit (1);
      }
      for (int i=0; i<= MaxGrado; i++)
         Coef[i] = otro.Coef[i];
   }
   return (*this);
}
```

En `DemoPolinomio.cpp`:

```
   ......
   Polinomio p5, p6;
   p6 = p5 = p4;

   p6.PintaPolinomio("p6 asig");
   cout << "p6(1) = " << setw(6) << setprecision(2)
        << p6.Evalua(1) << endl << endl;
```


y el resultado de la ejecución es:

```
   ......
p6 asig
   MaxGrado = 10   Grado =  4
     3.00 X^2  -10.50 X^4

p6(1) =  -7.50
```

Obsérvese que **se devuelve una referencia**: la del objeto que recibe la petición de asignación (`*this`). De esta manera **se permite encadenar asignaciones**.

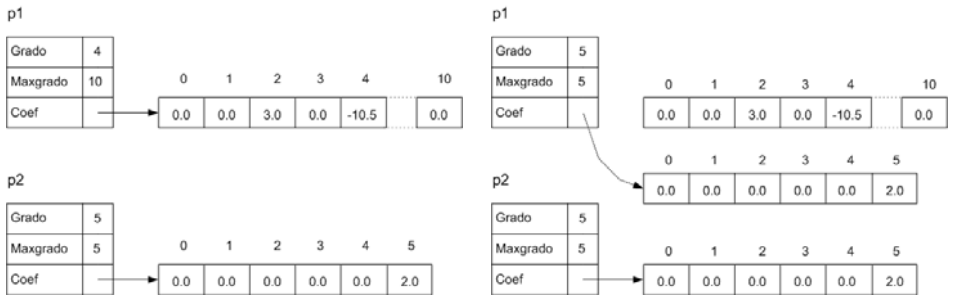
`a=b;` equivale a `a.operator = (b);`

`a=b=c;` equivale a `a = (b.operator = (c));`

`a=b=c;` NO es `(a.operator = (b)) = c;`

Problema: ¿Qué ocurre si el objeto al que se asigna el nuevo valor (parte izquierda de la asignación) tiene contenido (reservó memoria dinámicamente)?

```
Polinomio p1;
p1.PonCoeficiente(2, 3.0);
p1.PonCoeficiente(4,-10.5);
p1.PintaPolinomio("p1");

Polinomio p2 (5);
p2.PonCoeficiente(5,2.0);
p2.PintaPolinomio("p2");

p1 = p2;
p1.PintaPolinomio("p1 despues de p1=p2");
```

p1 y p2 antes de p1=p2 — p1 y p2 después de p1=p2

Se pierde la memoria que reservó el const. de `p1`.

Solución: Antes de la copia efectiva, liberar memoria:

```
Polinomio & Polinomio :: operator = (const Polinomio & otro){
   if (this != &otro) {
      delete [] Coef; // Liberar memoria reservada
      MaxGrado = otro.MaxGrado;
      Grado = otro.Grado;
      ......
   }
   return (*this);
}
```


3.9.2 Código reutilizable

El operador = y el constructor de copia tienen prototipos parecidos (reciben una referencia constante a otro objeto de la clase):

```
Polinomio & operator = (const Polinomio & otro);
Polinomio (const Polinomio & otro);
```

y realizan tareas parecidas sobre el objeto.

Sin embargo:

- El constructor actúa sobre un objeto que se está creando, por lo que no tiene sentido comprobar si `this != &otro`
- El constructor es el que reserva, inicialmente, memoria: no tiene sentido liberarla.

Las asignaciones que se realizan entre los datos miembros son iguales *¿porqué no unificarlas en una función auxiliar y privada?*

```
void Copia (const Polinomio & otro);
```

Buscando otras tareas comunes:

- La reserva de memoria la realizan *todos* los constructores y el operador de asignación.
- La liberación de la memoria la realizan el destructor y el operador de asignación.

Esta reserva (y la liberación respectiva) podría ser muy compleja. Repetir el código da lugar a programas difícilmente mantenibles: *¿porqué no realizar estas tareas con una función auxiliar y privada?*

```
void PideMemoria (int n);
void LiberaMemoria (void);
```


En `ModuloPolinomio.h`:

```
class Polinomio{
  private:
    ...... // Datos miembro
  public:
    ...... // Constructores, destructor, etc.
  private:
    // Nuevo: Funciones auxiliares privadas
    void Copia (const Polinomio & otro);
    void PideMemoria (int n);
    void LiberaMemoria (void);
};
```

En ModuloPolinomio.cpp:

```
void Polinomio :: Copia (const Polinomio & otro){
   MaxGrado = otro.MaxGrado;
   Grado = otro.Grado;
   PideMemoria (MaxGrado+1);
   for (int i=0; i<=MaxGrado; i++)
      Coef[i] = otro.Coef[i];
}
void Polinomio :: PideMemoria (int n){
   Coef = new (nothrow) float [n+1];
   if (!Coef) {
      cerr << "\nError: no hay memoria\n\n";
      exit (1);
   }
}
```



```
void Polinomio :: LiberaMemoria (void){
   delete [] Coef;
}
```

Con esta novedad, los métodos afectados son:

- Los constructores.
- El destructor.
- El operador de asignación.


```
// Constructor por defecto y con parametros
Polinomio :: Polinomio (int GradoMaximo = 10) : MaxGrado(GradoMaximo){
   Grado = 0;
   PideMemoria (MaxGrado+1);
   for (int i=0; i<= MaxGrado; i++) Coef[i] = 0.0;
}

// Constructor de copia
Polinomio :: Polinomio (const Polinomio & otro){
   Copia (otro);
}
```



```
// Destructor
Polinomio :: ~Polinomio (void){
   LiberaMemoria ();
}

// Operador de asignacion
Polinomio &  Polinomio :: operator = (const Polinomio & otro){
   if (this != &otro) {
      LiberaMemoria ();
      Copia (otro);
   }
   return (*this);
}
```

3.9.3 Operador []

El objetivo es acceder a los coeficientes del polinomio empleando una notación similar a la del acceso a las casillas de un vector.

Ahora hacemos:	Queremos:
`p1.PonCoeficiente (0, 6.4);`	`p1[0] = 6.4;`
`if (p1.LeeCoeficiente (i) != 0)`	`if (p1[i] != 0)`

Recordar: `p1[i]` equivale a `p1.operator[](i)`

- Se trata de un operador binario en el que el primer operando se modifica: se implementa como una *función miembro* de la clase. El primer operando (`p1`) es el objeto de la clase y es implícito.

- **Importante**: Permitimos que `p1[i]` pueda emplearse, indistintamente, en la parte derecha e izquierda (*lvalue*) de una asignación, por lo que la función que implementa el operador *devuelve una referencia*.

En `ModuloPolinomio.h`:

```
class Polinomio{
......
public:
   ......
   float & operator [] (const int indice);
   ......
};
```


En `ModuloPolinomio.cpp`:

```
float & Polinomio :: operator [] (const int indice){
   if ((indice<0) && (indice>MaxGrado)) {
      cerr << "\nError: grado incorrecto\n\n";
      exit (1);
   }
   if (indice > Grado)
      Grado = indice; // !!PROBLEMA!!

   return (Coef[indice]);
}
```


Problema:

La instrucción *problemática* "parchea" algunos casos:

```
Polinomio p1; // p1.Grado = 0      p1[1] = 3.0;  // p1.Grado = 1
p1[0] = 1.0;  // p1.Grado = 0      p1[4] = 4.0;  // p1.Grado = 4
```

Sin embargo, es errónea si hacemos:

```
p1[6] = 0.0;  // p1.Grado = 6 !!NO!!
```

incluso si:

```
cout << p1[7]; // p1.Grado = 7 !!NO!!
```

Conclusión: No puede mantenerse la coherencia si empleamos el dato miembro `Grado` y sobrecargamos el operador `[]`. Preferimos descartar el dato miembro `Grado` y "sustituirlo" por una función privada que calcula el grado cuando sea necesario.

En ModuloPolinomio.h:

```
class Polinomio{
private:
   float * Coef;
   int MaxGrado;
public:
   // Constructores - destructor
   Polinomio (int GradoMaximo = 10);
   Polinomio (const Polinomio & otro);
   ~Polinomio (void);
   // Operadores sobrecargados
   Polinomio &  operator = (const Polinomio & otro);
   float & operator [] (const int indice);
   // Otros metodos
   float Evalua (float x) const;
   void PintaPolinomio (const char * const msg) const;
```



```
private:
   int  Grado (void) const;
   void Copia (const Polinomio & otro);
   void PideMemoria (int n);
   void LiberaMemoria (void);
};
```

En ModuloPolinomio.cpp:

```
Polinomio :: Polinomio (int GradoMaximo = 10): MaxGrado(GradoMaximo){
   PideMemoria (MaxGrado+1);
   for (int i=0; i<= MaxGrado; i++) Coef[i] = 0.0;
}

Polinomio :: Polinomio (const Polinomio & otro){
   Copia (otro);
}
```



```
Polinomio :: ~Polinomio (void){
   LiberaMemoria ();
}
Polinomio & Polinomio :: operator = (const Polinomio & otro){
   if (this != &otro) {
      LiberaMemoria ();
      Copia (otro);
   }
   return (*this);
}
float & Polinomio :: operator [] (const int indice){
   if ((indice<0) && (indice>MaxGrado)) {
      cerr << "\nError: grado incorrecto\n\n";
      exit (1);
   }
   return (Coef[indice]);
}
```



```
void Polinomio :: PintaPolinomio (const char * const msg) const{
   cout.setf (ios::showpoint);
   cout.setf (ios::fixed);
   cout.setf (ios::right);
   int grado = Grado();
   cout << msg << endl;
   cout << "   MaxGrado = " << setw(2) << MaxGrado;
   cout << "   Grado = " << setw(2) << grado << endl;
   cout << "   ";
   if (Coef[0]!=0)
      cout <<setw(6)<<setprecision(2)<<Coef[0]<<"  ";
   for (int i=1; i<=grado; i++)
      if (Coef[i]!=0)
         cout << setw(6) << setprecision(2) << Coef[i] << " X^" << i << "  ";
   cout << endl << endl;
}
```

```
float Polinomio :: Evalua (float x) const{
   float res=0.0;
   int grado = Grado();

   for (int i=1; i<=grado; i++)
      if (Coef[i]!=0) res += (Coef[i] * pow (x,i));
   return (res);
}

void Polinomio :: Copia (const Polinomio & otro){
   MaxGrado = otro.MaxGrado;

   PideMemoria (MaxGrado+1);
   for (int i=0; i<=MaxGrado; i++)
      Coef[i] = otro.Coef[i];
}
```



```
void Polinomio :: PideMemoria (int n){
   Coef = new (nothrow) float [n+1];
   if (!Coef) {
      cerr << "\nError: no hay memoria\n\n";
      exit (1);
   }
}
void Polinomio :: LiberaMemoria (void){
   delete [] Coef;
}
int Polinomio :: Grado (void) const{
   int i;
   bool sigo = true;
   for (i=MaxGrado; (i>=0) && sigo; i--)
      if (Coef[i] != 0.0) sigo = false;
   return (i+1);
}
```


3.9.4 Operadores aritméticos

El objetivo es emplear los operadores aritméticos (+, −, etc.) para sumar, restar, etc. datos de tipo `Polinomio`.

Operador +

Deseamos escribir:

```
Polinomio p1, p2,p3;
......
p3 = p1 + p2;
```

en lugar de:

```
p3 = SumaPolinomios (p1, p2);
```


Se trata de un operador binario en el que los dos operandos deben ser siempre de tipo `Polinomio` y donde **no** se modifica ningún operando:

podría implementarse con una función `friend` o con una función miembro.

Nos decidimos por la *función miembro*.

El operador + se sobrecarga con este prototipo:

```
Polinomio operator + (const Polinomio & otro);
```

declarándose como una función miembro (pública) de la clase `Polinomio`, por lo que:

```
p3 = p1 + p2;
```

es igual que:

```
p3 = p1.operator+(p2);
```

En `ModuloPolinomio.cpp`:

```
Polinomio Polinomio::operator + (const Polinomio & otro){
  int MayorMaxGrado = (MaxGrado > otro.MaxGrado) ? MaxGrado : otro.MaxGrado;

  Polinomio tmp (MayorMaxGrado);

  for(int i=0; i<= tmp.MaxGrado; i++)
    tmp.Coef[i] = Coef[i] + otro.Coef[i];

  return (tmp);
}
```


Operador `*`

El operador `*` puede sobrecargarse para el producto de dos polinomios como antes (función miembro):

```
Polinomio operator * (const Polinomio & otro);
```

de manera que, por ejemplo:

`p3 = p1 * p2;` equivale a `p3 = p1.operator * (p2);`

Si se desea permitir, además, la multiplicación de un dato `float` por un polinomio:

```
Polinomio p3 = p1 * 2.0;
```

equivale a:

```
Polinomio p3 = p1.operator * (2.0);
```

y provoca efectos inesperados (por conversión implícita) aunque *no produce errores de compilación*.

La siguiente instrucción, en cambio, sí provoca errores de compilación:

```
Polinomio p4 = 2.0 * p1;
```

¿Por qué?

El operador `*` se permitirá en estos casos:

1. `p = p1 * p2;`
2. `p = p1 * 2.0;`
3. `p = 2.0 * p1;`

El caso 3 no puede implementarse con una función miembro, por lo que se emplea una función `friend`:

Una función `friend` *de una clase puede acceder a la parte privada de esa clase, aunque no sea una función miembro de la clase.*

Un operador en el que no pueda asegurarse que el primer argumento sea siempre un objeto de la clase no se implementará como una función miembro.

En el caso de un operador en el que alguna versión deba implementarse con una función `friend` *se recomienda sobrecargar el operador con funciones* `friend`*, con tantas versiones como casos se permitan.*

En `ModuloPolinomio.h`:

```
class Polinomio
{
......
public:
   ......
   friend Polinomio operator * (const Polinomio & p1, const Polinomio & p2);
   friend Polinomio operator * (const Polinomio & p, const float c);
   friend Polinomio operator * (const float c, const Polinomio & p);
   ......
};
```

En ModuloPolinomio.cpp:

```
Polinomio operator * (const float c, const Polinomio & p){
   Polinomio tmp (p.MaxGrado);

   for (int i=0; i<= tmp.MaxGrado; i++)
      tmp.Coef[i] = c * p.Coef[i];

   return (tmp);
}

Polinomio operator * (const Polinomio & p, const float c){
   return (c * p);
}

Polinomio operator * (const Polinomio & p1, const Polinomio & p2){
   // EJERCICIO
}
```


Notas:

- La etiqueta `friend` se especifica únicamente en la declaración.
- Las funciones `friend` **no** son miembros de la clase, de ahí que **no** se definan así:

```
Polinomio Polinomio::operator * (const Polinomio & p1, const Polinomio & p2){
   ......
}
```


3.9.5 Operadores << y >>

El objetivo es emplear los operadores << y >> de manera que pueda insertarse en el flujo `cout` un objeto de tipo `Polinomio` y puedan tomarse de `cin` los coeficientes de un polinomio.

```
Polinomio p1, p2;
......
cout << p1;
......
cin >> p2;
```

La implementación de estos operadores en el contexto de `cout` o `cin` involucra el uso de la clase base de éstas, `ostream` e `istream`.

Estas clases están declaradas en `iostream.h`, donde los operadores << y >> están sobrecargados para los tipos básicos y para la clase `string`.

La orden `cout << p1` involucra a dos operandos: un `ostream &` y otro `Polinomio &` y equivale a `cout.operator << (p1)`

En definitiva:

se trata de sobrecargar operadores de las clases `ostream` *e* `istream` *y no podemos añadir funciones miembro a estas clases.*

Estos operadores no pueden ser funciones miembro de la clase en la que se declaran ya que el primer operando (el implícito) no es un objeto de la clase:

`cout<<p1;` equivale a `cout.operator << (p1);`

Se declararán como funciones `friend` si necesitan acceder a la parte privada de la clase `Polinomio`.

En ModuloPolinomio.h:

```
class Polinomio
{
......
public:
   ......
   friend ostream & operator << (ostream &, const Polinomio &);
   friend istream & operator >> (istream &, Polinomio &);
   ......
};
```


En ModuloPolinomio.cpp:

```
ostream & operator << (ostream & out, const Polinomio & p){
   out.setf (ios::showpoint);
   out.setf (ios::fixed);
   out.setf (ios::right);
   int grado = p.Grado();
   out << "--------------------------------\n";
   out << "   MaxGrado = " << setw(2) << p.MaxGrado;
   out << "   Grado = " << setw(2) << grado << endl;
   cout << "   ";
   if (p.Coef[0]!=0)
      out << setw(6) << setprecision(2) << p.Coef[0] << "  ";
   for (int i=1; i<=grado; i++)
      if (p.Coef[i]!=0)
         out << setw(6) << setprecision(2) << p.Coef[i] << " X^" << i << "  ";
   out << "\n--------------------------------\n\n";
   return (out);
}
```



```
istream & operator >> (istream & in, Polinomio & p){
   int   ValGrado;
   float ValCoef;
   bool sigo = true;
   while (sigo) {
      cout << "Introduzca grado (Max = "<< p.MaxGrado << ", -1 para terminar):";
      in >> ValGrado;
      if (ValGrado < 0)
         sigo = false;
      else {
         cout << "   Introduzca coeficiente de grado " << ValGrado << ": ";
         in >> ValCoef;
         p[ValGrado] = ValCoef;
      }
   }
   return (in);
}
```


La ejecución de:

```
  Polinomio p1;
  p1[2] =   3.0;
  p1[4] = -10.5;
  p1[0] =   1.4;
  Polinomio p2 = p1 * 2.0;
  cout << "p1\n" << p1;
  cout << "p1(1) = " << setw(6) << setprecision(2) << p1.Evalua(1) << endl << endl;
  cout << "p2 = p1 * 2.0\n" << p2;
```

produce como resultado:

```
p1
--------------------------------
   MaxGrado = 10   Grado =  4
     1.40    3.00 X^2  -10.50 X^4
--------------------------------
```

```
p1(1) =  -7.50

p2 = p1 * 2.0
--------------------------------
   MaxGrado = 10   Grado =  4
     2.80    6.00 X^2  -21.00 X^4
--------------------------------
```

Respecto a la lectura, la ejecución de:

```
Polinomio p3;
cin  >> p3;
cout << "p3 leido\n" << p3;
```

produce como resultado:


```
Introduzca grado (Max = 10, -1 para terminar): 2
   Introduzca coeficiente de grado 2: 22.2
Introduzca grado (Max = 10, -1 para terminar): 5
   Introduzca coeficiente de grado 5: -5.5
Introduzca grado (Max = 10, -1 para terminar): 0
   Introduzca coeficiente de grado 0: 1
Introduzca grado (Max = 10, -1 para terminar): -1

p3 leido
--------------------------------
   MaxGrado = 10   Grado =  5
     1.00   22.20 X^2   -5.50 X^5
--------------------------------
```


3.10 Objetos y miembros `const`

Parámetros formales: ¿Paso de parámetros por valor o por referencia? Consideraremos:

1. El tamaño.

   En objetos o tipos no elementales debería hacerse por *referencia* (se pasa -y recibe- la dirección de memoria).

2. ¿Puede modificarse el argumento?

   Si **no** queremos que se modifique, debemos indicar que se trata de un objeto constante (`const`), tanto en el prototipo como en la definición.

Al declarar un objeto `const` no hay posibilidad de modificarlos: se produce un error en tiempo de compilación si intentamos modificarlos.

Iniciación de miembros `const`.

Los datos miembro `const` y las referencias se deben iniciar en el constructor con la *lista de iniciación* o *iniciadores de miembros*.

```
int VarGlobal = 10;

class UnaClase{
   private:
      const float C = 0.16; // ERROR
      int & ref = VarGlobal;   // ERROR
   ......
};
```

Deberían iniciarse:

```
int VarGlobal = 10;

class UnaClase{
   private:
      const float C;
      int & ref;
   public:
      UnaClase (void);
      ......
};

UnaClase :: UnaClase (void) : C(0.16), ref(VarGlobal){
   ......
}
```


Tambien es obligatoria, con iniciadores de miembros, la iniciación de un objeto de otra clase que sea miembro de la clase considerada.

```
#include <iostream>
#include <iomanip>
using namespace std;

class Clase1 {
   private:
      int Campo1Clase1;
      int Campo2Clase1;
   public:
      Clase1  (int v1=0, int v2=0);
      friend ostream & operator << (ostream &, const Clase1 &);
};
```



```
class Clase2 {
   private:
      Clase1 Campo1Clase2;
      int    Campo2Clase2;
   public:
      Clase2  (int v1=0, int v2=0, int v3=0);
      friend ostream & operator << (ostream &, const Clase2 &);
};

Clase1 :: Clase1 (int v1, int v2){
   Campo1Clase1 = v1;
   Campo2Clase1 = v2;
}

Clase2 :: Clase2 (int v1, int v2, int v3): Campo1Clase2 (v1, v2){
   Campo2Clase2 = v3;
}
```



```
ostream & operator << (ostream & out,const Clase2 & o){
   out <<"---------------------------------\n";
   out <<"Campo1Clase2: "<< o.Campo1Clase2<< endl;
   out <<"Campo2Clase2: "<< o.Campo2Clase2<< endl;
   out <<"---------------------------------\n\n";
   return (out);
}

ostream & operator << (ostream & out,const Clase1 & o){
   out <<"\n   .............................\n";
   out <<"   Campo1Clase1: "<< o.Campo1Clase1<< endl;
   out <<"   Campo2Clase1: "<< o.Campo2Clase1<< endl;
   out <<"   .............................\n";
   return (out);
}
```

```
int main(int argc, char* argv[]){
   Clase2 obj(1,2,3), obj2(55), obj3;
   cout << "obj: \n"  << obj  << "\n\n";
   cout << "obj2: \n" << obj2 << "\n\n";
   cout << "obj3: \n" << obj3 << "\n\n";
   return (0);
}
```

La ejecución produce este resultado:

```
obj:
--------------------------------
Campo1Clase2:
   ..............................
   Campo1Clase1: 1
   Campo2Clase1: 2
   ..............................
Campo2Clase2: 3
--------------------------------
```



```
obj2:
--------------------------------
Campo1Clase2:
   ..............................
   Campo1Clase1: 55
   Campo2Clase1: 0
   ..............................
Campo2Clase2: 0
--------------------------------

obj3:
--------------------------------
Campo1Clase2:
   ..............................
   Campo1Clase1: 0
   Campo2Clase1: 0
   ..............................
Campo2Clase2: 0
--------------------------------
```


Funciones `const` **y no** `const`.

Si no deseamos que una función miembro pueda modificar el valor de los datos debe expresarse como `const`, tanto en la declaración:

```
class Polinomio{
   ......
   float Evalua (float x) const;
   ......
}
```

como en la definición:

```
float Polinomio :: Evalua (float x) const{
   ......
}
```

Las funciones declaradas `const` pueden acceder a los miembros `const`.

Es un error que una función miembro `const` llame a otra función miembro no `const`.

Una función se puede sobrecargar como `const` y no `const`, de forma que el compilador eligirá la correcta en función de que el objeto en cuestión sea `const` o no.

Objetos `const`.

Un objeto constante es un objeto que no puede modificarse, por lo que cualquier intento de modificación es un error sintáctico.

No se puede llamar desde objetos constantes, a funciones no constantes: el compilador deshabilita, para los objetos `const`, el acceso a las funciones miembro no `const`.

Aunque un constructor no es una función `const`, sí se puede utilizar para iniciar un objeto `const`.

```
const Polinomio PolConst(5);
```

Incluso es posible que el constructor tenga que realizar una llamada a otra función que no sea `const`, pero en el momento que se crea el objeto ya se atendrá a las normas de los objetos constantes, hasta que se active el destructor, que tampoco es `const`.

3.11 Miembros `static`

Cada uno de los objetos que se crean dispone de sus propios datos miembro.

¿Pueden existir algunos datos miembro de una clase que sean comunes a todos los objetos de la clase?

Sí: miembros `static`.

- No se necesita una copia para cada objeto.
- Los miembros `static` existen aunque no existan objetos de la clase (reserva en la declaración).
- Pueden ser públicos o privados.

- Los datos miembro `static` no se inician en la declaración de la clase, sino en el fichero de definición de la clase (`.cpp`).

  No debe especificarse el modificador `static`.

  Sólo pueden iniciarse una sola vez.

  ```
  TIPO Clase::MiembroStatic = ValorInicial;
  ```

Caso típico de uso de datos `static`: contabilizar el número de objetos que hay creados.

Libros.h

```
#ifndef LibrosH
#define LibrosH
#include <string>
using namespace std;
```



```
class Libro {
   private:
      string Titulo;
      int NumPags;
      static int NumLibros;
   public:
      Libro (void);
      Libro (string, int);
      ~Libro (void);
      void Pinta (void);
      static int CuantosLibros (void);
};
#endif
```

Acceso:

- Miembros `static` públicos:
  - Cuando no existen objetos de la clase:
    `Clase::MiembroPublicoStatic`
  - Si existen objetos de la clase se accede de la manera habitual, aunque puede accederse como antes (aconsejable).
- Miembros `static` privados: a través de funciones públicas `static` de la clase. Acceso habitual.
- Los datos `static` tienen características de datos globales pero su alcance es el ámbito de la clase.

Libros.cpp

```
#include <iostream>
#include "Libros.h"
using namespace std;

int Libro::NumLibros = 0;  // Iniciacion

Libro :: Libro (string titulo, int num){
   Titulo = titulo;
   NumPags = num;
   NumLibros++;
}

Libro :: Libro (void){
   Titulo = "";
   NumPags = 0;
}
```



```
Libro & Libro :: operator = (const Libro & otro){
  if (this != &otro){
    this->Titulo = otro.Titulo;
    this->NumPags = otro.NumPags;
    NumLibros++;
  }
}
Libro :: ~Libro (void){
   NumLibros--;
}
void Libro :: Pinta (void){
   cout << Titulo << " " << NumPags << endl;
}
int Libro :: CuantosLibros (void){
   return (NumLibros);
}
```


Funciones `static`:

- Son funciones genéricas: no actúan sobre ningún objeto concreto de la clase.
- Si una función sólo accede a datos miembro `static` puede declararse `static`.

  Se invoca con el operador `.`(si existe algún objeto de la clase) o con el operador `::` (si no existen).
- El modificador `static` sólo aparece en la declaración de la función, no en la definición.
- No pueden hacer llamadas a funciones no `static`, ni a datos miembro no `static`.
- No pueden utilizar el puntero `this` ya que éste hace referencia al objeto concreto de la clase sobre la que se está trabajando.

DemoLibros.cpp

```
#include <iostream>
#include "Libros.h"
using namespace std;

int main (void){
   const int MAX = 10;
   Libro Coleccion[MAX];
   cout << "Libros al empezar = " << Libro::CuantosLibros() << endl << endl;
   cout << "Introduzca datos (* en Titulo para terminar)\n\n";
   string titulo;
   int num;
   bool sigo = true;
   while (sigo){
      int pos = Libro::CuantosLibros();
      cout << "Libro " << pos+1 << endl << "  Titulo: "; cin  >> titulo;
```



```
      sigo = (titulo != "*") ? true : false;
      if (sigo){
         cout << "  Paginas: "; cin  >> num;
         Coleccion[pos] = Libro (titulo, num);
      }
      if (pos+1 == MAX){
         sigo = false;
         cout << "\nCUIDADO: Coleccion llena.\n\n";
      }
   }
   int TotalLibros = Libro::CuantosLibros();
   cout << "Libros al terminar = " << TotalLibros << endl << endl;

   for (int pos=0; pos<TotalLibros; pos++)
      Coleccion[pos].Pinta();
   return (0);
}
```


Un ejemplo de ejecución:

```
Libros al empezar = 0

Introduzca datos (* en Titulo para terminar)

Libro 1
  Titulo: Primero
  Paginas: 100
Libro 2
  Titulo: Otro
  Paginas: 200
Libro 3
  Titulo: Ultimo
  Paginas: 300
Libro 4
  Titulo: *
```



```
Libros al terminar = 3

Primero 100
Otro    200
Ultimo  300
```