Metodología de la Programación

Tema 4. Clases en C++ (Ampliación)

Andrés Cano Utrera
(acu@decsai.ugr.es)
Departamento de Ciencias de la Computación e I.A.

DECSAI

ugr Universidad de Granada
ETSIIT Universidad de Granada

Curso 2016-17

Contenido del tema

Contenido del tema

Introducción: tipos de datos abstractos

Tipo de dato abstracto

Un **tipo de dato abstracto** (T.D.A.) es una colección de **datos** (posiblemente de tipos distintos) y un **conjunto de operaciones** de interés sobre ellos, definidos mediante una *especificación que es independiente de cualquier implementación* (es decir, está especificado a un alto nivel de abstracción).

Introducción: tipos de datos abstractos

TDA para polinomios

- Datos:
  - grado
  - coeficientes
  - ...
- Operaciones:
  - sumar
  - multiplicar
  - derivar
  - ...

Algunos (datos/métodos) aparecen de forma natural y otros como herramientas auxiliares para facilitar la implementación o el uso....

Introducción: tipos de datos abstractos

Implementación de un TDA

- **¿Cómo pueden implementarse?**: struct y class son las herramientas que nos permiten definir nuevos tipos de datos abstractos en **C++**.
- **Diferencias**: la principal diferencia entre ellos consiste en que por defecto los datos miembro son públicos en struct, mientras que en las clases (por defecto) son privados.

```
struct Fecha{
    int dia, mes, anio;
};
int main(){
    Fecha f;
    f.dia=3; // OK
}
```

```
class Fecha{
    int dia, mes, anio;
};
int main(){
    Fecha f;
    f.dia=3; // ERROR
}
```


Introducción

- Aunque podemos definir miembros privados en un struct, habitualmente no suele hacerse. Es más adecuado usar clases: aunque no se indique explícitamente que los datos miembro son privados, de forma predeterminada se limita su acceso.

```
struct Fecha{
    private:
        int dia, mes, anio;
};
```

```
class Fecha{
    int dia, mes, anio;
};
```

- Tanto las estructuras como las clases pueden contener métodos, aunque habitualmente las estructuras no suelen hacerlo. Recordad:
  - Si un struct necesitase contener métodos usaríamos class.
  - Los struct suelen usarse sólo para agrupar datos.

Introducción

- Los tipos de datos abstractos que se suelen definir con struct normalmente usan únicamente *abstracción funcional* (ocultamos los algoritmos, ya que los datos son públicos):

```
struct TCoordenada {
    double x,y;
};
void setCoordenadas(TCoordenada &c,double cx, double cy);
double getY(TCoordenada c);
double getX(TCoordenada c);

int main(){
    TCoordenada p1;
    setCoordenadas(p1,5,10);
    cout<<"x="<<getX(p1)<<", y="<<getY(p1)<<endl;
}
```

Introducción

- Los tipos de datos abstractos que se suelen definir con class usan además *abstracción de datos* (ocultamos la representación):

```
class TCoordenada {
    private:
        double x,y;

    public:
        void setCoordenadas(double cx, double cy);
        double getY();
        double getX();
};
int main(){
    TCoordenada p1;
    p1.setCoordenadas(5,10);
    cout<<"x="<<p1.getX()<<", y="<<p1.getY()<<endl;
}
```


Contenido del tema

La clase Polinomio

TDA Polinomio

- Construiremos una clase `Polinomio` para poder trabajar con polinomios del tipo:

$$4.5 \cdot x^3 + 2.3 \cdot x + 1$$

- El número de coeficientes es desconocido a priori: usaremos memoria dinámica.

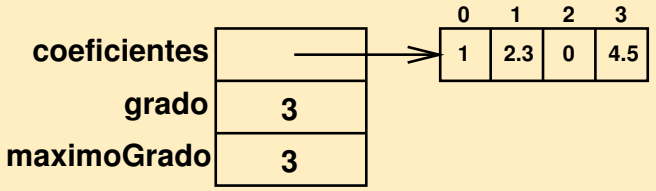
Como se vio al considerar el TDA Polinomio, los datos miembro naturales necesarios para representar este tipo de dato son:

- grado del polinomio
- coeficientes

La clase Polinomio

Implementación del TDA Polinomio: datos miembro

- Usaremos un ***array dinámico*** (guardado en memoria dinámica) para los coeficientes para permitir polinomios de cualquier grado.
- Se necesitará un dato miembro que indique el **máximo grado** posible, es decir, el tamaño concreto del array de coeficientes.



Este dato miembro se hace necesario al implementar la clase mediante memoria dinámica. ¿Sería necesario si hubiéramos usado la clase **vector**?

La clase Polinomio

```
#ifndef POLINOMIO
#define POLINOMIO
#include <assert.h>
class Polinomio {
  private:
     // Array con los coeficientes del polinomio
     float *coeficientes;

     //Grado del polinomio
     int grado;

     // Maximo grado posible: limitacion debida a la implementacion
     //de la clase: el array de coeficientes tiene un tamaño limitado
     int maximoGrado;

  public:

     ........................................
};

#endif
```


La clase Polinomio

Implementación del TDA Polinomio: métodos

Consideraremos ahora los diferentes métodos que deberían completar la definición del TDA para los polinomios (de la clase **Polinomio**). Conviene seguir el siguiente orden:

- constructores
- operaciones naturales sobre los polinomios (deberían ser métodos públicos)
- es posible que aparezcan otros métodos que resulten convenientes como métodos auxiliares (bien por la forma en que se ha hecho la implementación, bien por seguir el principio de descomposición modular....). Estos métodos deberían ser privados.

Asumimos que cada vez que se agregue un método debe incorporarse a la declaración de la clase, en el archivo **Polinomio.h**.

Contenido del tema

Los constructores de la clase Polinomio

Los constructores se encargan de inicializar de forma conveniente los datos miembro. En este caso deben además reservar la memoria dinámica que sea necesaria.

Constructor por defecto

Es el constructor sin parámetros. Una clase lo puede tener mediante:

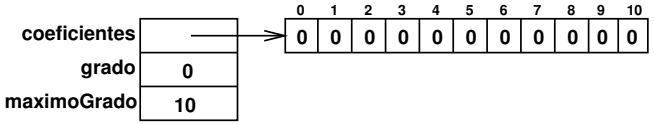
- El compilador lo crea implícitamente cuando la clase no define ningún constructor.
  - Tal constructor no inicializa los datos miembro de la clase.
  - Solo llama al constructor por defecto de cada dato miembro que sea un objeto de otra clase.
  - Un dato miembro no inializado probablemente contendrá un valor basura.
- Definiéndolo explícitamente en la clase.

Los constructores de la clase Polinomio

Constructor por defecto de la clase Polinomio

Crea espacio para un polinomio de hasta grado 10. Cabe plantearse qué valores dar a los datos miembro:

- 10 para el grado máximo: entendemos que correspondería a un polinomio donde la variable apareciese elevada a 10 (x^{10})
- 0 para el grado
- los coeficientes deberían inicializarse todos a cero



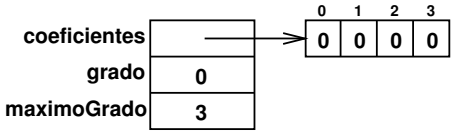
Constructores: constructor por defecto

```
/**
 * Constructor por defecto de la clase. El trabajo de este
 * constructor se limita a crear un objeto nuevo, con
 * capacidad maxima para guardar once coeficientes
 */
Polinomio::Polinomio(){
   // Se inicializan los datos miembro maximoGrado y grado
   maximoGrado=10;
   grado=0;

   // Se reserva espacio para el array de coeficientes
   coeficientes=new float[maximoGrado+1];

   // Se inicializan todos los coeficientes a 0
   for(int i=0; i<=maximoGrado; i++){
      coeficientes[i]=0.0;
   }
}
```


Los constructores de la clase Polinomio

Constructor con un parámetro que indica el grado máximo

Crea espacio para un polinomio con tamaño justo para que quepa un polinomio del grado máximo indicado.
Como en el constructor previo, el dato miembro grado se inicializa a 0 y los coeficientes toman también este valor



Constructores: constructor con valor de máximo grado

```
/**
 * Constructor de la clase indicando el maximo grado posible
 * @param maximoGrado valor del grado maximo
 */
Polinomio::Polinomio(int maximoGrado){
   // Si maximo grado es negativo se hace que el programa finalice
   assert(maximoGrado>=0);

   // Si el valor de maximoGrado es correcto, se asigna su
   // valor al dato miembro
   this->maximoGrado=maximoGrado;

   // Se inicializa a 0 el valor de grado
   grado=0;

   // Se reserva espacio para el array de coeficientes
   coeficientes=new float[maximoGrado+1];

   // Se inicializan a valor 0
   for(int i=0; i <= maximoGrado; i++){
      coeficientes[i]=0.0;
   }
}
```

Constructores

Ambos constructores comparten un trozo de código.
Resulta conveniente definir un método auxiliar que englobe las tareas comunes presentes en ambos constructores.
De esta forma pueden reescribirse apoyándonos en el método auxiliar (que será privado).

```
Polinomio::Polinomio(){
   maximoGrado=10;
   grado=0;

   coeficientes=new float[maximoGrado+1];

   for(int i=0; i<=maximoGrado; i++){
      coeficientes[i]=0.0;
   }
}
```

```
Polinomio::Polinomio(int maximoGrado){
   assert(maximoGrado>=0);

   this->maximoGrado=maximoGrado;

   grado=0;

   coeficientes=new float[maximoGrado+1];

   for(int i=0; i <= maximoGrado; i++){
      coeficientes[i]=0.0;
   }
}
```


Constructores

Podemos definir un método auxiliar (**inicializar**) que englobe estas sentencias. Este es un ejemplo de código que aparece no como consecuencia del análisis de la clase, sino como resultado del proceso de implementación.

```
/**
 * Metodo privado para inicialar el valor de grado y para
 * crear array de coeficientes de tamaño dado por el valor
 * de maximoGrado (más uno), poniendolos todos a cero
 */
void Polinomio::inicializar() {
   // Se inicializa a 0 el valor de grado
   grado = 0;

   // Se reserva espacio para el array de coeficientes
   coeficientes = new float[maximoGrado + 1];

   // Se inicializan a valor 0
   for (int i = 0; i <= maximoGrado; i++) {
      coeficientes[i] = 0.0;
   }
}
```


Constructores

Los constructores quedarían ahora:

```
Polinomio::Polinomio(){
   maximoGrado=10;

   inicializar();
}
```

```
Polinomio::Polinomio(int maximoGrado){
   assert(maximoGrado>=0);

   this->maximoGrado=maximoGrado;

   inicializar();
}
```


Constructores

De momento, la declaración de la clase contendría:

```
#ifndef POLINOMIO
#define POLINOMIO
#include <assert.h>
class Polinomio {
  private:
     // Array con los coeficientes del polinomio
     float *coeficientes;

     //Grado del polinomio
     int grado;

     // Maximo grado posible: limitacion debida a la implementacion
     //de la clase: el array de coeficientes tiene un tamaño limitado
     int maximoGrado;

     // Metodo inicializar para facilitar la programacion
     // de los constructores
     void inicializar();

  public:

     // Constructor por defecto
```

Constructores

Definición de constructores usando parámetros por defecto

Observando los dos constructores, se aprecia que la única diferencia consiste en la asignación explícita de valor al dato miembro maximoGrado:

- Podemos usar un **parámetro por defecto** para definir los dos constructores con uno solo.

```
.........................
class Polinomio{
 ..................
   /**
    * Constructor indicando el maximo grado posible
    * @param maxGrado
    */
   Polinomio(int maximoGrado = 10);
 ..................
}
```


Constructores

```
/**
 * Constructor de la clase indicando el maximo grado posible
 * @param maximoGrado valor del grado maximo
 */
Polinomio::Polinomio(int maximoGrado) {
   // Si maximo grado es negativo se hace que el programa
   // finalice
   assert(maximoGrado >= 0);

   // Si el valor de maximoGrado es correcto, se asigna su
   // valor al dato miembro
   this->maximoGrado = maximoGrado;

   // Se inicializan los demas datos miembro
   inicializar();
}
```


Contenido del tema

Los métodos de la clase Polinomio

Interfaz básico y adicional

A la hora de decidir qué métodos incluimos en la clase, debemos distinguir entre los que constituyen la **interfaz básica** y los que constituyen la **interfaz adicional**.

- Los métodos de la interfaz básica:
  - Deberían ser **pocos**: definen la funcionalidad básica.
  - Deberían definir una interfaz **completa**.
  - Suelen utilizar directamente los datos miembro de la clase.
- Los métodos de la interfaz adicional:
  - Pueden ser métodos de la clase o funciones externas en el tipo de dato abstracto.
  - Facilitan el uso del tipo de dato abstracto.
  - No deberían extenderse demasiado.
  - Aunque sean métodos, no es conveniente que accedan directamente a los datos miembro de la clase, ya que un cambio en la representación del TDA supondría cambiar todos los métodos adicionales.

Métodos const

Los ***métodos de la interfaz básica*** deben permitir acceder a los datos miembro de la clase.

- En la clase **Polinomio** los datos miembro de interés son el grado y los coeficientes de cada término.
- Para acceder a su valor se precisan los métodos siguientes:
  - **obtenerGrado**: obtiene el grado del polinomio
  - **obtenerCoeficiente**: obtiene el coeficiente asociado a un determinado término

Métodos const

Los métodos anteriores no modifican el objeto sobre el que se llaman, por lo que se declararán de forma especial para remarcar esta característica: **métodos const**.

Métodos const

```
/**
 * Obtiene el grado del objeto
 * @return grado
 */
int Polinomio::obtenerGrado() const {
   return grado;
}

/**
 * Permite acceder a los coeficientes del objeto.
 * @param indice asociado al coeficiente
 * @return coeficiente solicitado
 */
float Polinomio::obtenerCoeficiente(int indice) const {
   float salida = 0.0;
   // Se comprueba si el indice es menor o igual que el grado
   if (indice>=0 && indice <= grado){
      salida = coeficientes[indice];
   }
   return salida;
}
```


Métodos const

Los métodos anteriores se han definido como **métodos const**.

- Esto impide que accidentalmente incluyamos en tales métodos alguna sentencia que modifique algún dato miembro de la clase.
- Además, permite que sean utilizados con objetos declarados como *constantes*.

Métodos inline

Cuando un método es muy sencillo puede implementarse en la propia declaración de la clase como **método inline**.

- Los métodos **inline** se tratan de forma especial: no dan lugar a llamada a métodos (evitando el uso de la pila, etc). El compilador sustituye las llamadas al método por el bloque de sentencias que lo componen.
- Conviene limitar este tipo de métodos a aquellos que consten de pocas líneas de código.

  *El método de obtención de coeficiente puede reescribirse de forma más compacta. La declaración de la clase quedaría tal y como se indica a continuación (la palabra reservada **inline** es opcional).*

Métodos inline I

```
#ifndef POLINOMIO
#define POLINOMIO

#include <assert.h>

class Polinomio {
private:
   /**
    * Array con los coeficientes del polinomio
    */
   float *coeficientes;

   /**
    * Grado del polinomio
    */
   int grado;
```


Métodos inline II

```
   /**
    * Maximo grado posible: limitacion debida a la implementacion
    * de la clase: el array de coeficientes tiene un tamaño limitado
    */
   int maximoGrado;

   /**
    * Metodo auxiliar para inicializar los datos miembro
    */
   void inicializar();

public:
   /*
    * Constructor por defecto
    */
   Polinomio();
```


Métodos inline III

```
   /**
    * Constructor indicando el maximo grado posible
    * @param maxGrado
    */
   Polinomio(int maxGrado);

   // Metodos con const al final: no modifican al objeto
   // sobre el que se hace la llamada. Metodos inline: se
   // sustituyen por el codigo correspondiente

   /**
    * Obtiene el grado del objeto
    * @return grado
    */
   inline int obtenerGrado() const {
      return grado;
   }
```


Métodos inline IV

```
   /**
    * Permite acceder a los coeficientes del objeto. Si no se
    * trata de un coeficiente valido, devuelve 0
    * @param indice asociado al coeficiente
    * @return coeficiente solicitado
    */
   inline float obtenerCoeficiente(int indice) const {
      // Devuelve 0 si indice es mayor que grado o indice
      // menor que 0
      return ((indice > grado || indice < 0) ? 0.0 : coeficientes[indice]);
   }
};

#endif
```

Otros métodos de la interfaz básica

También hay que incluir métodos que permitan asignar valores a los coeficientes (ya que el grado se determinará teniendo en cuenta sus valores). Por esta razón se incorpora el método :

- **asignarCoeficiente**: permite asignar el coeficiente asociado a un determinado término

Otros métodos de la interfaz básica

Analizando qué debe hacer este método de asignación de coeficientes encontramos cuatro situaciones diferentes:

- si el índice pasado como argumento es mayor que el máximo grado, hay que **reservar más espacio de memoria** para los coeficientes, al excederse la capacidad de almacenamiento previo
- si el índice es mayor al actual grado y no es cero hay que actualizar el grado
- si el índice coincide con el máximo grado y es cero, entonces hay que **determinar el nuevo grado** del polinomio analizando los coeficientes
- en la situación normal basta con asignar el correspondiente coeficiente

El código se muestra en la siguiente transparencia.

Otros métodos de la interfaz básica

```
void Polinomio::asignarCoeficiente(int i, float c){
   if(i>=0){  // Si el indice del coeficiente es valido
      if(i>maximoGrado){  // Si necesitamos mas espacio
         float *aux=new float[i+1]; // Reservamos nueva memoria
         for(int j=0;j<=grado;++j)  // Copiamos coeficientes a nueva memoria
            aux[j]=coeficientes[j];
         delete[] coeficientes;             // Liberamos memoria antigua
         coeficientes=aux;                  // Reasignamos puntero de coeficientes
         for(int j=grado+1;j<=i;++j) //Hacemos 0 el resto de nuevos coeficientes
            coeficientes[j]=0.0;
         maximoGrado=i;  // Asignamos el nuevo numero maximo grado del polinomio
      }
      coeficientes[i]=c;  // Asignamos el nuevo coeficiente

      // actualizamos el grado
      if(c!=0.0 && i>grado)//Si coeficiente!=0 e indice coeficiente>antiguo grado
         grado=i;           // lo actualizamos al valor i
      else if(c==0.0 && i==grado)//Si coeficiente==0.0 e indice coeficiente==grado
         while(coeficientes[grado]==0.0 && grado>0)//Actualizamos grado con el
            grado--;                 //primer termino cuyo coeficiente no sea cero
   }
}
```


Métodos de la interfaz adicional: imprimir

- Añadimos un método para imprimir un polinomio en la forma: $4.5 \cdot x^3 + 2.3 \cdot x + 1$
- Puesto que no constituye un método de la *interfaz básica*, no accederá directamente a los datos miembro.

```
void Polinomio::imprimir() const{
  cout<<obtenerCoeficiente(obtenerGrado()); //Imprimir termino de grado may
  if(obtenerGrado()>0)
    cout<<"x^"<<obtenerGrado();
  for(int i=obtenerGrado()-1;i>=0;--i){   //Recorrer el resto de terminos
    if(obtenerCoeficiente(i)!=0.0){     //Si el coeficiente no es 0.0
      cout<<" + "<<obtenerCoeficiente(i);  //imprimirlo
      if(i>1)
        cout<<"x^"<<i;
      else if (i==1)
        cout<<"x";
    }
  }
  cout<<endl;
}
```

Métodos de la interfaz adicional: imprimir

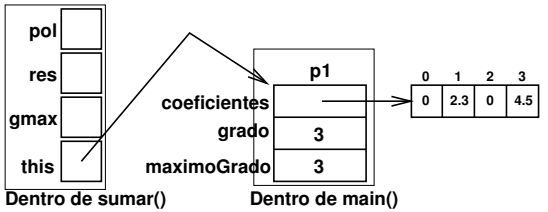
Un ejemplo de cómo se mostraría un objeto de la clase se muestra aquí:

```
+ 5.75x^4 + 4.56x^3 - 1.67x^2 + 2.3x + 5.8
```


Puntero this

Desde los métodos de una clase (o constructores), disponemos de un puntero que apunta al objeto usado para la llamada: **puntero** `this`.

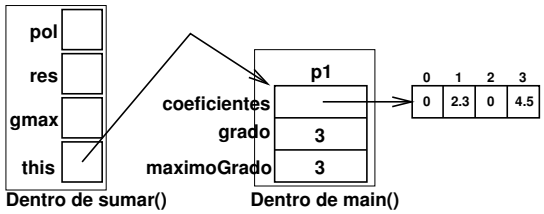
```
class Polinomio{
    ...
    public:
        Polinomio sumar(const Polinomio &pol) const;
}
Polinomio Polinomio::sumar(const Polinomio &pol) const{
    int gmax=(this->grado>pol.grado)?this->grado:pol.grado;
    int gmin=(this->grado<pol.grado)?this->grado:pol.grado;
    Polinomio res(gmax);
    for(int i=0;i<=gmin;++i) // asignar suma de coeficientes comunes
        res.asignarCoeficiente(i,this->coeficientes[i]+pol.coeficientes[i]);
    for(int i=gmin+1;i<=gmax;++i) // asignar resto de coeficientes
        res.asignarCoeficiente(i,(this->grado<pol.grado)?pol.coeficientes[i]:this->coeficientes[i]);
    return res;
}
int main(){
    Polinomio p1,p2;
    p1.asignarCoeficiente(3,4.5);
    p1.asignarCoeficiente(1,2.3);
    ...
    Polinomio p3=p1.sumar(p2);
}
```



Puntero this

Puesto que `sumar()` puede considerarse un método de la interfaz adicional, es mejor que no acceda directamente a los datos miembro de la clase.

```
class Polinomio{
    ...
    public:
        Polinomio sumar(const Polinomio &pol) const;
}
Polinomio Polinomio::sumar(const Polinomio &pol) const{
    int gmax=(this->obtenerGrado()>pol.obtenerGrado())?this->obtenerGrado():pol.obtenerGrado();
    Polinomio res(gmax);
    for(int i=0;i<=gmax;++i){
        res.asignarCoeficiente(i,this->obtenerCoeficiente(i) + pol.obtenerCoeficiente(i));
    }
    return res;
}
int main(){
    Polinomio p1,p2;
    p1.asignarCoeficiente(3,4.5);
    p1.asignarCoeficiente(1,2.3);
    ...
    Polinomio p3=p1.sumar(p2);
}
```



Ejemplo de uso del método **sumar** I

Un ejemplo de uso del método sumar se muestra a continuación:

```
int main(){
    // Prueba de la suma
    Polinomio sumando1(5);
    sumando1.asignarCoeficiente(0,3.8);
    sumando1.asignarCoeficiente(1,7.3);
    sumando1.asignarCoeficiente(2,-2.38);
    sumando1.asignarCoeficiente(3,-8.13);
    sumando1.asignarCoeficiente(4,6.63);
    sumando1.asignarCoeficiente(5,12.98);
    cout << "Sumando1: ";
    sumando1.imprimir();

    Polinomio sumando2(4);
    sumando2.asignarCoeficiente(0,5.8);
    sumando2.asignarCoeficiente(1,2.3);
    sumando2.asignarCoeficiente(2,-1.67);
    sumando2.asignarCoeficiente(3,4.56);
```

Ejemplo de uso del método **sumar** II

```
    sumando2.asignarCoeficiente(4,5.75);
    cout << "Sumando2: ";
    sumando2.imprimir();

    Polinomio resultado=sumando1.sumar(sumando2);
    cout << "Resultado: ";
    resultado.imprimir();
}
```

El resultado obtenido es:

```
Sumando1: + 12.98x^5 + 6.63x^4 - 8.13x^3 - 2.38x^2 + 7.3x + 3.8
Sumando2: + 5.75x^4 + 4.56x^3 - 1.67x^2 + 2.3x + 5.8
Resultado: + 12.98x^5 + 12.38x^4 - 3.57x^3 - 4.05x^2 + 9.6x + 9.6
```


Contenido del tema

Funciones y clases amigas (friend)

Las funciones y clases amigas (`friend`) pueden acceder a la parte privada de otra clase.

¡Cuidado!

Deben usarse puntualmente, por cuestiones justificadas de eficiencia. No es conveniente usarlas indiscriminadamente ya que **rompen el encapsulamiento** que proporcionan las clases.

```
class A {
    private:
        ...
    public:
        ...
    friend class B;
        ...
    friend tipo funcion(parametros);
};
```

- B es una clase amiga de A.
- Desde los métodos de B podemos acceder a la parte privada de A.
- `funcion()` es una función amiga de A.
- Desde `funcion()` podemos acceder a la parte privada de A.

Funciones y clases amigas (friend): Ejemplo

```
class ClaseA {
    int x;
    ...
    public:
        ...
        friend class ClaseB;
        friend void func();
};

void func() {
    ClaseA z;
    z.x = 6; // Acceso a z
    ...
}
```

```
class ClaseB {
    ...
    public:
        void unmetodo();
};
void ClaseB::unmetodo() {
    ClaseA v;
    v.x = 3; // Acceso a v
    ...
}
```

Contenido del tema

Usando la clase Polinomio

```
1 int main(){
2     Polinomio p1; // caben polinomios hasta grado 10
3     p1.asignarCoeficiente(3,4.5);
4     p1.asignarCoeficiente(1,2.3);
5     p1.imprimir();
6 }
```

- La línea 2 declara y crea un objeto `Polinomio` llamando al constructor por defecto.
- Las líneas 3 y 4 llaman al método `asignarCoeficiente()` de la clase `Polinomio`.
- La línea 5 llama al método `imprimir()` de la clase `Polinomio`.

¡Cuidado!

¿Qué ocurre con la memoria dinámica reservada por el constructor?

Usando la clase Polinomio

```
1 int main(){
2     Polinomio p1(3); // caben polinomios hasta grado 3
3     p1.asignarCoeficiente(3,4.5);
4     p1.asignarCoeficiente(1,2.3);
5     p1.imprimir();
6     p1.asignarCoeficiente(5,1.5); // caben polinomios hasta grado 5
7     p1.imprimir();
8 }
```

- La línea 2 declara y crea un objeto `Polinomio` en el que caben polinomios de hasta grado 3.
- La línea 6 hace que se amplíe el tamaño máximo del polinomio.

¡Cuidado!

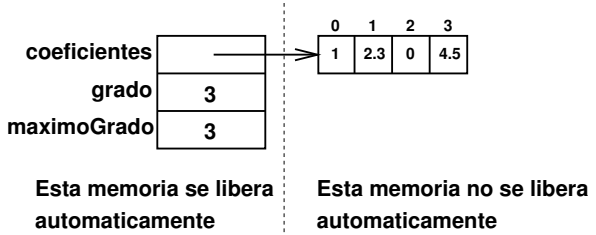
¿Qué ocurre con la memoria dinámica reservada por el constructor?

Contenido del tema

Destrucción automática de objetos locales

- Como se ha visto en temas anteriores, las variables locales se destruyen automáticamente al finalizar la función en la que se definen.
- En el siguiente código, p1 es una variable local: se destruirá automáticamente al acabar funcion().

```
float funcion(){
    ...
    Polinomio p1(3);
    ...
    return calculo;
}
int main() {
    ...
    a=funcion();
    ...
}
```



¿Cómo liberar la memoria dinámica del objeto?

- Como primera solución, podríamos pensar en un método que permita liberar la memoría dinámica del objeto.
- El método debe llamarse antes de que se destruya el objeto.

```
class Polinomio{
    private:
        ...
    public:
        ...
        void liberar();
};
...
void Polinomio::liberar(){
    delete[] coeficientes;
    grado=0;
    maximoGrado=-1;
}
```

```
float funcion(){
    ...
    Polinomio p1(3);
    ...
    p1.liberar();
    return calculo;
}
int main() {
    ...
    a=funcion();
    ...
}
```


El destructor de la clase Polinomio

- Se puede automatizar el proceso de destrucción implementando un método especial denominado destructor.
- El destructor es único, no lleva parámetros y no devuelve nada.
- Se ejecuta de forma automática, en el momento de destruir cada objeto de la clase:
  - Los objetos que son locales a una función o trozo de código, justo antes de acabar la función o trozo de código.
  - Los objetos variable global, justo antes de acabar el programa.

```
class Polinomio {
    private:
        ...
    public:
        ...
        ~Polinomio();
};
Polinomio::~Polinomio()
{
    delete[] coeficientes;
}
```

```
float funcion(){
    ...
    Polinomio p1(3);
    ...
    return calculo;
}
int main() {
    ...
    a=funcion();
    ...
}
```


Ejemplo de llamadas al destructor

Al ejecutar el siguiente ejemplo puede verse en qué momento se llama el destructor de la clase.

```
#include <iostream>
using namespace std;
class Prueba{
    public:
        Prueba();
        ~Prueba();
};
Prueba::Prueba(){
    cout<<"Constructor"<<endl;
}
Prueba::~Prueba(){
    cout<<"Destructor"<<endl;
}
void funcion(){
    Prueba local;
    cout<<"funcion()"<<endl;
}
Prueba varGlobal;
int main(){
    cout<<"Comienza main()"<<endl;
    Prueba ppal;
    cout<<"Antes de llamar a funcion()"<<endl;
    funcion();
    cout<<"Despues de llamar a funcion()"<<endl;
    cout<<"Termina main()"<<endl;
}
```

En la traza se han agregado comentarios para aclarar en qué momento se genera cada línea.

```
Constructor // Construccion objeto varGlobal
Comienza main() // Inicio ejecucion main
Constructor // Construccion objeto ppal
Antes de llamar a funcion()
Constructor // Construccion objeto local
funcion() // Ejecucion de funcion
Destructor // Se destruye objeto local (en el ambito de la funcion)
Despues de llamar a funcion() // De vuelta en main
Termina main() // Finaliza ejecucion main
Destructor // Se destruye objeto ppal
Destructor // Se destruye objeto varGlobal
```

Contenido del tema

Clases con datos miembro de otras clases

Un dato miembro de una clase puede ser de un tipo definido por otra clase

```
class Punto2D {
   double x,y;
   public:
      Punto2D() {
         cout<<"Ejecutando Punto2D()"<<endl;
         x=y=0.0; };
      Punto2D(double x, double y) {
         cout<<"Ejecutando Punto2D(double x, double y)"<<endl;
         this->x=x; this->y=y;};
      ~Punto2D() {cout<<"Ejecutando ~Punto2D()"<<endl; };
      void setX(double x) {this->x=x;};
      void setY(double y) {this->y=y;};
      double getX() const {return x;};
      double getY() const {return y;};
      void print() const {cout<<"x="<<getX()<<", y="<<getY()<<endl;};
};
```


Clases con datos miembro de otras clases

```
class Linea2D {
    Punto2D p1, p2;
    public:
        Linea2D();
        ~Linea2D();
        Punto2D getP1() const {return p1;};
        Punto2D getP2() const {return p2;};
        void print() const {cout<<"p1=";p1.print();
                            cout<<"p2=";p2.print();};
};

Linea2D::Linea2D()
{ // <-- En este punto se crean p1 y p2
    cout<<"Ejecutando Linea2D()"<<endl;
    p1.setX(-1); p1.setY(-1); // una vez creados les asignamos valores
    p2.setX(1); p2.setY(1); // (-1,-1) y (1,1) respectivamente
}
Linea2D::~Linea2D()
{
    cout<<"Ejecutando ~Linea2D()"<<endl;
} // <-- En este punto se destruyen p1 y p2
```


Clases con datos miembro de otras clases

Constructor

Un constructor de una clase:

- Llama al constructor por defecto de cada miembro.
- Ejecuta el cuerpo del constructor.

Destructor

El destructor de una clase:

- Ejecuta el cuerpo del destructor de la clase del objeto.
- Luego llama al destructor de cada dato miembro.

Clases con datos miembro de otras clases

- Ejecutando el siguiente código podemos ver en qué orden se ejecutan los constructores y destructores de las dos clases (Punto2D y Linea2D) al crear o destruir un objeto de la clase Linea2D.

```
int main(int argc, char *argv[]){
    cout<<"Comienza main()"<<endl;
    Linea2D lin; //<-- Aqui el compilador inserta llamada a constructor sobre lin
    lin.print();
                 //<-- lin deja de existir, el compilador inserta llamada
                 //    al destructor sobre lin
}
```

```
Comienza main()
Ejecutando Punto2D()
Ejecutando Punto2D()
Ejecutando Linea2D()
p1=x=-1, y=-1
p2=x=1, y=1
Ejecutando ~Linea2D()
Ejecutando ~Punto2D()
Ejecutando ~Punto2D()
```

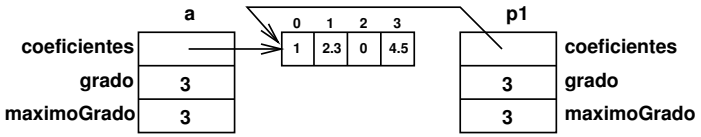

Contenido del tema

El constructor de copia por defecto

Construyamos una función (externa a la clase) que sume dos polinomios.

```
void sumar(Polinomio p1,Polinomio p2,Polinomio &res){
    int gmax=(p1.obtenerGrado()>p2.obtenerGrado())?p1.obtenerGrado():p2.obtenerGrado();
    for(int i=0;i<=gmax;++i)
        res.asignarCoeficiente(i,p1.obtenerCoeficiente(i)+p2.obtenerCoeficiente(i));
}
int main(){
    Polinomio a, b, r;
    ...
    sumar(a,b,r);
}
```

- En la llamada a `sumar()` se copian los objetos `a` y `b` en los parámetros formales `p1` y `p2` usando el **constructor de copia por defecto** proporcionado por C++.
- Este constructor hace una copia de cada dato miembro usando el constructor de copia de cada uno: **¿qué problemas da esto?**.



La copia se evita con el paso por referencia

- Haciendo que p1 y p2 se pasen por referencia constante, evitamos la copia de estos objetos.

```
void sumar(const Polinomio &p1,const Polinomio &p2,Polinomio &res){
    int gmax=(p1.obtenerGrado()>p2.obtenerGrado())?p1.obtenerGrado():p2.obtenerGrado();
    for(int i=0;i<=gmax;++i)
        res.obtenerCoeficiente(i,p1.obtenerCoeficiente(i)+p2.obtenerCoeficiente(i));
}
int main(){
    Polinomio a, b, r;
    ...
    sumar(a,b,r);
}
```

- Pero lo adecuado es indicar cómo se haría una copia de forma adecuada mediante la definición de un constructor de copia propio para esta clase.

Creación de un constructor de copia

- Es posible crear un constructor de copia que haga una copia correcta de un objeto de la clase en otro.
- Al ser un constructor, tiene el mismo nombre que la clase.
- No devuelve nada y tiene como único parámetro, constante y por referencia, el objeto de la clase que se quiere copiar.
- Copia el objeto que se pasa como parámetro en el objeto que construye el constructor.
- Se llama automáticamente al hacer un paso por valor para copiar el parámetro actual en el parámetro formal.

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

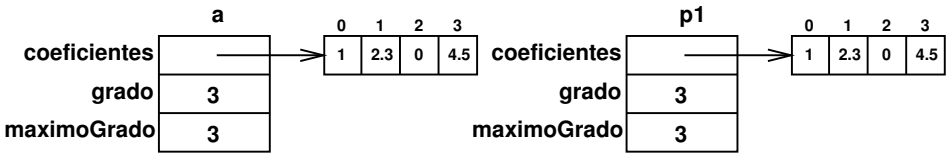

Creación de un constructor de copia

```
class Polinomio {
    private:
        ...
    public:
        ...
        Polinomio(const Polinomio &p);
};
Polinomio::Polinomio(const Polinomio &p){
    maximoGrado=p.maximoGrado;
    grado=p.grado;
    coefientes=new float[maximoGrado+1];
    for(int i=0; i<=maximoGrado; ++i)
        coefientes[i]=p.coefientes[i];
}
```


Creación de un constructor de copia

- Ahora la copia se hace correctamente.

```
void sumar(Polinomio p1,Polinomio p2,Polinomio &res){
    int gmin=(p1.obtenerGrado()<p2.obtenerGrado())?p1.obtenerGrado():p2.obtenerGrado();
    for(int i=0;i<=gmin;++i)
        res.asignarCoeficiente(i,p1.obtenerCoeficiente(i)+p2.obtenerCoeficiente(i));
}
int main(){
    Polinomio a, b, r;
    ...
    sumar(a,b,r);
}
```



¿Cuándo llama C++ al constructor de copia?

- Como acabamos de ver, se llama cuando se pasa un parámetro por valor al llamar a una función o método.
- También podemos llamarlo de forma explícita en las siguientes formas:

```
Polinomio p1,p2;
...
Polinomio p3(p1);
Polinomio p4=p2;
```

- Hay otros casos en que podría llamarse, pero depende del compilador que usemos:
  - Cuando una función devuelve (`return`) un objeto por valor.

Ejemplo sin constructor de copia

```
class Ejemplo{
    private:
        int *p; // La clase usa memoria dinamica
        int z;  // y un miembro estatico
    public:
        Ejemplo();   //Constructor por defecto
        ~Ejemplo();  //Destructor
        void get(int &p, int &z) const {p=*(this->p); z=this->z;};
        void set(int p, int z){*(this->p)=p; this->z=z;};
        void print() const {cout<<"*p="<<*p<<"; z="<<z<<endl;};
} ;
Ejemplo::Ejemplo() {
    cout << " Constructor "<<endl;
    p = new int; // Reservamos memoria
    *p = 2; // Iniciamos *p y z con el valor 2
    z = 2;
}
Ejemplo::~Ejemplo() {
    cout << " Destructor "<<endl;
    delete p; // Liberamos memoria dinamica
}
```


Ejemplo sin constructor de copia

- Al no haber constructor de copia, se usa el de por defecto en las llamadas a `funcParamValor()`.
- Hay un problema al destruir el objeto `x` de `funcParamValor()`.

```
void funcParamValor(Ejemplo x) {
    cout << " Funcion funcParamValor(Ejemplo x)  ";
    x.print();
}
void funcParamRef(Ejemplo &x) {
    cout << " Funcion funcParamRef(Ejemplo &x) ";
    x.print();
}
int main() {
    cout << "Creamos a"<<endl;
    Ejemplo a;
    cout << "Mostramos valores de a: ";
    a.print();
    cout << "Llamamos a la funcion funcParamRef()"<<endl;
    funcParamRef(a);
    cout << "Mostramos valores de a: ";
    a.print();
    cout << "Llamamos a la funcion funcParamValor()"<<endl;
    funcParamValor(a);
    cout << "Mostramos valores de a: ";
    a.print();
    cout << "Fin"<<endl;
}
```


Ejemplo sin constructor de copia I

La salida obtenida es:

```
Creamos a
 Constructor
Mostramos valores de a: *p=2; z=2
Llamamos a la funcion funcParamRef()
 Funcion funcParamRef(Ejemplo &x) *p=2; z=2
Mostramos valores de a: *p=2; z=2
Llamamos a la funcion funcParamValor()
 Funcion funcParamValor(Ejemplo x) *p=2; z=2
 Destructor
Mostramos valores de a: *p=0; z=2
Fin
 Destructor
*** glibc detected *** ./Ejemplov12.bin: double free or corruption (fasttop): ↙
↳ 0x0000000001649010 ***
======= Backtrace: =========
/lib64/libc.so.6[0x3c78c7cb3e]
./Ejemplov12.bin[0x400a79]
./Ejemplov12.bin[0x400be3]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x3c78c21a05]
```


Ejemplo sin constructor de copia II

```
./Ejemplov12.bin[0x400909]
======= Memory map: ========
00400000-00402000 r-xp 00000000 08:06 12978941 Ejemplov12.bin
.....................................................................................
↳
Abortado (‘core’ generado)
```

Ejemplo añadiendo el constructor de copia

- El problema se soluciona al añadir el constructor de copia

```
class Ejemplo{
        ...
    public:
        ...
        Ejemplo(const Ejemplo &x); // Constructor de copia
};
...
Ejemplo::Ejemplo(const Ejemplo &x) {
    cout << " Constructor de copia "<<endl;
    p = new int; // reservamos memoria para la copia
    *p = *(x.p); // Copiamos valores de *p y z
    z = x.z;
}
```


Ejemplo con constructor de copia I

La salida obtenida ahora carece de errores:

```
Creamos a
 Constructor
Mostramos valores de a: *p=2; z=2
Llamamos a la funcion func_param_ref()
 Funcion func_param_ref *p=2; z=2
Mostramos valores de a: *p=2; z=2
Llamamos a la funcion func_param_valor()
 Constructor de copia
 Funcion func_param_valor *p=2; z=2
 Destructor
Mostramos valores de a: *p=2; z=2
Fin
 Destructor
```


Con. copia por defecto en clases con datos de otras clases I

El **constructor de copia por defecto** proporcionado por C++ hace una copia de cada dato miembro llamando a sus constructores de copia.

```
class Punto {
    double x,y;
public:
    Punto();
    Punto(const Punto& punto);
    double getX() const{return x;}
    double getY() const{return y;}
    void set(double newXValue,double newYValue){
        this->x=newXValue;this->y=newYValue;}
    void escribir() const {cout << "(" << x << "," << y << ")";}
};
Punto::Punto(){
  cout<<"Punto::Punto()"<<endl;
  x=10.0; y=20.0;
}
Punto::Punto(const Punto& punto){
  cout<<"Punto::Punto(const Punto& punto)"<<endl;
  this->x=punto.x; this->y=punto.y;
}
```


Con. copia por defecto en clases con datos de otras clases II

```
class Circulo {
    Punto centro;
    double radio;
public:
    Circulo();
    ~Circulo();
    void set(Punto centro, double radio){this->centro=centro;this->radio=radio;}
    Punto getCentro() const{return centro;}
    double getRadio() const{return radio;}
    void escribir() const;
};
Circulo::Circulo(){
    cout<< "Circulo::Circulo()"<<endl;
    radio=1.0;
}
Circulo::~Circulo(){
    cout<< "Circulo::~Circulo()"<<endl;
}
void Circulo::escribir() const{
    cout << radio << "-";
    centro.escribir();
}
```

Con. copia por defecto en clases con datos de otras clases III

```
int main()
{
    Circulo c1;
    Circulo c2=c1;

    c1.escribir(); cout<<endl;
    c2.escribir(); cout<<endl;
}
```

La salida obtenida es:

```
Punto::Punto()
Circulo::Circulo()
Punto::Punto(const Punto& punto)
1-(10,20)
1-(10,20)
Circulo::~Circulo()
Circulo::~Circulo()
```


Contenido del tema

¿Cómo llamar a los distintos constructores?

- A continuación vemos varios ejemplos de creación de objetos con llamadas a distintos constructores.

```
Polinomio p1; // Usa el constructor por defecto
Polinomio p2(p1); // Usa el constructor de copia
Polinomio p3=p1; //  Usa el constructor de copia
Polinomio p4(3); // Usa el constructor con un parametro int
```

- En las líneas 2 y 3 que aparecen a continuación se usa el constructor por defecto o el que tiene un `int`, pero a la vez se está usando el método `operator=` para hacer la asignación (lo veremos en el próximo tema).

```
Polinomio p, x; //Usa el constructor por defecto
p = Polinomio();//Crea p con constructor por defecto y lo asigna a p con operator=
x = Polinomio(3);//Crea p con constructor con int y lo asigna a x con operator=
```


Conversiones implícitas

Conversión implícita

Cualquier constructor (excepto el de copia) de un sólo parámetro puede ser usado por el compilador de C++ para hacer una conversión automática de un tipo al tipo de la clase del constructor.

```
class Polinomio{
...
public:
    Polinomio(int max_g);
...
}

double evalua(const Polinomio p1, double x){
    double res=0.0;
    for(int i=0;i<=p1.obtenerGrado();i++){
        res+=p1.obtenerCoeficiente(i)*pow(x,i);
    }
    return res;
}

int main(){
    Polinomio p1;
    p1.asignarCoeficiente(3,4.5);
    ...
    evalua(p1,2.5); //
    evalua(3,2.5);  // Se hace un casting implícito del entero 3 a un objeto Polinomio
}
```

Conversiones implícitas

Especificador explicit

En caso de que queramos impedir que se haga este tipo de conversión implícita, declararemos el constructor correspondiente como `explicit`.

```
class Polinomio{
...
public:
    explicit Polinomio(int max_g);
...
}

double evalua(const Polinomio p1, double x){
    double res=0.0;
    for(int i=0;i<=p1.obtenerGrado();i++){
        res+=p1.obtenerCoeficiente(i)*pow(x,i);
    }
    return res;
}

int main(){
    Polinomio p1;
    p1.asignarCoeficiente(3,4.5);
    ...
    evalua(p1,2.5); //
    evalua(3,2.5);  // Error de compilación
}
```


Conversiones implícitas

```
g++ -Wall -g -c pruebaPolinomio.cpp -o pruebaPolinomio.o
pruebaPolinomio.cpp: En la función 'int main()':
pruebaPolinomio.cpp:68:15: error: no se encontró una función coincidente para la llamada a 'Polin
pruebaPolinomio.cpp:68:15: nota: el candidato es:
In file included from pruebaPolinomio.cpp:2:0:
Polinomio.h:22:19: nota: Polinomio Polinomio::sumar(const Polinomio&) const
Polinomio.h:22:19: nota:   no hay una conversión conocida para el argumento 1 de 'int' a 'const P
make: *** [pruebaPolinomio.o] Error 1
```

Listas de inicialización en constructores

- Añadimos un nuevo constructor a Linea2D

```
class Linea2D {
    Punto2D p1, p2;
    public:
        Linea2D();
        Linea2D(const Punto2D &pun1, const Punto2D &pun2);
        ...
};

Linea2D::Linea2D(const Punto2D &pun1, const Punto2D &pun2)
{   // Aqui se crean p1 y p2
    // A continuacion se les da el valor inicial
    cout<<"Llamando a Linea2D::Linea2D(const Punto2D &pun1, const Punto2D &pun2)"<<endl;
    p1.setX(pun1.getX()); // Se inicia p1
    p1.setY(pun1.getY());
    p2.setX(pun2.getX()); // Se inicia p2
    p2.setY(pun2.getY());
}

int main(int argc, char *argv[])
{
    cout<<"Comienza main()"<<endl;
    Punto2D p1,p2;
    p1.setX(10);p1.setY(10);
    p2.setX(20);p2.setY(20);
    Linea2D lin(p1,p2);
                //<----  Aqui el compilador inserta llamada a constructor sobre lin
    lin.print();
                //<----- lin deja de existir, el compilador inserta llamada
                //       al destructor sobre lin
}
```


Listas de inicialización en constructores

- Con la lista de inicialización se usa el constructor deseado para los datos miembro de Linea2D en lugar del constructor por defecto.

```
class Linea2D {
    Punto2D p1, p2;
    public:
        Linea2D();
        Linea2D(const Punto2D &pun1, const Punto2D &pun2);
        ...
};

Linea2D::Linea2D(const Punto2D &pun1, const Punto2D &pun2)
    : p1(pun1), p2(pun2) // Se crean p1 y p2 usando el constructor deseado (de copia en este caso)
{
    cout<<"Llamando a Linea2D::Linea2D(const Punto2D &pun1, const Punto2D &pun2)"<<endl;
}

int main(int argc, char *argv[])
{
    cout<<"Comienza main()"<<endl;
    Punto2D p1,p2;
    p1.setX(10);p1.setY(10);
    p2.setX(20);p2.setY(20);
    Linea2D lin(p1,p2);
                //<----  Aqui el compilador inserta llamada a constructor sobre lin
    lin.print();
                //<----- lin deja de existir, el compilador inserta llamada
                //       al destructor sobre lin
}
```

Creación/destrucción de objetos en memoria dinámica

- La segunda línea del siguiente trozo de código reserva espacio en memoria dinámica para un objeto `Polinomio` seguida de una llamada a su constructor por defecto.

```
Polinomio *p;
p=new Polinomio();
```

- Podemos usar el constructor deseado:

```
Polinomio *p,*q;
p=new Polinomio(3);
q=new Polinomio(*p);
```

- Para llamar al destructor y luego liberar la memoria dinámica ocupada por el objeto usaremos:

```
delete p;
```

- Los operadores `new[]` y `delete[]` tienen un comportamiento similar.

```
Polinomio *p;
p=new Polinomio[100];
delete[] p;
```