

Metodología de la Programación

Tema 4. Clases en C++ (Ampliación)

Departamento de Ciencias de la Computación e I.A.



ETSIIT Universidad de Granada

Curso 2012-13

Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

Introducción

Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

Introducción

- Un **tipo de dato abstracto** (T.D.A.) es una colección de datos (posiblemente de tipos distintos), junto con unas operaciones sobre ellos, definidos mediante una especificación que es independiente de cualquier implementación.
- Los struct y class son las herramientas que nos permiten definir nuevos tipos de datos abstractos en C++.
- La diferencia entre ellos es que por defecto en los struct, los miembros son públicos, mientras que en class por defecto los miembros 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. Se usa un class en su lugar.

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

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

- Tanto struct como class pueden contener métodos, pero habitualmente los struct no suelen hacerlo.

- Si un struct necesitase contener métodos usariamos un class.
- Los struct suelen usarse sólo para agrupar datos miembro.

Introducción

- Los tipos de datos abstractos que se suelen definir con struct suelen hacer uso sólo de *abstracción funcional* (ocultamos los algoritmos):

```
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

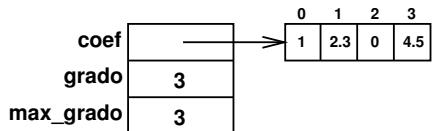
- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

La clase 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 impredecible: usaremos memoria dinámica.



La clase Polinomio

```
class Polinomio {
    private:
        float *coef;           // Array con los coeficientes
        int grado;             // Grado de este polinomio
        int max_grado;         // Maximo grado permitido en este polinomio
    public:
        Polinomio();           // Constructor por defecto
        Polinomio(int maxGrado); // Otro constructor
        void setCoeficiente(int i, float c);
        float getCoeficiente(int i) const;
        int getGrado() const;
        void print() const;
};
```

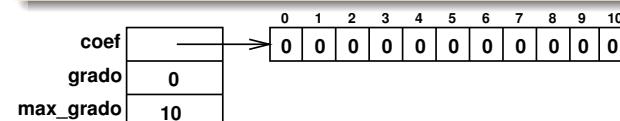
Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores**
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

Los constructores de la clase Polinomio

Constructor por defecto

Crea espacio para un polinomio de hasta grado 10.



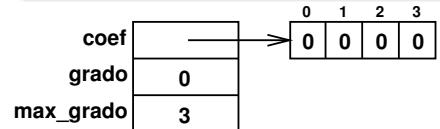
- Los constructores se encargan de inicializar los datos miembro.
- En este caso, además es necesario reservar la memoria dinámica necesaria.

```
Polinomio::Polinomio(){
    max_grado=10; // Tamaño por defecto
    grado=0;
    coef=new float[max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        coef[i]=0.0;
}
```

Los constructores de la clase Polinomio

Constructor con un parámetro que indica el grado

Crea espacio para un polinomio con tamaño justo para que quepa un polinomio del grado indicado.



```
Polinomio::Polinomio(int max_g){
    assert(max_g>=0);
    max_grado=max_g;
    grado=0;
    coef=new float[max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        coef[i]=0.0;
}
```

Los constructores de la clase Polinomio

- Puesto que ambos constructores comparte un trozo de código, podemos reescribirlos ayudándonos de un nuevo método privado:

```
Polinomio::Polinomio(){
    max_grado=10; // Tamaño por defecto
    inicializa();
}
Polinomio::Polinomio(int max_g){
    assert(max_g>=0);
    max_grado=max_g;
    inicializa();
}
void Polinomio::inicializa(){
    grado=0;
    coef=new float[max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        coef[i]=0.0;
}
```

Los constructores de la clase Polinomio

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

```
1 class Polinomio {
2     private:
3         float *coef;      // Array con los coeficientes
4         int grado;       // Grado de este polinomio
5         int max_grado;   // Máximo grado permitido en este polinomio
6         void inicializa();
7     public:
8         Polinomio(int maxGrado=10);
9         ...
10    };
11 Polinomio::Polinomio(int max_g=10){
12     assert(max_g>=0);
13     max_grado=max_g;
14     inicializa();
15 }
16 void Polinomio::inicializa(){
17     grado=0;
18     coef=new float[max_grado+1];
19     for(int i=0; i<=max_grado; ++i)
20         coef[i]=0.0;
21 }
```

Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase**
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

Los métodos de la clase Polinomio

- A la hora de decidir qué métodos incluimos en la clase, debemos distinguir entre los que constituyen el **interfaz básico** y los que constituyen el **interfaz adicional**.
- Los métodos del interfaz básico:
 - Deberían ser **pocos**: definen la funcionalidad básica.
 - Deberían definir una interfaz **completa**.
 - Suelen utilizar directamente los datos miembro de la clase.
- Los métodos del 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

```
int Polinomio::getGrado() const
{
    return grado;
}
float Polinomio::getCoeficiente(int i) const
{
    return ((i>grado) || (i<0))?0.0:coef[i];
}
```

- Se han definido como **métodos const**.

- Esto impide que accidentalmente incluyamos en tales métodos alguna sentencia que modifique algún dato miembro de la clase.

Métodos inline

```
class Polinomio {
private:
    float *coef;           // Array con los coeficientes
    int grado;             // Grado de este polinomio
    int max_grado;         // Maximo grado permitido en este polinomio
public:
    ...
    void setCoeficiente(int i, float c);
    inline float getCoeficiente(int i) const{
        return ((i>grado) || (i<0))?0.0:coef[i];
    }
    inline int getGrado() const{
        return grado;
    }
};
```

- Los métodos *inline* se implementan dentro de la definición de la clase.
- La palabra reservada *inline* es opcional.

```
1 void Polinomio::setCoeficiente(int i, float c){
2     if(i>0){ // Si el indice del coeficiente es valido
3         if(i>max_grado){ // Si necesitamos mas espacio
4             float *aux=new float[i+1]; // Reservamos nueva memoria
5             for(int j=0;j<=grado;++j) // Copiamos coeficientes a nueva memoria
6                 aux[j]=coef[j];
7             delete[] coef; // Liberamos memoria antigua
8             coef=aux; // Reasignamos puntero de coeficientes
9             for(int j=grado+1;j<=i;++j) //Hacemos 0 el resto de nuevos coeficientes
10                coef[j]=0.0;
11            max_grado=i; // Asignamos el nuevo numero maximo grado del polinomio
12        }
13        coef[i]=c; // Asignamos el nuevo coeficiente
14
15        // actualizamos el grado
16        if(c!=0.0 && i>grado)//Si coeficiente!=0 e indice coeficiente>antiguo grado
17            grado=i; // lo actualizamos al valor i
18        else if(c==0.0 && i==grado)//Si coeficiente==0.0 e indice coeficiente==grado
19            while(coef[grado]==0.0 && grado>0)//Actualizamos grado con el primer
20                grado--; //termino cuyo coeficiente no sea cero
21
22 }
```

Métodos adicionales

- Añadimos un método para imprimir un polinomio en la forma:

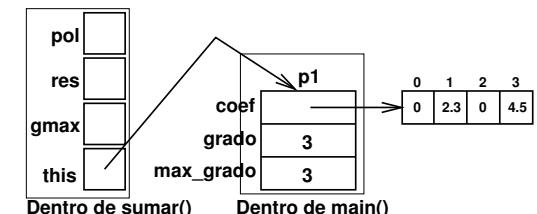
$$4.5 \cdot x^3 + 2.3 \cdot x + 1$$
- Puesto que no constituye un método del *interfaz básico* no accederá directamente a los datos miembro.

```
void Polinomio::print() const{
    cout << getCoeficiente(getGrado()); //Imprimir termino de grado mayor
    if(getGrado() > 0)
        cout << "x^" << getGrado();
    for(int i = getGrado() - 1; i >= 0; --i){ //Recorrer el resto de terminos
        if(getCoeficiente(i) != 0.0){ //Si el coeficiente no es 0.0
            cout << " + " << getCoeficiente(i); //imprimirllo
            if(i > 0)
                cout << "x^" << i;
        }
    }
    cout << endl;
}
```

Puntero this

Desde dentro de los métodos de una clase (o constructores), disponemos de un puntero que apunta al objeto que hace la llamada del método: el puntero se llama *this*.

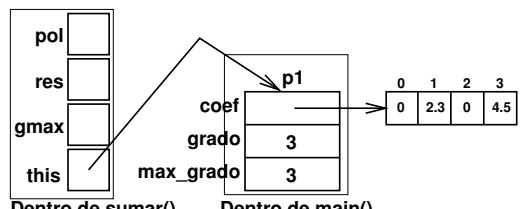
```
class Polinomio{
    ...
public:
    Polinomio Polinomio::sumar(const Polinomio &pol) const;
}
Polinomio Polinomio::sumar(const Polinomio &pol) const{
    int gmax=(this->getGrado()>pol.getGrado())?this->getGrado():pol.getGrado();
    Polinomio resultado(gmax);
    for(int i=0;i<=gmax;++i){
    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



Puntero this

- Puesto que *sumar()* puede considerarse una función del interfaz adicional, es mejor que no acceda directamente a los datos miembro de la clase.

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



Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

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.

Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase**
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

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();
};

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

Usando la clase Polinomio

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



- La línea 2 declara y crea un objeto `Polinomio` llamando al constructor por defecto.
- Las líneas 3 y 4 llaman al método `setCoeficiente()` de la clase `Polinomio`.
- La línea 5 llama al método `print()` 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.setCoeficiente(3,4.5);
4     p1.setCoeficiente(1,2.3);
5     p1.print();
6     p1.setCoeficiente(5,1.5); // caben polinomios hasta grado 5
7     p1.print();
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?

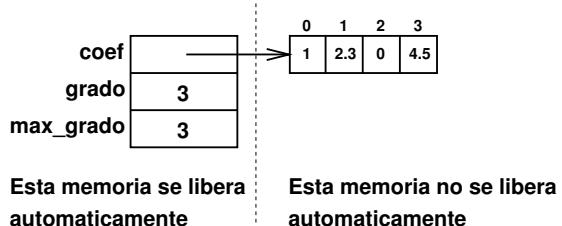
Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

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 memoria dinámica del objeto.
- El método debe llamarse antes de que se destruya el objeto.

```
class Polinomio{
    float funcion(){
        ...
        Polinomio p1(3);
        ...
        void liberar();
    };
    ...
    void Polinomio::liberar(){
        delete[] coef;
        grado=0;
        max_grado=-1;
    }
}
```

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 {
    float funcion(){
        ...
        Polinomio p1(3);
        ...
        return calculo;
    }
    ~Polinomio();
};

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



Ejemplo de llamadas al destructor

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

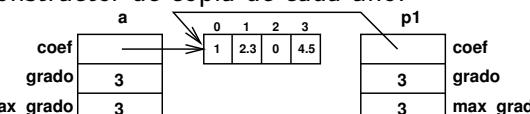
```
1 #include <iostream>
2 using namespace std;
3 class Prueba{
4     public:
5         Prueba();
6         ~Prueba();
7 };
8 Prueba::Prueba(){
9     cout<<"Constructor"<<endl;
10 }
11 Prueba::~Prueba(){
12     cout<<"Destructor"<<endl;
13 }
14 void funcion(){
15     Prueba local;
16     cout<<"funcion()"<<endl;
17 }
18 Prueba varGlobal;
19 int main(){
20     cout<<"Comienza main()"<<endl;
21     Prueba ppal;
22     cout<<"Antes de llamar a funcion()"<<endl;
23     funcion();
24     cout<<"Despues de llamar a funcion()"<<endl;
25     cout<<"Termina main()"<<endl;
26 }
```

Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Otros métodos
 - Métodos adicionales en el interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia**
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Otros ejemplos
- 9 Llamadas a constructores y destructores
 - Clases con atributos de otras clases
 - Conversiones implícitas
 - Listas de inicialización
 - Creación/destrucción de objetos en memoria dinámica

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.getGrado()>p2.getGrado())?p1.getGrado():p2.getGrado();
 for(int i=0;i<=gmax;i++)
 res.setCoeficiente(i,p1.getCoeficiente(i)+p2.getCoeficiente(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**.



## 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.getGrado()>p2.getGrado())?p1.getGrado():p2.getGrado();
 for(int i=0;i<=gmax;++i)
 res.setCoeficiente(i,p1.getCoeficiente(i)+p2.getCoeficiente(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);
};
```

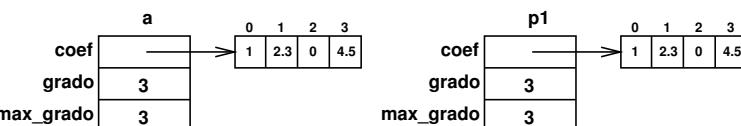
## Creación de un constructor de copia

```
class Polinomio {
 private:
 ...
 public:
 ...
 Polinomio(const Polinomio &p);
};
Polinomio::Polinomio(const Polinomio &p){
 cout<<"Llamando al constructor de copia"<<endl;
 max_grado=p.max_grado;
 grado=p.grado;
 coef=new float[max_grado+1];
 for(int i=0; i<=max_grado; ++i)
 coef[i]=p.coef[i];
}
```

## Creación de un constructor de copia

- Ahora la copia se hace correctamente.

```
void sumar(Polinomio p1,Polinomio p2,Polinomio &res){
 int gmax=(p1.getGrado()>p2.getGrado())?p1.getGrado():p2.getGrado();
 for(int i=0;i<=gmax;++i)
 res.setCoeficiente(i,p1.getCoeficiente(i)+p2.getCoeficiente(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.

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

- También podemos llamarlo de forma explícita en las siguientes formas:

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

## Ejemplo sin constructor de copia

- Al no haber constructor de copia, se usa el de por defecto.
- En las llamadas a funcParamValor() se usa este constructor.
- 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

```
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){p=*(this->p); z=this->z;};
 void set(int p, int z){*(this->p)=p; this->z=z;};
 void print(){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 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;
}
```



## Otro ejemplo sin constructor de copia

- En este ejemplo, podemos comprobar que el constructor de copia por defecto proporcionado por C++ hace una copia de cada dato miembro llamando a los distintos constructores de copia.

```
class Ejemplo{
 int z;
 Otra o;
public:
 Ejemplo() { //Constructor por defecto
 ~Ejemplo(); //Destructor
 void get(int &z){z=this->z;};
 void set(int z){this->z=z;};
 void print(){cout<<"z="<<z<<endl;
 o.print();};
 };
 Ejemplo::Ejemplo(){
 cout << "Ejemplo::Ejemplo()"<<endl;
 z = 2;
 }
 Ejemplo::~Ejemplo(){
 cout << "Ejemplo::~Ejemplo()"<<endl;
 }
 int main(){
 Ejemplo a;
 Ejemplo b=a; //Llamada al constructor de copia
 a.print(); b.print();
 }
}
```



## Contenido del tema

- 1 Introducción
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
  - Métodos const
  - Métodos inline
  - Otros métodos
  - Métodos adicionales en el interfaz de la clase
  - Puntero this
- 5 Funciones y clases friend
- 6 Usando la clase
- 7 El destructor
- 8 El constructor de copia
  - El constructor de copia por defecto
  - Creación de un constructor de copia
  - Otros ejemplos
- 9 Llamadas a constructores y destructores
  - Clases con atributos de otras clases
  - Conversiones implícitas
  - Listas de inicialización
  - Creación/destrucción de objetos en memoria dinámica

## Clases con atributos 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(){cout << "x=" << getX() << ", y=" << getY() << endl;};
};

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

## ¿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=
```

## Clases con atributos de otras clases

```

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

```

### 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.

## 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.getGrado();i++){
 res+=p1.getCoeficiente(i)*pow(x,i);
 }
 return res;
}

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

```

## Clases con atributos 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;
 // Aquí el compilador inserta llamada a constructor sobre lin
 lin.print();
 // lin deja de existir, el compilador inserta llamada
 // al destructor sobre lin
}

```



## 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.getGrado();i++){
 res+=p1.getCoeficiente(i)*pow(x,i);
 }
 return res;
}

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

```

## 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 'Polinomio::sumar(const Polinomio&)' const
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
```



- 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)
: 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);
 //---- Aquí el compilador inserta llamada a constructor sobre lin
 lin.print();
 //---- lin deja de existir, el compilador inserta llamada
 // al destructor sobre lin
}
```



- 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);
 //---- Aquí el compilador inserta llamada a constructor sobre lin
 lin.print();
 //---- lin deja de existir, el compilador inserta llamada
 // al destructor sobre lin
}
```



- 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)
// Aquí se crean p1 y p2
// A continuación 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);
 //---- Aquí 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;
```