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.

UNIVERSIDAD
DE GRANADA

Curso 2017-18

Contenido del tema I

Contenido del tema II

- Listas de inicialización en constructores
- Arrays de objetos
- Creación/destrucción de objetos en memoria dinámica
- Uso de constructores para hacer conversiones implícitas

Contenido del tema

Abstracción

Abstracción

Proceso mental que consiste en realzar los detalles relevantes sobre el objeto de estudio, mientras que se ignoran los detalles irrelevantes.

Conducción de un automóvil

Para conducir un automóvil no necesito conocer cómo funciona el motor y la electrónica del coche, solo necesito saber unos pocos detalles sobre los pedales y cuadro de mandos.

Abstracción

Abstracción

Proceso mental que consiste en realzar los detalles relevantes sobre el objeto de estudio, mientras que se ignoran los detalles irrelevantes.

Conducción de un automóvil

Para conducir un automóvil no necesito conocer cómo funciona el motor y la electrónica del coche, solo necesito saber unos pocos detalles sobre los pedales y cuadro de mandos.

Abstracción

Abstracción

Proceso mental que consiste en realzar los detalles relevantes sobre el objeto de estudio, mientras que se ignoran los detalles irrelevantes.

Conducción de un automóvil

Para conducir un automóvil no necesito conocer cómo funciona el motor y la electrónica del coche, solo necesito saber unos pocos detalles sobre los pedales y cuadro de mandos.

Contenido del tema

Abstracción funcional

Abstracción funcional

Consiste en ocultar los detalles de cómo están hechas las operaciones (funciones).
Solo se necesita conocer cómo se usan las operaciones (funciones).

Ejemplos de funciones en C++

Podemos usar las siguientes funciones sin saber cómo están implementadas:

```
double sqrt(double x);
size_t strlen(const char *cadena);
void qsort(void *elementos, size_t num_elemem, size_t tam_elem,
           int (*comparador)(const void *, const void *));
```

Abstracción funcional

Abstracción funcional

Consiste en ocultar los detalles de cómo están hechas las operaciones (funciones).
Solo se necesita conocer cómo se usan las operaciones (funciones).

Ejemplos de funciones en C++

Podemos usar las siguientes funciones sin saber cómo están implementadas:

```
double sqrt(double x);
size_t strlen(const char *cadena);
void qsort(void *elementos, size_t num_elemem, size_t tam_elem,
           int (*comparador)(const void *, const void *));
```

Ejemplo (media de un array): solución directa sin módulos

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

int main(int argc, char* argv[]){
   int *arrayint;
   int nElementos;
   double suma;

   cout << "Número de elementos: ";
   cin >> nElementos;
   if(nElementos<=0){
       cerr << "Número de elementos deber ser positivo" << endl;
   }
   else{
      arrayint = new int[nElementos];
      for(int i=0; i<nElementos; i++)
         cin >> arrayint[i];

      suma=0.0;
      for(int i=0; i<nElementos; i++)
         suma+=arrayint[i];

      for(int i=0; i<nElementos; i++)
         cout << arrayint[i] << " ";
      cout << endl;
      cout << "Media=" << suma/nElementos << endl;

      delete[] arrayint; // Libera la memoria dinámica reservada
   }
}
```

Ejemplo (media de un array): solución directa sin módulos

Inconvenientes del diseño anterior

- Código no reutilizable
  - El código que lee un array.
  - El código que muestra un array en la salida estándar.
  - El código para hacer la suma de los elementos de un array.
- Suele conllevar repetición de código
- Cualquier modificación suele implicar cambios en muchas partes.

Ejemplo (media de un array): solución directa sin módulos

Inconvenientes del diseño anterior

- Código no reutilizable
  - El código que lee un array.
  - El código que muestra un array en la salida estándar.
  - El código para hacer la suma de los elementos de un array.
- Suele conllevar repetición de código
- Cualquier modificación suele implicar cambios en muchas partes.

Ejemplo (media de un array): solución directa sin módulos

Inconvenientes del diseño anterior

- Código no reutilizable
  - El código que lee un array.
  - El código que muestra un array en la salida estándar.
  - El código para hacer la suma de los elementos de un array.
- Suele conllevar repetición de código
- Cualquier modificación suele implicar cambios en muchas partes.

Ejemplo (media de un array): solución directa sin módulos

Inconvenientes del diseño anterior

- Código no reutilizable
  - El código que lee un array.
  - El código que muestra un array en la salida estándar.
  - El código para hacer la suma de los elementos de un array.
- Suele conllevar repetición de código
- Cualquier modificación suele implicar cambios en muchas partes.

Ejemplo (media de un array): solución directa sin módulos

Inconvenientes del diseño anterior

- Código no reutilizable
  - El código que lee un array.
  - El código que muestra un array en la salida estándar.
  - El código para hacer la suma de los elementos de un array.
- Suele conllevar repetición de código
- Cualquier modificación suele implicar cambios en muchas partes.

Ejemplo (media de un array): solución directa sin módulos

Inconvenientes del diseño anterior

- Código no reutilizable
  - El código que lee un array.
  - El código que muestra un array en la salida estándar.
  - El código para hacer la suma de los elementos de un array.
- Suele conllevar repetición de código
- Cualquier modificación suele implicar cambios en muchas partes.

Ejemplo (media de un array): usando Abstracción Funcional

```
// Arraydin_int.h

#ifndef _ARRAYDIN_INT_H
#define _ARRAYDIN_INT_H

void inicializar(int * &arrayint, int &nElementos);
void redimensionar (int* &arrayint, int& nElementos, int aumento);
void liberar(int * &arrayint, int &nElementos);
bool leer(int * &arrayint, int &nElementos);
void mostrar(const int *arrayint, int nElementos):
double sumar(const int *arrayint, int nElementos);

#endif
```

Ejemplo (media de un array): usando Abstracción Funcional

```
// Arraydin_int.cpp

#include <iostream>
using namespace std;

void inicializar(int * &arrayint, int &nElementos){
    arrayint=0;
    nElementos=0;
}

void redimensionar (int* &arrayint, int& nElementos, int aumento){
   if(nElementos+aumento > 0){
      int *v_ampliado = new int[nElementos+aumento];

      for (int i=0; (i<nElementos) && (i<nElementos+aumento); i++)
         v_ampliado[i] = arrayint[i];
      delete[] arrayint;
      arrayint = v_ampliado;
      nElementos+=aumento;
   }
}
```

Ejemplo (media de un array): usando Abstracción Funcional

```
void liberar(int * &arrayint, int &nElementos){
    delete[] arrayint;
    arrayint = 0;
    nElementos = 0;
}

bool leer(int * &arrayint, int &nElementos){
    int n;
    cin >> n;
    if(n<=0)
        return false;
    redimensionar(arrayint, nElementos, n);
    for(int i=0; i<nElementos; i++)
        cin >> arrayint[i];
    return true;
}
```

Ejemplo (media de un array): usando Abstracción Funcional

```
void mostrar(const int *arrayint, int nElementos){
    for(int i=0; i<nElementos; i++)
         std::cout << arrayint[i] << " ";
    cout << endl;
}

double sumar(const int *arrayint, int nElementos){
    double suma=0.0;
    for(int i=0; i<nElementos; i++)
        suma+=arrayint[i];
    return suma;
}
```

Ejemplo (media de un array): usando Abstracción Funcional

```
#include <iostream>
#include "Arraydin_int.h"

using namespace std;

int main(int argc, char* argv[]){
   int *arrayint;
   int nElementos;
   double suma;

   inicializar(arrayint, nElementos);
   if(leer(arrayint, nElementos)){
      mostrar(arrayint, nElementos);
      suma=sumar(arrayint, nElementos);
      cout << "Media="  << suma/nElementos << endl;
      liberar(arrayint, nElementos);
   }
   else
      cerr << "Array no válido" << endl;
}
```

Ejemplo (media de un array): solución directa sin módulos

Ventajas del diseño anterior

- El programa es **más fácil de desarrollar**
- El programa es **más fácil de mantener**
  - Es más sencillo de entender
  - Es más fácil localizar posibles errores
  - Es más sencillo de mejorar el código, localizando las partes que requieren más recursos
  - Es más sencillo incorporar nuevas funcionalidades
- El programa es **más fácil de reutilizar**

Ejemplo (media de un array): solución directa sin módulos

Ventajas del diseño anterior

- El programa es **más fácil de desarrollar**
- El programa es **más fácil de mantener**
  - Es más sencillo de entender
  - Es más fácil localizar posibles errores
  - Es más sencillo de mejorar el código, localizando las partes que requieren más recursos
  - Es más sencillo incorporar nuevas funcionalidades
- El programa es **más fácil de reutilizar**

Ejemplo (media de un array): solución directa sin módulos

Ventajas del diseño anterior

- El programa es **más fácil de desarrollar**
- El programa es **más fácil de mantener**
  - Es más sencillo de entender
  - Es más fácil localizar posibles errores
  - Es más sencillo de mejorar el código, localizando las partes que requieren más recursos
  - Es más sencillo incorporar nuevas funcionalidades
- El programa es **más fácil de reutilizar**

Ejemplo (media de un array): solución directa sin módulos

Ventajas del diseño anterior

- El programa es **más fácil de desarrollar**
- El programa es **más fácil de mantener**
  - Es más sencillo de entender
  - Es más fácil localizar posibles errores
  - Es más sencillo de mejorar el código, localizando las partes que requieren más recursos
  - Es más sencillo incorporar nuevas funcionalidades
- El programa es **más fácil de reutilizar**

Ejemplo (media de un array): solución directa sin módulos

Ventajas del diseño anterior

- El programa es **más fácil de desarrollar**
- El programa es **más fácil de mantener**
  - Es más sencillo de entender
  - Es más fácil localizar posibles errores
  - Es más sencillo de mejorar el código, localizando las partes que requieren más recursos
  - Es más sencillo incorporar nuevas funcionalidades
- El programa es **más fácil de reutilizar**

Ejemplo (media de un array): solución directa sin módulos

Ventajas del diseño anterior

- El programa es **más fácil de desarrollar**
- El programa es **más fácil de mantener**
  - Es más sencillo de entender
  - Es más fácil localizar posibles errores
  - Es más sencillo de mejorar el código, localizando las partes que requieren más recursos
  - Es más sencillo incorporar nuevas funcionalidades
- El programa es **más fácil de reutilizar**

Ejemplo (media de un array): solución directa sin módulos

Ventajas del diseño anterior

- El programa es **más fácil de desarrollar**
- El programa es **más fácil de mantener**
  - Es más sencillo de entender
  - Es más fácil localizar posibles errores
  - Es más sencillo de mejorar el código, localizando las partes que requieren más recursos
  - Es más sencillo incorporar nuevas funcionalidades
- El programa es **más fácil de reutilizar**

Contenido del tema

Abstracción de datos

Inconveniente del diseño anterior

Es un diseño difícil de **modificar**: En caso de necesitar cambiar la representación del array dinámico, será necesario modificar la interfaz del módulo construido (parámetros de las funciones) y por tanto, los programas que lo usan.

Ejemplo: representación alternativa para el array dinámico

Podría interesarnos una nueva representación para el array dinámico que es más eficiente si necesitamos redimensionar continuamente:

```
int *arrayint;
int nElementos;  // número de elementos que hay actualmente en el array
int reservados;  // capacidad que tiene actualmente el array
```

Este cambio llevaría a modificar los parámetros de las funciones del módulo `Arraydin_int` y por tanto de todos los proyectos que lo usasen.

Abstracción de datos

Inconveniente del diseño anterior

Es un diseño difícil de **modificar**: En caso de necesitar cambiar la representación del array dinámico, será necesario modificar la interfaz del módulo construido (parámetros de las funciones) y por tanto, los programas que lo usan.

Ejemplo: representación alternativa para el array dinámico

Podría interesarnos una nueva representación para el array dinámico que es más eficiente si necesitamos redimensionar continuamente:

```
int *arrayint;
int nElementos;  // número de elementos que hay actualmente en el array
int reservados;  // capacidad que tiene actualmente el array
```

Este cambio llevaría a modificar los parámetros de las funciones del módulo `Arraydin_int` y por tanto de todos los proyectos que lo usasen.

Abstracción de datos

Inconveniente del diseño anterior

Es un diseño difícil de **modificar**: En caso de necesitar cambiar la representación del array dinámico, será necesario modificar la interfaz del módulo construido (parámetros de las funciones) y por tanto, los programas que lo usan.

Ejemplo: representación alternativa para el array dinámico

Podría interesarnos una nueva representación para el array dinámico que es más eficiente si necesitamos redimensionar continuamente:

```
int *arrayint;
int nElementos;  // número de elementos que hay actualmente en el array
int reservados;  // capacidad que tiene actualmente el array
```

Este cambio llevaría a modificar los parámetros de las funciones del módulo `Arraydin_int` y por tanto de todos los proyectos que lo usasen.

Abstracción de datos

Abstracción de datos

La **solución** es crear un módulo que oculte la representación de los datos (además de los detalles de las operaciones).
La **abstracción de datos** consiste en definir un nuevo tipo de dato de forma que los programas que lo usan, no dependan de la representación interna de ese tipo de dato.

Ejemplo (media de un array): usando Abstracción de Datos

Tipo de dato ArrayDinamico

```
#ifndef _ARRAYDIN_INT_H
#define _ARRAYDIN_INT_H

struct ArrayDinamico{
  int *arrayint;
  int nElementos;
};

void inicializar(ArrayDinamico &arrayint);
void redimensionar (ArrayDinamico &arrayint, int aumento);
void liberar(ArrayDinamico &arrayint);
bool leer(ArrayDinamico &arrayint);
void mostrar(const ArrayDinamico &arrayint);
double sumar(const ArrayDinamico &arrayint);
#endif
```

Ejemplo (media de un array): usando Abstracción de Datos

```
#include <iostream>
#include "Arraydin_int.h"

using namespace std;

int main(int argc, char* argv[]){
   ArrayDinamico arrayint;
   double suma;

   inicializar(arrayint);
   if(leer(arrayint)){
      mostrar(arrayint);
      suma=sumar(arrayint);
      cout << "Media="  << suma/nElementos << endl;
      liberar(arrayint);
   }
   else
      cerr << "Array no válido" << endl;
}
```

Ejemplo (media de un array): usando Abstracción de Datos

Cambio de la representación del Tipo de Dato ArrayDinamico

Si alguna vez, cambiamos la representación de los datos del ArrayDinamico, **no será necesario** cambiar los proyectos que lo usan ya que los parámetros de las funciones del módulo no necesitan cambiarse:

```
#ifndef _ARRAYDIN_INT_H
#define _ARRAYDIN_INT_H

struct ArrayDinamico{
  int *arrayint;
  int nElementos;
  int reservados;
};

void inicializar(ArrayDinamico &arrayint);
void redimensionar (ArrayDinamico &arrayint, int aumento);
void liberar(ArrayDinamico &arrayint);
bool leer(ArrayDinamico &arrayint);
void mostrar(const ArrayDinamico &arrayint):
double sumar(const ArrayDinamico &arrayint);
#endif
```

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).

Tipos de datos abstractos: struct y class

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 struct y class consiste en que por defecto los datos miembro son públicos en struct, mientras que en class (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
}
```

Tipos de datos abstractos: struct y class

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 struct y class consiste en que por defecto los datos miembro son públicos en struct, mientras que en class (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
}
```

Tipos de datos abstractos: struct y class

¿Datos privados en struct?

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

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

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

Tipos de datos abstractos: struct y class

¿Métodos en struct?

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 solo para agrupar datos.

Tipos de datos abstractos: struct y class

¿Métodos en struct?

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 solo para agrupar datos.

Tipos de datos abstractos: struct y class

Inconvenientes de definir TDAs con struct

Los tipos de datos abstractos que se definen con struct no evitan que los usuarios del tipo de dato, usen directamente la representación interna del tipo de dato (no hay **ocultamiento** de la representación), lo cual puede provocar:

- Hacer que un objeto quede en **estado inconsistente**.
- Dificultad de modificaciones futuras del tipo de dato abstracto.

Tipos de datos abstractos: struct y class

Inconvenientes de definir TDAs con struct

Los tipos de datos abstractos que se definen con struct no evitan que los usuarios del tipo de dato, usen directamente la representación interna del tipo de dato (no hay **ocultamiento** de la representación), lo cual puede provocar:

- Hacer que un objeto quede en **estado inconsistente**.
- **Dificultad de modificaciones** futuras del tipo de dato abstracto.

Tipos de datos abstractos: struct y class

Objeto en estado inconsistente: Tipo de dato abstracto Fecha con struct

```
#ifndef _FECHA_H_
#define _FECHA_H_
struct Fecha{
   int dia, mes, anio;
};

void setFecha(Fecha &fecha, int day, int month, int year);
...
#endif
```

```
#include <iostream>
#include "Fecha.h"

int main(int argc, char* argv[]){
   Fecha fecha;

   fecha.dia=29;
   fecha.mes=2;
   fecha.anio=2018; // fecha queda en estado inconsistente
   ...
}
```

Tipos de datos abstractos: struct y class

TDA difícil de modificar: Tipo de dato abstracto TCoordenada con struct

Supongamos que representamos un punto usando sus coordenadas cartesianas.

```
#ifndef _TCOORDENADA_H_
#define _TCOORDENADA_H_
struct TCoordenada {
    double x,y;
};
void setCoordenadas(TCoordenada &c, double cx, double cy);
double getY(TCoordenada c);
double getX(TCoordenada c);
...
#endif
```

```
#include <iostream>
#include "TCoordenada.h"

int main(int argc, char* argv[]){
   TCoordenada p1;
   setCoordenadas(p1,5,10);
   cout<<"x="<<p1.x<<", y="<<p1.y<<endl; // uso directo de la representación: problemático
   ...
}
```

Tipos de datos abstractos: struct y class

TDA difícil de modificar: Tipo de dato abstracto TCoordenada con struct

En el futuro, decidimos cambiar la representación usando coordenadas polares: provoca problemas en los proyectos que usan este TDA.

```
#ifndef _TCOORDENADA_H_
#define _TCOORDENADA_H_
struct TCoordenada { // punto en coordenadas polares
    double r; // módulo
    double alfa; // ángulo con eje X
};
void setCoordenadas(TCoordenada &c, double cx, double cy);
double getY(TCoordenada c);
double getX(TCoordenada c);
...
#endif
```

```
#include <iostream>
#include "TCoordenada.h"

int main(int argc, char* argv[]){
   TCoordenada p1;
   setCoordenadas(p1,5,10);
   cout<<"x="<<p1.x<<", y="<<p1.y<<endl; // ERROR DE COMPILACIÓN: debemos cambiar para adaptarnos
                                         // a la modificación del TDA
   ...
}
```

Tipos de datos abstractos: struct y class

TDA con class

Los tipos de datos abstractos que se definen con class impiden que los programas que usan el tipo, accedan directamente a su representación, evitando los problemas que ello conlleva.
Diseñar una clase implica diseñar dos partes (estrechamente relacionadas):

- **Parte interna**, especialmente la representación del tipo.
- **Interfaz**: establece un contrato de uso que no puede romperse (si cambiamos la interfaz, la nueva implementación dejará de ser compatible con los módulos que usaban el tipo).

Tipos de datos abstractos: struct y class

TDA con class

Los tipos de datos abstractos que se definen con class impiden que los programas que usan el tipo, accedan directamente a su representación, evitando los problemas que ello conlleva.
Diseñar una clase implica diseñar dos partes (estrechamente relacionadas):

- **Parte interna**, especialmente la representación del tipo.
- **Interfaz**: establece un contrato de uso que no puede romperse (si cambiamos la interfaz, la nueva implementación dejará de ser compatible con los módulos que usaban el tipo).

Tipos de datos abstractos: struct y class

```
#ifndef _TCOORDENADA_H_
#define _TCOORDENADA_H_
class TCoordenada {
    double x,y;
public:
    void setCoordenadas(double cx, double cy);
    double getY();
    double getX();
    ...
};
#endif
```

```
#include <iostream>
#include "TCoordenada.h"
int main(int argc, char* argv[]){
     TCoordenada p1;
     p1.setCoordenadas(5,10);
     cout<<"x="<<p1.getX()<<", y="<<p1.getY()<<endl; // Ahora no podemos acceder directamente a la
                                                     // representación interna del TDA
     ...
}
```

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.
- Los datos miembro que usaremos para representar este tipo de dato son:
  - Grado del polinomio (0 si p(x)==0)
  - Coeficientes: lista con los coeficientes de cada monomio.
- Algunas operaciones que podrían definirse son: Suma, Multiplicacion, Derivada, ...

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.
- Los datos miembro que usaremos para representar este tipo de dato son:
  - Grado del polinomio (0 si p(x)==0)
  - Coeficientes: lista con los coeficientes de cada monomio.
- Algunas operaciones que podrían definirse son: Suma, Multiplicacion, Derivada, ...

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.
- Los datos miembro que usaremos para representar este tipo de dato son:
  - Grado del polinomio (0 si p(x)==0)
  - Coeficientes: lista con los coeficientes de cada monomio.
- Algunas operaciones que podrían definirse son: Suma, Multiplicacion, Derivada, ...

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.
- Los datos miembro que usaremos para representar este tipo de dato son:
  - Grado del polinomio (0 si p(x)==0)
  - Coeficientes: lista con los coeficientes de cada monomio.
- Algunas operaciones que podrían definirse son: Suma, Multiplicacion, Derivada, ...

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.
- Los datos miembro que usaremos para representar este tipo de dato son:
  - Grado del polinomio (0 si p(x)==0)
  - Coeficientes: lista con los coeficientes de cada monomio.
- Algunas operaciones que podrían definirse son: Suma, Multiplicacion, Derivada, ...

La clase Polinomio

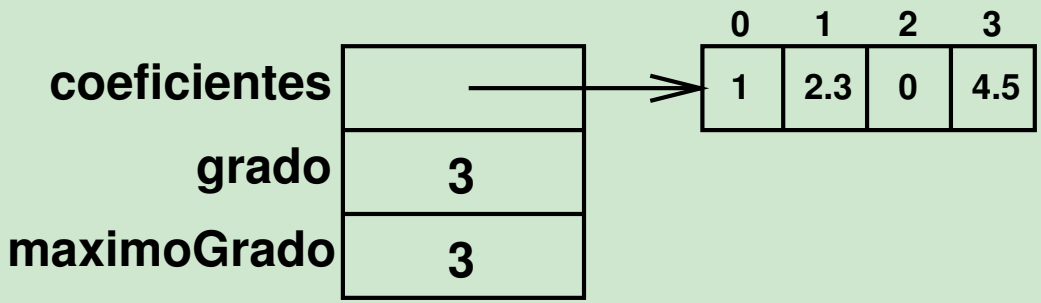
TDA Polinomio

- 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.
- Los datos miembro que usaremos para representar este tipo de dato son:
  - Grado del polinomio (0 si p(x)==0)
  - Coeficientes: lista con los coeficientes de cada monomio.
- Algunas operaciones que podrían definirse son: Suma, Multiplicacion, Derivada, ...

La clase Polinomio

Implementación del TDA Polinomio: datos miembro (parte interna)

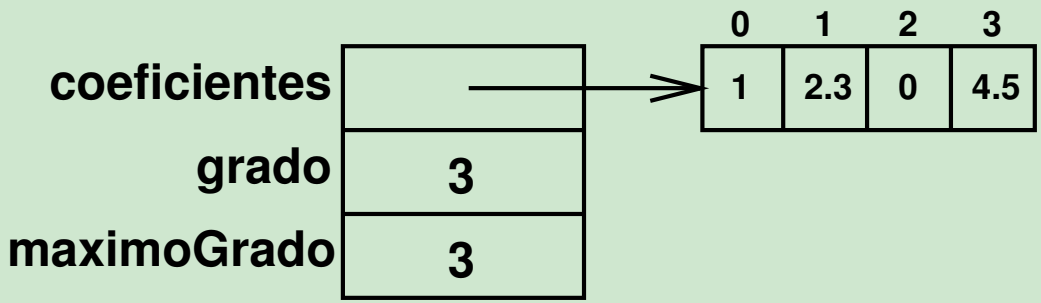
- **coeficientes**: ***array dinámico*** con los coeficientes, que permite polinomios de cualquier grado. Se requiere siempre que tenga una dimensión de **al menos un elemento**.
- **grado**: grado del polinomio. Es necesario para conocer qué parte del array dinámico estamos usando.
  - Vale 0 para un polinomio nulo.
- **maximoGrado**: indica el **máximo grado** posible (capacidad del array), es decir, el tamaño concreto del array de coeficientes.



La clase Polinomio

Implementación del TDA Polinomio: datos miembro (parte interna)

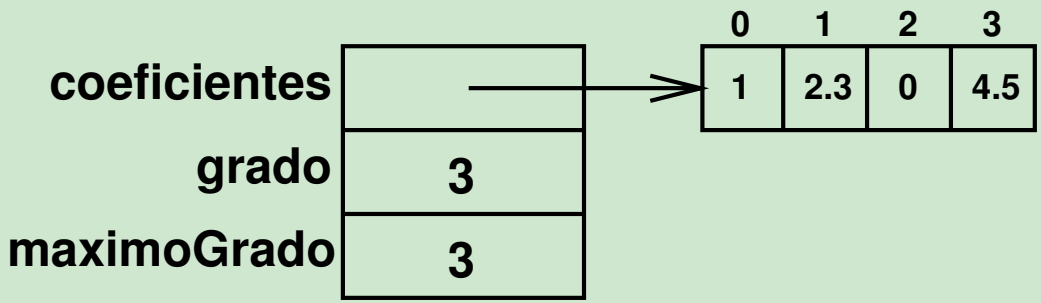
- **coeficientes**: ***array dinámico*** con los coeficientes, que permite polinomios de cualquier grado. Se requiere siempre que tenga una dimensión de **al menos un elemento**.
- **grado**: grado del polinomio. Es necesario para conocer qué parte del array dinámico estamos usando.
  - Vale 0 para un polinomio nulo.
- **maximoGrado**: indica el **máximo grado** posible (capacidad del array), es decir, el tamaño concreto del array de coeficientes.



La clase Polinomio

Implementación del TDA Polinomio: datos miembro (parte interna)

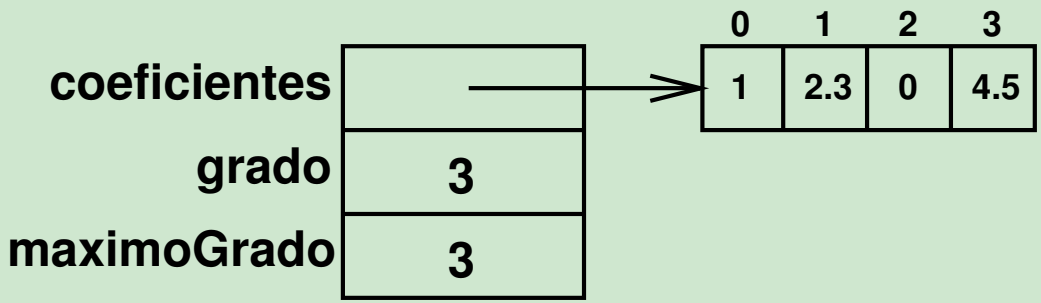
- **coeficientes**: ***array dinámico*** con los coeficientes, que permite polinomios de cualquier grado. Se requiere siempre que tenga una dimensión de **al menos un elemento**.
- **grado**: grado del polinomio. Es necesario para conocer qué parte del array dinámico estamos usando.
  - Vale 0 para un polinomio nulo.
- **maximoGrado**: indica el **máximo grado** posible (capacidad del array), es decir, el tamaño concreto del array de coeficientes.



La clase Polinomio

Implementación del TDA Polinomio: datos miembro (parte interna)

- **coeficientes**: ***array dinámico*** con los coeficientes, que permite polinomios de cualquier grado. Se requiere siempre que tenga una dimensión de **al menos un elemento**.
- **grado**: grado del polinomio. Es necesario para conocer qué parte del array dinámico estamos usando.
  - Vale 0 para un polinomio nulo.
- **maximoGrado**: indica el **máximo grado** posible (capacidad del array), es decir, el tamaño concreto del array de coeficientes.



La clase Polinomio

Algunas consideraciones sobre los datos miembro de Polinomio

- El dato miembro **grado** podría eliminarse, pues podría obtenerse a partir del array **maximoGrado**. Lo mantenemos por razones de eficiencia.
- El dato miembro **maximoGrado** se hace necesario al implementar la clase mediante memoria dinámica. ¿Sería necesario si hubiéramos usado la clase **vector**?

La clase Polinomio

Algunas consideraciones sobre los datos miembro de Polinomio

- El dato miembro **grado** podría eliminarse, pues podría obtenerse a partir del array **maximoGrado**. Lo mantenemos por razones de eficiencia.
- El dato miembro **maximoGrado** 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;

     //Máximo grado posible: limitación debida a la implementación
     //de la clase: el array de coeficientes tiene un tamaño limitado
     int maximoGrado;

  public:

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

#endif
```

La clase Polinomio

Implementación del TDA Polinomio: métodos (interfaz de la clase)

Conviene seguir el siguiente orden al definir los métodos de una clase:

- Constructores
- Operaciones naturales sobre los objetos de la clase (deberían ser métodos públicos)
- Otros métodos auxiliares que resulten convenientes (bien por la forma en que se ha hecho la implementación, bien por seguir el principio de descomposición modular....). A menudo, estos métodos serán privados.

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

La clase Polinomio

Implementación del TDA Polinomio: métodos (interfaz de la clase)

Conviene seguir el siguiente orden al definir los métodos de una clase:

- Constructores
- Operaciones naturales sobre los objetos de la clase (deberían ser métodos públicos)
- Otros métodos auxiliares que resulten convenientes (bien por la forma en que se ha hecho la implementación, bien por seguir el principio de descomposición modular....). A menudo, estos métodos serán privados.

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

La clase Polinomio

Implementación del TDA Polinomio: métodos (interfaz de la clase)

Conviene seguir el siguiente orden al definir los métodos de una clase:

- Constructores
- Operaciones naturales sobre los objetos de la clase (deberían ser métodos públicos)
- Otros métodos auxiliares que resulten convenientes (bien por la forma en que se ha hecho la implementación, bien por seguir el principio de descomposición modular....). A menudo, estos métodos serán privados.

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

La clase Polinomio

Implementación del TDA Polinomio: métodos (interfaz de la clase)

Conviene seguir el siguiente orden al definir los métodos de una clase:

- Constructores
- Operaciones naturales sobre los objetos de la clase (deberían ser métodos públicos)
- Otros métodos auxiliares que resulten convenientes (bien por la forma en que se ha hecho la implementación, bien por seguir el principio de descomposición modular....). A menudo, estos métodos serán 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

Constructores de una clase

Los constructores se encargan de inicializar de forma conveniente los datos miembro.
En clases como **Polinomio**, 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 (un dato miembro no inicializado probablemente contendrá un valor basura).
  - Solo llama al constructor por defecto de cada dato miembro que sea un objeto de otra clase.
- Definiéndolo **explícitamente** en la clase.

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 (un dato miembro no inicializado probablemente contendrá un valor basura).
  - Solo llama al constructor por defecto de cada dato miembro que sea un objeto de otra clase.
- Definiéndolo **explícitamente** en la clase.

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 (un dato miembro no inicializado probablemente contendrá un valor basura).
  - Solo llama al constructor por defecto de cada dato miembro que sea un objeto de otra clase.
- Definiéndolo **explícitamente** en la clase.

Constructor por defecto

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

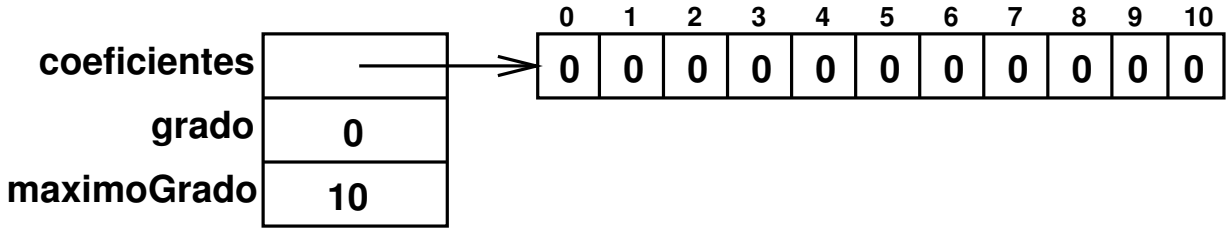
- El compilador lo crea **implícitamente** cuando la clase no define ningún constructor.
  - Tal constructor no inicializa los datos miembro de la clase (un dato miembro no inicializado probablemente contendrá un valor basura).
  - Solo llama al constructor por defecto de cada dato miembro que sea un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

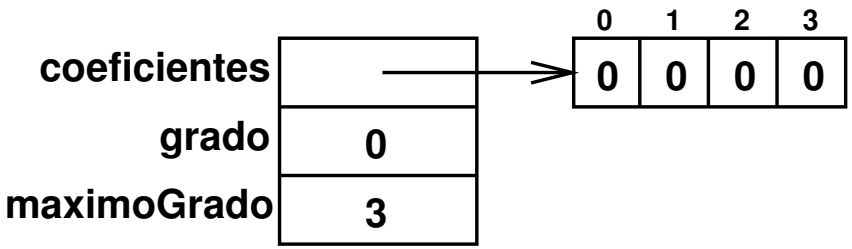
```
/**
 * Constructor por defecto de la clase. El trabajo de este
 * constructor se limita a crear un objeto nuevo, con
 * capacidad máxima para guardar once coeficientes
 */
Polinomio::Polinomio(){
   // Se inicializan los datos miembro maximoGrado y grado
   maximoGrado=10;
   grado=0;

   // 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;
   }
}
```

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

int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    ...
}
```

Los constructores de la clase Polinomio

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 máximo grado posible
 * @param maximoGrado valor del grado máximo
 */
Polinomio::Polinomio(int maximoGrado){
   // Si máximo 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;
   }
}
```

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

int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    Polinomio pol2(20); // Polinomio creado con constructor Polinomio(int)
    ...
}
```

Constructores

Código común en constructores

A menudo, varios constructores comparten un trozo de código, cuya repetición puede evitarse con un método auxiliar que habitualmente 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

Método auxiliar usado en los constructores de Polinomio

Añadimos el método privado inicializar() a la clase Polinomio.

```
/**
 * Método privado para inicialar el valor de grado y para
 * crear array de coeficientes de tamaño dado por el valor
 * de maximoGrado (más uno), poniéndolos 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();
}
```

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

int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    Polinomio pol2(20); // Polinomio creado con constructor Polinomio(int)
    ...
}
```

Constructores

De momento, la declaración de la clase contendría:

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

     // Máximo grado posible: limitación debida a la implementación
     //de la clase: el array de coeficientes tiene un tamaño limitado
     int maximoGrado;

     // Método inicializar para facilitar la programación de los constructores
     void inicializar();

  public:
     Polinomio(); // Constructor por defecto
     Polinomio(int gradoMaximo); // Constructor indicando el grado máximo
};

#endif
```

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 máximo grado posible
    * @param maximoGrado valor del grado máximo
    */
   Polinomio(int maximoGrado = 10);
 ...................
}
```

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 máximo grado posible
    * @param maximoGrado valor del grado máximo
    */
   Polinomio(int maximoGrado = 10);
 ...................
}
```

Constructores

```
/**
 * Constructor de la clase indicando el máximo grado posible
 * @param maximoGrado valor del grado máximo
 */
Polinomio::Polinomio(int maximoGrado) {
   // Si máximo 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 demás datos miembro
   inicializar();
}
```

Contenido del tema

Operaciones del interfaz del TDA

Selección de operaciones del interfaz del TDA

Es importante seleccionar bien el conjunto de operaciones de un TDA, teniendo en cuenta tanto el problema que vamos a resolver ahora como los que podríamos resolver en el futuro.

- Debe incluir un número de operaciones mínimo que definan la **funcionalidad básica**.
  O sea, el usuario debería poder resolver los problemas usando estas operaciones, sin necesidad de acceder a la parte interna.
- Es posible incluir otras funciones que sin ser básicas, permitan obtener un tipo mucho más útil (**funcionalidad adicional**).
- Las operaciones pueden implementarse como métodos de la clase o como funciones externas en el tipo de dato abstracto.

Operaciones del interfaz del TDA

Selección de operaciones del interfaz del TDA

Es importante seleccionar bien el conjunto de operaciones de un TDA, teniendo en cuenta tanto el problema que vamos a resolver ahora como los que podríamos resolver en el futuro.

- Debe incluir un número de operaciones mínimo que definan la **funcionalidad básica**.
  O sea, el usuario debería poder resolver los problemas usando estas operaciones, sin necesidad de acceder a la parte interna.
- Es posible incluir otras funciones que sin ser básicas, permitan obtener un tipo mucho más útil (**funcionalidad adicional**).
- Las operaciones pueden implementarse como métodos de la clase o como funciones externas en el tipo de dato abstracto.

Operaciones del interfaz del TDA

Selección de operaciones del interfaz del TDA

Es importante seleccionar bien el conjunto de operaciones de un TDA, teniendo en cuenta tanto el problema que vamos a resolver ahora como los que podríamos resolver en el futuro.

- Debe incluir un número de operaciones mínimo que definan la **funcionalidad básica**.
  O sea, el usuario debería poder resolver los problemas usando estas operaciones, sin necesidad de acceder a la parte interna.
- Es posible incluir otras funciones que sin ser básicas, permitan obtener un tipo mucho más útil (**funcionalidad adicional**).
- Las operaciones pueden implementarse como métodos de la clase o como funciones externas en el tipo de dato abstracto.

Métodos del interfaz básico

Métodos del interfaz básico

- Deberían ser **pocos**: definen la funcionalidad básica.
- Suelen utilizar directamente los datos miembro de la clase.
- En el conjunto de operaciones distinguimos entre *métodos consultores* y *métodos modificadores*.
  IMPORTANTE: No tiene por qué haber un método consultor y modificador por cada dato miembro.

Métodos del interfaz básico

Métodos del interfaz básico

- Deberían ser **pocos**: definen la funcionalidad básica.
- Suelen utilizar directamente los datos miembro de la clase.
- En el conjunto de operaciones distinguimos entre *métodos consultores* y *métodos modificadores*.
  IMPORTANTE: No tiene por qué haber un método consultor y modificador por cada dato miembro.

Métodos del interfaz básico

Métodos del interfaz básico

- Deberían ser **pocos**: definen la funcionalidad básica.
- Suelen utilizar directamente los datos miembro de la clase.
- En el conjunto de operaciones distinguimos entre ***métodos consultores*** y ***métodos modificadores***.
  IMPORTANTE: No tiene por qué haber un método consultor y modificador por cada dato miembro.

Métodos del interfaz adicional

Métodos del interfaz adicional

- Facilitan el uso del tipo de dato abstracto.
- No deberían extenderse demasiado.
- Aunque se implementen como 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 del interfaz adicional

Métodos del interfaz adicional

- Facilitan el uso del tipo de dato abstracto.
- No deberían extenderse demasiado.
- Aunque se implementen como 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 del interfaz adicional

Métodos del interfaz adicional

- Facilitan el uso del tipo de dato abstracto.
- No deberían extenderse demasiado.
- Aunque se implementen como 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.

Ejemplo: operaciones del TDA Polinomio

TDA Polinomio

En la clase **Polinomio** los datos miembro de interés son el **grado** y los **coeficientes** de cada término.

- Para consultar su valor se precisan los métodos consultores siguientes:
  - **obtenerGrado**: obtiene el grado del polinomio.
  - **obtenerCoeficiente**: obtiene el coeficiente asociado a un determinado término.
- Para modificar un coeficiente implementaremos el método modificador asignarCoeficiente

Ejemplo: operaciones del TDA Polinomio

TDA Polinomio

En la clase **Polinomio** los datos miembro de interés son el **grado** y los **coeficientes** de cada término.

- Para consultar su valor se precisan los métodos consultores siguientes:
  - **obtenerGrado**: obtiene el grado del polinomio.
  - **obtenerCoeficiente**: obtiene el coeficiente asociado a un determinado término.
- Para modificar un coeficiente implementaremos el método modificador asignarCoeficiente

Ejemplo: operaciones del TDA Polinomio

TDA Polinomio

En la clase **Polinomio** los datos miembro de interés son el **grado** y los **coeficientes** de cada término.

- Para consultar su valor se precisan los métodos consultores siguientes:
  - **obtenerGrado**: obtiene el grado del polinomio.
  - **obtenerCoeficiente**: obtiene el coeficiente asociado a un determinado término.
- Para modificar un coeficiente implementaremos el método modificador asignarCoeficiente

Ejemplo: operaciones del TDA Polinomio

TDA Polinomio

En la clase **Polinomio** los datos miembro de interés son el **grado** y los **coeficientes** de cada término.

- Para consultar su valor se precisan los métodos consultores siguientes:
  - **obtenerGrado**: obtiene el grado del polinomio.
  - **obtenerCoeficiente**: obtiene el coeficiente asociado a un determinado término.
- Para modificar un coeficiente implementaremos el método modificador **asignarCoeficiente**

Contenido del tema

Métodos const

Métodos const

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

Métodos const

Los métodos consultores no modifican el objeto sobre el que se llaman, por lo que se declararán de forma especial para remarcar esta característica: **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 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 índice es menor o igual que el grado
   if (indice>=0 && indice <= grado){
      salida = coeficientes[indice];
   }
   return salida;
}
```

Métodos const

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

int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    ...
    cout << "Grado del polinomio: " << pol1.obtenerGrado() << endl;
    cout << "Coeficientes del polinomio: "
    for(int i=0;i<=pol1.obtenerGrado(); i++){
        cout << i << ": " << pol1.obtenerCoeficiente(i) << " ";
    }
    cout << endl;
}
```

Contenido del tema

Métodos inline

Método inline

Es un método declarado explícitamente como **inline** o bien definido dentro del ámbito de la declaración de la clase.

- 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 (muy sencillos).

Métodos inline

Método inline

Es un método declarado explícitamente como **inline** o bien definido dentro del ámbito de la declaración de la clase.

- 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 (muy sencillos).

Métodos inline

Métodos inline en clase Polinomio

Los métodos **obtenerGrado()** y **obtenerCoeficiente(int)** pueden hacerse inline ya que son muy simples.
La declaración de la clase quedaría tal y como se indica a continuación (la palabra reservada **inline** es opcional).

```
#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

```
    /**
     * Máximo grado posible: limitación debida a la implementación
     * de la clase: el array de coeficientes tiene un tamaño limitado
     */
    int maximoGrado;

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

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

Métodos inline

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

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

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

Métodos inline

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

#endif
```

Fíjese que el método **obtenerCoeficiente(int)** lo hemos reescrito además de forma **más compacta** que en la versión previa.

Métodos inline

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

#endif
```

Fíjese que el método **obtenerCoeficiente(int)** lo hemos reescrito además de forma **más compacta** que en la versión previa.

Contenido del tema

Otros métodos de la interfaz básica

Métodos modificadores

Modifican el valor de alguno de los datos miembro de un objeto.

Modificadores de la clase Polinomio

No incluiremos un método para modificar el grado, pues este se determinará en base a los coeficientes del polinomio.
Incluieremos un método que permita asignar valores a los coeficientes:

- **asignarCoeficiente(int indice, float coeficiente)**: permite asignar el coeficiente asociado a un determinado término.

Otros métodos de la interfaz básica

Métodos modificadores

Modifican el valor de alguno de los datos miembro de un objeto.

Modificadores de la clase Polinomio

No incluiremos un método para modificar el grado, pues este se determinará en base a los coeficientes del polinomio.

Incluieremos un método que permita asignar valores a los coeficientes:

- **asignarCoeficiente(int indice, float coeficiente)**: permite asignar el coeficiente asociado a un determinado término.

Otros métodos de la interfaz básica

Métodos modificadores

Modifican el valor de alguno de los datos miembro de un objeto.

Modificadores de la clase Polinomio

No incluiremos un método para modificar el grado, pues este se determinará en base a los coeficientes del polinomio.
Incluieremos un método que permita asignar valores a los coeficientes:

- **asignarCoeficiente(int indice, float coeficiente)**: 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 el nuevo coeficiente no es cero, hay que actualizar el grado.
- Si el índice coincide con el máximo grado y el nuevo coeficiente es cero, entonces hay que **determinar el nuevo grado** del polinomio analizando los coeficientes.
- En la situación normal, basta con asignar el correspondiente coeficiente.

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 el nuevo coeficiente no es cero, hay que actualizar el grado.
- Si el índice coincide con el máximo grado y el nuevo coeficiente es cero, entonces hay que **determinar el nuevo grado** del polinomio analizando los coeficientes.
- En la situación normal, basta con asignar el correspondiente coeficiente.

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 el nuevo coeficiente no es cero, hay que actualizar el grado.
- Si el índice coincide con el máximo grado y el nuevo coeficiente es cero, entonces hay que **determinar el nuevo grado** del polinomio analizando los coeficientes.
- En la situación normal, basta con asignar el correspondiente coeficiente.

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 el nuevo coeficiente no es cero, hay que actualizar el grado.
- Si el índice coincide con el máximo grado y el nuevo coeficiente es cero, entonces hay que **determinar el nuevo grado** del polinomio analizando los coeficientes.
- En la situación normal, basta con asignar el correspondiente coeficiente.

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 el nuevo coeficiente no es cero, hay que actualizar el grado.
- Si el índice coincide con el máximo grado y el nuevo coeficiente es cero, entonces hay que **determinar el nuevo grado** del polinomio analizando los coeficientes.
- En la situación normal, basta con asignar el correspondiente coeficiente.

Otros métodos de la interfaz básica

```
void Polinomio::asignarCoeficiente(int i, float c){
   if(i>=0){  // Si el índice del coeficiente es válido
      if(i>maximoGrado){  // Si necesitamos más 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 número máximo grado del polinomio
      }
      coeficientes[i]=c;  // Asignamos el nuevo coeficiente

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

Otros métodos de la interfaz básica

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

int main(){
    Polinomio pol1(2); // Polinomio creado con constructor Polinomio(int)
    pol1.asignarCoeficiente(0, 1.0);
    pol1.asignarCoeficiente(1, 2.3);
    pol1.asignarCoeficiente(3, 4.5); // requiere redimensionamiento
    ...
    cout << "Grado del polinomio: " << pol1.obtenerGrado() << endl;
    cout << "Coeficientes del polinomio: "
    for(int i=0;i<=pol1.obtenerGrado(); i++){
        cout << i << ": " << pol1.obtenerCoeficiente(i) << " ";
    }
    cout << endl;
}
```

Contenido del tema

Métodos de la interfaz adicional

Métodos de la interfaz adicional

Estos métodos pueden implementarse sin acceder directamente a la parte interna de un objeto.

De esta forma, un futuro cambio en la representación interna de la clase no implica un cambio en estos métodos.

Método imprimir() en la clase Polinomio

Permitirá 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.

Métodos de la interfaz adicional

Métodos de la interfaz adicional

Estos métodos pueden implementarse sin acceder directamente a la parte interna de un objeto.
De esta forma, un futuro cambio en la representación interna de la clase no implica un cambio en estos métodos.

Método imprimir() en la clase Polinomio

Permitirá 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.

Métodos de la interfaz adicional

Métodos de la interfaz adicional

Estos métodos pueden implementarse sin acceder directamente a la parte interna de un objeto.
De esta forma, un futuro cambio en la representación interna de la clase no implica un cambio en estos métodos.

Método imprimir() en la clase Polinomio

Permitirá 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.

Métodos de la interfaz adicional

```
void Polinomio::imprimir() const{
  cout<<obtenerCoeficiente(obtenerGrado()); //Imprimir término de grado mayor
  if(obtenerGrado()>0)
    cout<<"x^"<<obtenerGrado();
  for(int i=obtenerGrado()-1;i>=0;--i){   //Recorrer el resto de términos
    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;
}
```

Otros métodos de la interfaz básica

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

int main(){
    Polinomio pol1(2); // Polinomio creado con constructor Polinomio(int)
    pol1.asignarCoeficiente(0, 1.0);
    pol1.asignarCoeficiente(1, 2.3);
    pol1.asignarCoeficiente(3, 4.5); // requiere redimensionamiento
    pol1.imprimir();
}
```

```
+ 4.5x^3 + 2.3x + 1
```

Otros métodos de la interfaz básica

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

int main(){
    Polinomio pol1(2); // Polinomio creado con constructor Polinomio(int)
    pol1.asignarCoeficiente(0, 1.0);
    pol1.asignarCoeficiente(1, 2.3);
    pol1.asignarCoeficiente(3, 4.5); // requiere redimensionamiento
    pol1.imprimir();
}
```

```
+ 4.5x^3 + 2.3x + 1
```

Contenido del tema

Puntero this

Puntero this

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

- Puede usarse para:
  - Referenciar un dato miembro del objeto apuntado.
  - Llamar a un método de instancia del objeto apuntado.
- Solo es necesario usarlo en caso de que un dato miembro haya sido ocultado por un parámetro de un método. En otros casos es opcional.

Puntero this

Puntero this

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

- Puede usarse para:
  - Referenciar un dato miembro del objeto apuntado.
  - Llamar a un método de instancia del objeto apuntado.
- Solo es necesario usarlo en caso de que un dato miembro haya sido ocultado por un parámetro de un método. En otros casos es opcional.

Puntero this

Puntero this

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

- Puede usarse para:
  - Referenciar un dato miembro del objeto apuntado.
  - Llamar a un método de instancia del objeto apuntado.
- Solo es necesario usarlo en caso de que un dato miembro haya sido ocultado por un parámetro de un método. En otros casos es opcional.

Puntero this

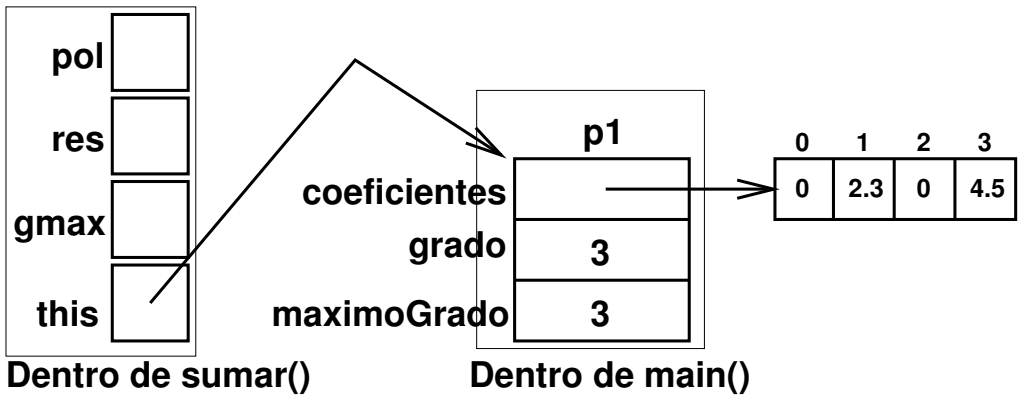
Puntero this

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

- Puede usarse para:
  - Referenciar un dato miembro del objeto apuntado.
  - Llamar a un método de instancia del objeto apuntado.
- Solo es necesario usarlo en caso de que un dato miembro haya sido ocultado por un parámetro de un método. En otros casos es opcional.

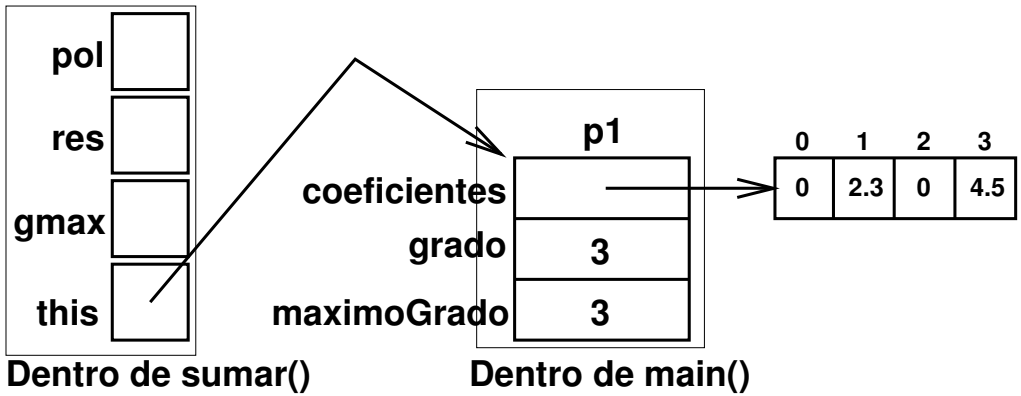
Puntero this

```
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(3), p2;
    p1.asignarCoeficiente(3,4.5);
    p1.asignarCoeficiente(1,2.3);
    ...
    Polinomio p3=p1.sumar(p2);
}
```



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(3),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)

Funciones y clases amigas

Usa clase puede declarar que una función externa u otra clase es amiga usando la palabra reservada **friend**.

Tales funciones y clases amigas podrán acceder a la parte privada la clase.

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

- Los métodos de `B` pueden acceder a la parte privada de `A`.
- La función `funcion()` puede acceder a la parte privada de `A`.

¡Cuidado!

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

Funciones y clases amigas (friend)

Funciones y clases amigas

Usa clase puede declarar que una función externa u otra clase es amiga usando la palabra reservada **friend**.
Tales funciones y clases amigas podrán acceder a la parte privada la clase.

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

- Los métodos de `B` pueden acceder a la parte privada de `A`.
- La función `funcion()` puede acceder a la parte privada de `A`.

¡Cuidado!

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

Funciones y clases amigas (friend)

Funciones y clases amigas

Usa clase puede declarar que una función externa u otra clase es amiga usando la palabra reservada **friend**.
Tales funciones y clases amigas podrán acceder a la parte privada la clase.

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

- Los métodos de `B` pueden acceder a la parte privada de `A`.
- La función `funcion()` puede acceder a la parte privada de `A`.

¡Cuidado!

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

Funciones y clases amigas (friend)

Funciones y clases amigas

Usa clase puede declarar que una función externa u otra clase es amiga usando la palabra reservada **friend**.
Tales funciones y clases amigas podrán acceder a la parte privada la clase.

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

- Los métodos de `B` pueden acceder a la parte privada de `A`.
- La función `funcion()` puede acceder a la parte privada de `A`.

¡Cuidado!

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

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

Ejemplo de uso de 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 }
```

¡Cuidado!

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

Usando la clase Polinomio

Ejemplo de uso de 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 }
```

¡Cuidado!

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

Usando la clase Polinomio

Otro ejemplo de uso de 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); // redimensionar hasta grado 5
7     p1.imprimir();
8 }
```

¡Cuidado!

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

Usando la clase Polinomio

Otro ejemplo de uso de 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); // redimensionar hasta grado 5
7     p1.imprimir();
8 }
```

¡Cuidado!

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

Destrucción automática de objetos locales

Destrucción automática de variables locales

Las variables locales se destruyen automáticamente al finalizar la función en la que se definen, según se ha visto anteriormente.

Destrucción automática de objetos locales

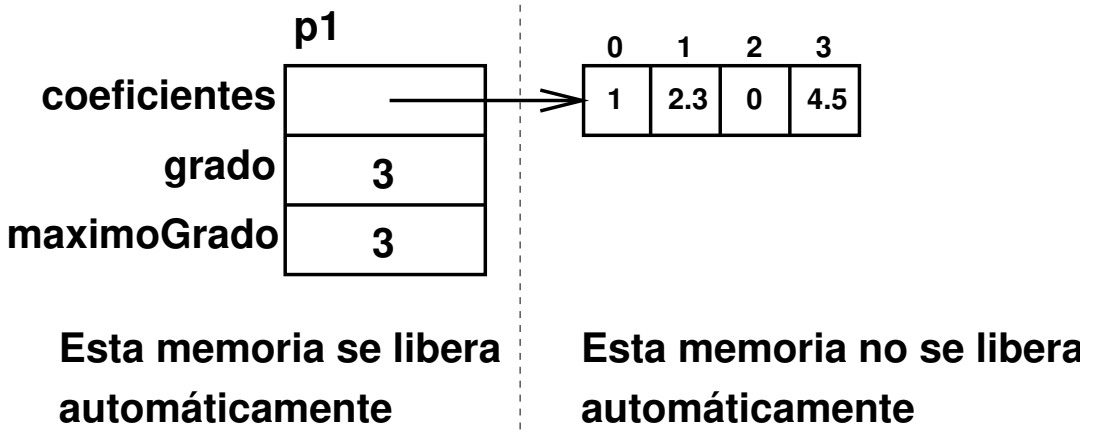
Destrucción automática de variables locales

Las variables locales se destruyen automáticamente al finalizar la función en la que se definen, según se ha visto anteriormente.

Ejemplo

En el siguiente código, `p1` es una variable local de main(): se destruirá automáticamente al acabar `funcion()`.
Pero, ¿qué ocurre con la memoria dinámica reservada por el constructor?.

```
int main() {
    Polinomio p1(3);
    ...
    p1.imprimir();
}
```



¿Cómo liberar la memoria dinámica del objeto?

Una mala solución para liberar la MD de un objeto

Hacer un método para liberar la memoría dinámica del objeto y llamarlo explícitamente antes de que se destruya el objeto.

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

```
int main() {
    Polinomio p1(3);
    ...
    p1.imprimir();
    ...
    p1.liberar();
}
```

El destructor de la clase

Destructor de una clase

Automatiza el proceso de destrucción.

- El destructor es único, no lleva parámetros y no devuelve nada.
- Se ejecuta de forma automática, en el momento de destruir un objeto de la clase:
  - Los objetos 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.
  - Los objetos pasados por valor a un función justo antes de acabar la función.

El destructor de la clase

Destructor de una clase

Automatiza el proceso de destrucción.

- El destructor es único, no lleva parámetros y no devuelve nada.
- Se ejecuta de forma automática, en el momento de destruir un objeto de la clase:
  - Los objetos 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.
  - Los objetos pasados por valor a un función justo antes de acabar la función.

El destructor de la clase

Destructor de una clase

Automatiza el proceso de destrucción.

- El destructor es único, no lleva parámetros y no devuelve nada.
- Se ejecuta de forma automática, en el momento de destruir un objeto de la clase:
  - Los objetos 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.
  - Los objetos pasados por valor a un función justo antes de acabar la función.

El destructor de la clase

Destructor de una clase

Automatiza el proceso de destrucción.

- El destructor es único, no lleva parámetros y no devuelve nada.
- Se ejecuta de forma automática, en el momento de destruir un objeto de la clase:
  - Los objetos 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.
  - Los objetos pasados por valor a un función justo antes de acabar la función.

El destructor de la clase

Destructor de una clase

Automatiza el proceso de destrucción.

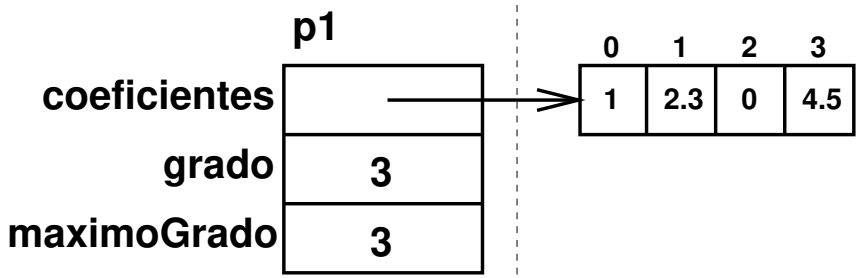
- El destructor es único, no lleva parámetros y no devuelve nada.
- Se ejecuta de forma automática, en el momento de destruir un objeto de la clase:
  - Los objetos locales a una función o trozo de código, justo antes de acabar la función o trozo de código.
  - Los objetos variable global, justo antes de acabar el programa.
  - Los objetos pasados por valor a un función justo antes de acabar la función.

El destructor de la clase

Destructor de la clase Polinomio

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

```
int main() {
    Polinomio p1(3);
    ...
    p1.imprimir();
    ...
} // Aquí se destruirá automáticamente el
  // objeto p1 (MD incluida)
```



Esta memoria se libera automáticamente

Esta memoria la libera el destructor

Ejemplo de llamadas al destructor

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

```
#include <iostream>
using namespace std;
class Cotilla{
    public:
        Cotilla();
        ~Cotilla();
};
Cotilla::Cotilla(){
    cout<<"Constructor"<<endl;
}
Cotilla::~Cotilla(){
    cout<<"Destructor"<<endl;
}
void funcion(){
    Cotilla local;
    cout<<"funcion()"<<endl;
}
Cotilla varGlobal;
int main(){
    cout<<"Comienza main()"<<endl;
    Cotilla ppal;
    cout<<"Antes de llamar a funcion()"<<endl;
    funcion();
    cout<<"Después 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 // Construcción objeto varGlobal
Comienza main() // Inicio ejecucion main()
Constructor // Construcción objeto ppal
Antes de llamar a funcion()
Constructor // Construcción objeto local de funcion()
funcion() // Ejecución de funcion()
Destructor // Se destruye objeto local (en el ámbito de funcion() )
Después de llamar a funcion() // De vuelta en main()
Termina main() // Finaliza ejecución main()
Destructor // Se destruye objeto ppal
Destructor // Se destruye objeto varGlobal
```

Ejemplo de llamadas al destructor

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

```
#include <iostream>
using namespace std;
class Cotilla{
    public:
        Cotilla();
        ~Cotilla();
};
Cotilla::Cotilla(){
    cout<<"Constructor"<<endl;
}
Cotilla::~Cotilla(){
    cout<<"Destructor"<<endl;
}
void funcion(){
    Cotilla local;
    cout<<"funcion()"<<endl;
}
Cotilla varGlobal;
int main(){
    cout<<"Comienza main()"<<endl;
    Cotilla ppal;
    cout<<"Antes de llamar a funcion()"<<endl;
    funcion();
    cout<<"Después 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 // Construcción objeto varGlobal
Comienza main() // Inicio ejecucion main()
Constructor // Construcción objeto ppal
Antes de llamar a funcion()
Constructor // Construcción objeto local de funcion()
funcion() // Ejecución de funcion()
Destructor // Se destruye objeto local (en el ámbito de funcion() )
Después de llamar a funcion() // De vuelta en main()
Termina main() // Finaliza ejecución main()
Destructor // Se destruye objeto ppal
Destructor // Se destruye objeto varGlobal
```

Contenido del tema

Clases con datos miembro de otras clases

Clase con datos miembro de otras clases

Una clase puede contener 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

Clase con datos miembro de otras clases

Una clase puede contener 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

Clase con datos miembro de otras clases

Una clase puede contener 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

```
class Punto {
   double x,y;
   public:
      Punto() {
         cout<<"Ejecutando Punto()"<<endl;
         x=y=0.0;
      };
      Punto(double x, double y) {
         cout<<"Ejecutando Punto(double x, double y)"<<endl;
         this->x=x; this->y=y;
      };
      ~Punto() {
         cout<<"Ejecutando ~Punto()"<<endl;
      };
      void setXY(double x, double y) {this->x=x; 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 Linea {
    Punto p1, p2;
    public:
        Linea();
        ~Linea();
        Punto getP1() const {return p1;};
        Punto getP2() const {return p2;};
        void setP1(const Punto &punto);
        void setP2(const Punto &punto);
        void print() const {cout<<"p1=";p1.print();
                            cout<<"p2=";p2.print();};
};
Linea::Linea()
{ // <-- En este punto se crean p1 y p2
    cout<<"Ejecutando Linea()"<<endl;
    p1.setXY(-1, -1);  // una vez creados les asignamos valores
    p2.setXY(1, 1); // (-1,-1) y (1,1) respectivamente
}
Linea::~Linea()
{
    cout<<"Ejecutando ~Linea()"<<endl;
} // <-- En este punto se destruyen p1 y p2
```

Clases con datos miembro de otras clases

Ejemplo de uso de clase Linea

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

```
int main(int argc, char *argv[]){
    cout<<"Comienza main()"<<endl;
    Linea lin; //<-- Aquí 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 Punto() // creación de objeto lin.p1
Ejecutando Punto() // creación de objeto lin.p2
Ejecutando Linea()
p1=x=-1, y=-1
p2=x=1, y=1
Ejecutando ~Linea()
Ejecutando ~Punto() // destrucción de objeto lin.p2
Ejecutando ~Punto() // destrucción de objeto lin.p1
```

Clases con datos miembro de otras clases

Ejemplo de uso de clase Linea

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

```
int main(int argc, char *argv[]){
    cout<<"Comienza main()"<<endl;
    Linea lin; //<-- Aquí 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 Punto() // creación de objeto lin.p1
Ejecutando Punto() // creación de objeto lin.p2
Ejecutando Linea()
p1=x=-1, y=-1
p2=x=1, y=1
Ejecutando ~Linea()
Ejecutando ~Punto() // destrucción de objeto lin.p2
Ejecutando ~Punto() // destrucción de objeto lin.p1
```

Contenido del tema

Contenido del tema

El constructor de copia por defecto

El constructor de copia por defecto

El constructor de copia por defecto, generado por el compilador, hace una copia de cada dato miembro del objeto original, usando el constructor de copia de cada uno.

Ejemplo de uso del constructor de copia por defecto: clase Linea

Aquí, el constructor de copia por defecto funciona correctamente.

```
double longitud(Linea linea){ // Paso por copia de objeto Linea
    double cateto1=linea.getP1().getX()-linea.getP2().getX();
    double cateto2=linea.getP1().getY()-linea.getP2().getY();
    return sqrt(cateto1*cateto1 + cateto2*cateto2);
}
int main(int argc, char *argv[]){
    Linea lin;
    lin.setP1(Punto(0,0));
    lin.setP2(Punto(10,20));
    cout <<  "Longitud de lin: " << longitud(lin) << endl;
}
```

El constructor de copia por defecto

El constructor de copia por defecto

El constructor de copia por defecto, generado por el compilador, hace una copia de cada dato miembro del objeto original, usando el constructor de copia de cada uno.

Ejemplo de uso del constructor de copia por defecto: clase Linea

Aquí, el constructor de copia por defecto funciona correctamente.

```
double longitud(Linea linea){ // Paso por copia de objeto Linea
    double cateto1=linea.getP1().getX()-linea.getP2().getX();
    double cateto2=linea.getP1().getY()-linea.getP2().getY();
    return sqrt(cateto1*cateto1 + cateto2*cateto2);
}
int main(int argc, char *argv[]){
    Linea lin;
    lin.setP1(Punto(0,0));
    lin.setP2(Punto(10,20));
    cout <<  "Longitud de lin: " << longitud(lin) << endl;
}
```

El constructor de copia por defecto

Ejemplo de uso del constructor de copia por defecto: clase Polinomio

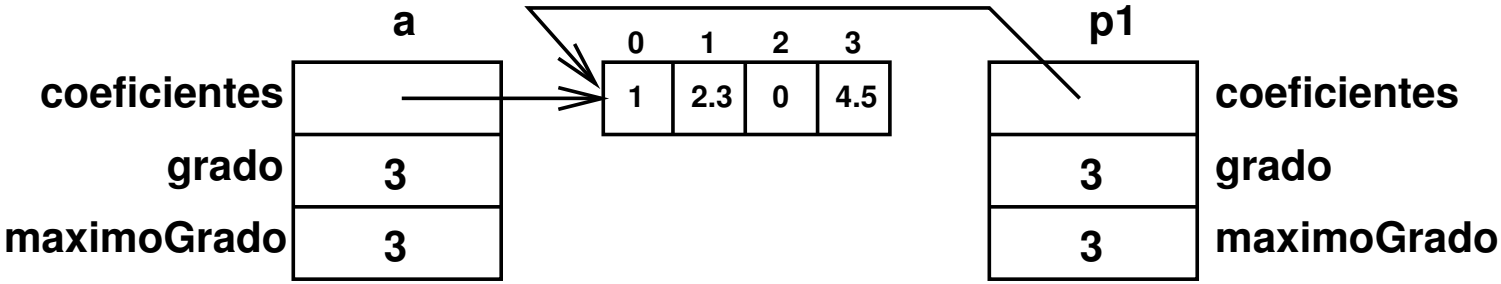
Construyamos una función (externa a la clase) que sume dos polinomios. Aquí, el constructor de copia por defecto **no funciona correctamente**.

```
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, resultado;
    ...
    sumar(a,b,resultado);
}
```

El constructor de copia por defecto

Problema del ejemplo anterior

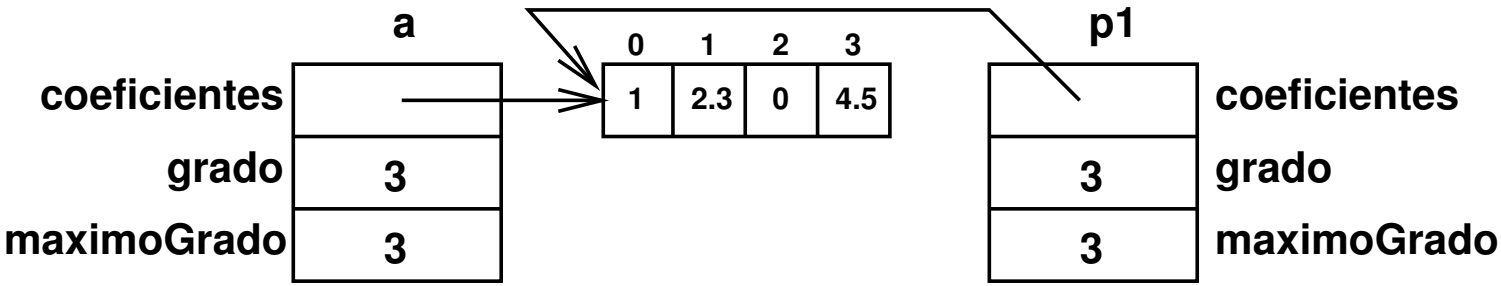
- En la llamada a `sumar(a, b, resultado)` se copian los objetos `a` y `b` en los parámetros formales `p1` y `p2` usando el **constructor de copia por defecto** proporcionado por C++.
- Este constructor hace una copia de cada dato miembro usando el constructor de copia de cada uno: **¿qué problemas da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



El constructor de copia por defecto

Problema del ejemplo anterior

- En la llamada a `sumar(a, b, resultado)` 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

Una mala solución para el ejemplo anterior

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

```
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.asignarCoeficiente(i,p1.obtenerCoeficiente(i)+
                                 p2.obtenerCoeficiente(i));
}
int main(){
    Polinomio a, b, resultado;
    ...
    sumar(a,b,resultado);
}
```

La copia se evita con el paso por referencia

Una mala solución para el ejemplo anterior

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

```
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.asignarCoeficiente(i,p1.obtenerCoeficiente(i)+
                                 p2.obtenerCoeficiente(i));
}
int main(){
    Polinomio a, b, resultado;
    ...
    sumar(a,b,resultado);
}
```

La copia se evita con el paso por referencia

Una mala solución para el ejemplo anterior

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

```
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.asignarCoeficiente(i,p1.obtenerCoeficiente(i)+
                                 p2.obtenerCoeficiente(i));
}
int main(){
    Polinomio a, b, resultado;
    ...
    sumar(a,b,resultado);
}
```

Contenido del tema

Creación de un constructor de copia

Creación de un constructor de copia

Es posible crear nuestro propio constructor de copia que haga una **copia correcta** de un objeto de la clase en otro.

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

- 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.

Creación de un constructor de copia

Creación de un constructor de copia

Es posible crear nuestro propio constructor de copia que haga una **copia correcta** de un objeto de la clase en otro.

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

- 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.

Creación de un constructor de copia

Creación de un constructor de copia

Es posible crear nuestro propio constructor de copia que haga una **copia correcta** de un objeto de la clase en otro.

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

- 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.

Creación de un constructor de copia

Creación de un constructor de copia

Es posible crear nuestro propio constructor de copia que haga una **copia correcta** de un objeto de la clase en otro.

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

- 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.

Creación de un constructor de copia

Solución correcta: implementación del constructor de copia en Polinomio

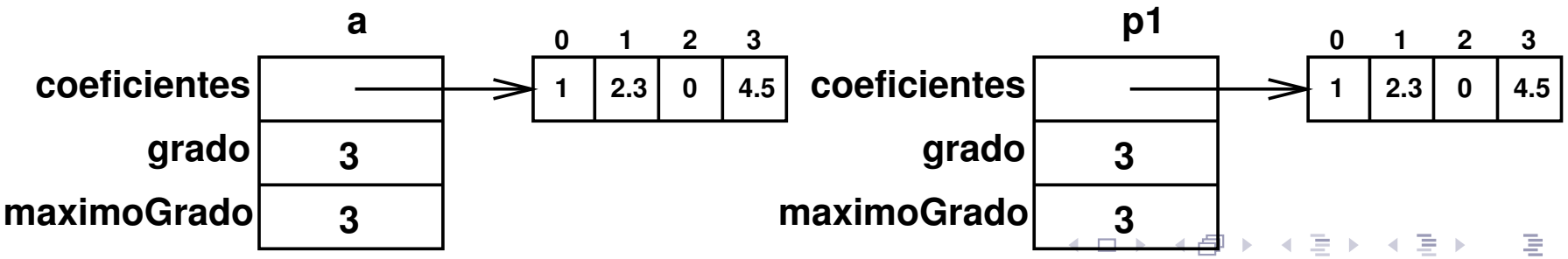
```
class Polinomio {
    ...
    public:
        ...
        Polinomio(const Polinomio &p);
};

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

Ejemplo de uso del constructor de copia de Polinomio

Ahora la copia se hace correctamente.

```
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, resultado;
    ...
    sumar(a,b,resultado);
}
```

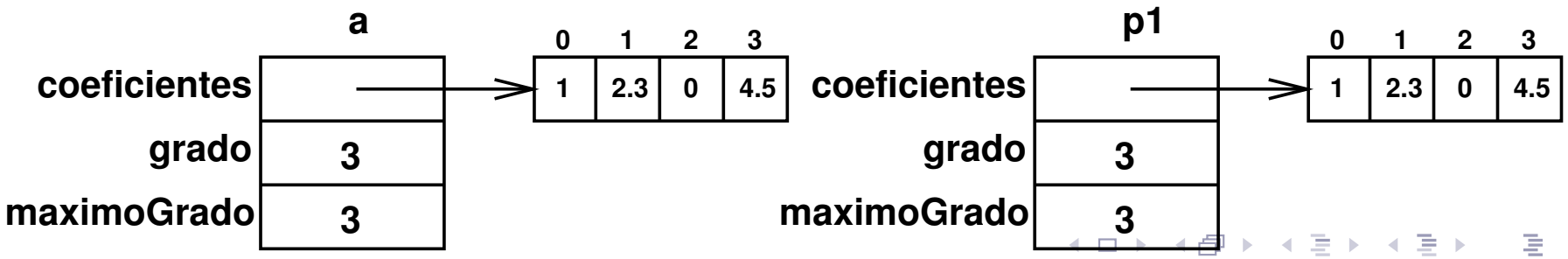


Creación de un constructor de copia

Ejemplo de uso del constructor de copia de Polinomio

Ahora la copia se hace correctamente.

```
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, resultado;
    ...
    sumar(a,b,resultado);
}
```



Contenido del tema

¿Cuándo llama C++ al constructor de copia?

Llamadas al constructor de copia

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

```
int main(){
   Polinomio p1, p2;
   ...
   Polinomio p3(p1); //Copia p1 en p3 usando constructor de copia
   Polinomio p4=p2;  //Copia p2 en p4 usando constructor de copia
}
```

¿Cuándo llama C++ al constructor de copia?

Llamadas al constructor de copia

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

```
int main(){
   Polinomio p1, p2;
   ...
   Polinomio p3(p1); //Copia p1 en p3 usando constructor de copia
   Polinomio p4=p2;  //Copia p2 en p4 usando constructor de copia
}
```

¿Cuándo llama C++ al constructor de copia?

Llamadas al constructor de copia

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

```
/**
 * Devuelve un polinomio con la derivada de un determinado orden
 */
Polinomio Polinomio::derivada(int orden){
   Polinomio derivada; // Creado con constructor por defecto
   ...
   return derivada; // El compilador podría usar (o no) el constructor
}                   //  de copia para devolver derivada
```

¿Cuándo llama C++ al constructor de copia?

Llamadas al constructor de copia

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

```
/**
 * Devuelve un polinomio con la derivada de un determinado orden
 */
Polinomio Polinomio::derivada(int orden){
   Polinomio derivada; // Creado con constructor por defecto
   ...
   return derivada; // El compilador podría usar (o no) el constructor
}                   //  de copia para devolver derivada
```

Contenido del tema

Contenido del tema

Llamadas al declarar variables locales y globales

Llamadas al declarar variables locales y globales

Según vimos anteriormente, el compilador llamará automáticamente a algún constructor al declarar una variable local o global.

El destructor será llamado automáticamente cuando acabe el ámbito en el que está declarada la variable.

```
int main(){
   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
   ...
} // Aquí se llama a los destructores de objetos anteriores
```

Llamadas al declarar variables locales y globales

Llamadas al declarar variables locales y globales

Según vimos anteriormente, el compilador llamará automáticamente a algún constructor al declarar una variable local o global.
El destructor será llamado automáticamente cuando acabe el ámbito en el que está declarada la variable.

```
int main(){
   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
   ...
} // Aquí se llama a los destructores de objetos anteriores
```

Llamadas al declarar variables locales y globales

Llamadas al declarar variables locales y globales

Según vimos anteriormente, el compilador llamará automáticamente a algún constructor al declarar una variable local o global.
El destructor será llamado automáticamente cuando acabe el ámbito en el que está declarada la variable.

```
int main(){
   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
   ...
} // Aquí se llama a los destructores de objetos anteriores
```

Contenido del tema

Llamadas explícitas a un constructor

Llamadas explícitas a un constructor

Consideramos que se hace una llamada explícita cuando escribimos el nombre del constructor junto con los parámetros para crear el objeto que se usará en alguna expresión.

```
int main(){
   Polinomio p1, p2;    // Usa el constructor por defecto
   p1 = Polinomio();    // Crea polinomio con constructor por defecto y
                        // lo asigna a p1 con operator=
   p2 = Polinomio(3);   // Crea polinomio con Polinomio(int) y
                        // lo asigna a p2 con operator=
   ...
}
```

Llamadas explícitas a un constructor

Otro ejemplo

Creación de objeto que hay que devolver con return en una función.

```
Polinomio calcularSplineCubico(const Punto array[], int utiles, int i){
   Polinomio p;
   bool valido=false;

   // ... Calculamos el polinomio p

   if(valido)
      return p;
   else
      return Polinomio(); // si no es válido, se devuelve polinomio nulo
}
```

Contenido del tema

Listas de inicialización en constructores

Nuevo constructor en clase Linea

Añadimos un nuevo constructor a Linea

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

Linea::Linea(const Punto &pun1, const Punto &pun2)
{   // Aquí se crean p1 y p2
    // A continuación se les da el valor inicial
    cout<<"Llamando a Linea::Linea(const Punto &pun1, const Punto &pun2)"<<endl;
    p1.setX(pun1.getX()); // Se iniciacializa p1
    p1.setY(pun1.getY());
    p2.setX(pun2.getX()); // Se iniciacializa p2
    p2.setY(pun2.getY());
}
```

Listas de inicialización en constructores

```
int main(int argc, char *argv[])
{
    cout<<"Comienza main()"<<endl;
    Punto p1,p2;
    p1.setX(10);
    p1.setY(10);
    p2.setX(20);
    p2.setY(20);
    Linea 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

Lista de inicialización

Permite usar el constructor deseado (en lugar del constructor por defecto) para inicializar los datos miembro de una clase

Lista de inicialización en nuevo constructor de clase Linea

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

Listas de inicialización en constructores

Lista de inicialización

Permite usar el constructor deseado (en lugar del constructor por defecto) para inicializar los datos miembro de una clase

Lista de inicialización en nuevo constructor de clase Linea

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

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

Listas de inicialización en constructores

```
int main(int argc, char *argv[])
{
    cout<<"Comienza main()"<<endl;
    Punto p1,p2;
    p1.setX(10);
    p1.setY(10);
    p2.setX(20);
    p2.setY(20);
    Linea 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
```

Contenido del tema

Creación/destrucción de objetos en memoria dinámica

Creación de un array de objetos

Según vimos en tema 1, podemos crear un array de objetos de la misma forma que un array de datos de tipo primitivo.
Cada uno de los objetos será creado usando el **constructor por defecto**.

Destrucción de un array de objetos

Un array de objetos será destruido automáticamente cuando acabe la función donde está declarado **llamando al destructor de cada objeto** del array

```
int main(){
   Polinomio array[1000]; // Crea un array de 1000 polinomios y llama
                          // al constructor por defecto para cada uno
   ...
}  // Llamada al destructor para cada uno de los 1000 polinomios y
   // liberación de la memoria ocupada por el array
```

Contenido del tema

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase.
  - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
   Polinomio *pol;
   pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
   ...
   delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Uso de otros constructores con new

Podemos usar el constructor deseado.

```
int main(){
   Polinomio *pol1, *pol2;
   pol1 = new Polinomio(3); // Reserva de MD para un objeto Polinomio
                            // y llamada al constructor Polinomio(int)
   ...
   pol2 = new Polinomio(*pol1); // Reserva de MD para un objeto Polinomio
                                // y llamada al constructor de copia
   ...
   delete pol1; // Llamada al destructor y liberación de la MD
   delete pol2;
}
```

Creación/destrucción de objetos en memoria dinámica

Array dinámico de objetos

También vimos en el tema 2, que usando los operadores `new[]` y `delete[]` podemos crear y destruir también arrays dinámicos de objetos.

- Operador `new[]`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los objetos del array.
  - Y llama al **constructor** para cada objeto del array.
- Operador `delete[]`:
  - Llama al **destructor** de la clase con cada objeto del array.
  - Y después **libera la memoria** ocupada por el array de objetos.

Creación/destrucción de objetos en memoria dinámica

```
int main(){
   Polinomio *arrayP;
   arrayP = new Polinomio[100]; // Reserva de MD para 100 polinomios y
                                // llamada al constructor por defecto
                                // para cada uno
   ...
   delete[] arrayP; // Llamada al destructor para cada uno de los
                    // 100 polinomios y liberación de la MD ocupada
                    // por los 100 polinomios
}
```

Contenido del tema

Uso de constructores para hacer conversiones implícitas

Uso de constructores para hacer conversiones implícitas

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.

Uso de constructores para hacer conversiones implícitas

```
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
}
```

Uso de constructores para hacer conversiones implícitas

```
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
}
```

Uso de constructores para hacer conversiones implícitas

Especificador explicit

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

Uso de constructores para hacer conversiones implícitas

```
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
}
```

Uso de constructores para hacer conversiones implícitas

```
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
}
```

Uso de constructores para hacer 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
```