

Metodología de la Programación

Tema 4. Clases en C++ (ampliación)

Departamento de Ciencias de la Computación e I.A.

Curso 2015-16





¡Esta es una licencia de Cultura
Libre!



Este obra cuyo autor es mgomez está bajo una
licencia de Reconocimiento-CompartirIgual 4.0
Internacional de Creative Commons.

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
4. Los métodos de la clase
5. Funciones y clases friend
6. Mejorando el uso de la clase
7. El destructor
8. El constructor de copia
9. Llamadas a constructores y destructores

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)
- no incorpora detalles de implementación

Ejemplo: TDA para polinomios.

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

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

Introducción: tipos de datos abstractos

Con respecto a la implementación:

- en C++ pueden implementarse mediante **struct** y **class**
- la principal diferencia entre estas dos posibilidades consiste en que, por defecto, los datos miembro son públicos en **struct**, mientras que en las clases son privados (por defecto)

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

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

Introducción: tipos de datos abstractos

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

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

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

Introducción: tipos de datos abstractos

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`
- las estructuras suelen usarse sólo para agrupar datos.

Introducción: tipos de datos abstractos

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

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

Introducción: tipos de datos abstractos

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

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

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
4. Los métodos de la clase
5. Funciones y clases friend
6. Mejorando el uso de la clase
7. El destructor
8. El constructor de copia
9. Llamadas a constructores y destructores

A modo de ejemplo que guía la exposición del tema construiremos una clase **Lista**, similar a la usada en prácticas. Se precisa también del uso de una clase auxiliar **Celda**. Si pensamos en esta clase auxiliar, sus datos miembro serán:

- valor a almacenar (de tipo real)
- puntero a la siguiente celda

En la clase **Lista** los datos miembro serán:

- longitud de la lista
- puntero a la primera celda

Pero hemos de tener en cuenta que la implementación de la clase se basa en el uso de memoria dinámica para permitir trabajar con listas de cualquier tamaño.

Muchas de las consideraciones que haremos en el tema están relacionadas con las implicaciones que tiene sobre una clase el uso de memoria dinámica

Comenzamos indicando la estructura básica de ambas clases:

```
#ifndef CELDA_H
#define CELDA_H

class Celda{
private:
    double valor;
    Celda * siguiente;
public:
    .....
};

#endif /* CELDA_H */
```

La clase Lista I

Los métodos necesarios para trabajar con la clase **Celda** son los que permitan acceder a los datos miembro (y podrían implementarse como métodos **inline** en el propio archivo de cabecera):

```
Celda(){
    valor=0;
    siguiente=0;
}

inline double obtenerValor() const{
    return valor;
}

inline Celda * obtenerSiguiente() const{
    return siguiente;
}
```



```
inline void asignarValor(double valor){  
    this->valor=valor;  
}
```

```
inline void asignarSiguiete(Celda * siguiente){  
    this->siguiete=siguiente;  
}
```

Para la clase `Lista`:

```
#ifndef LISTA_H
#define LISTA_H

class Lista{
private:
    int longitud;
    Celda * contenido;
public:
    .....
};

#endif /* LISTA_H */
```

Iremos considerando ahora, paso a paso, los diferentes métodos que deberían completar la definición del TDA. Conviene seguir el siguiente orden:

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

Asumimos que cada vez que se agregue un método debe incorporarse a la declaración de la clase correspondiente, en los archivos `Lista.h` o `Celda.h`.

Índice

1. Introducción
2. Clases con datos dinámicos
- 3. Los constructores**
4. Los métodos de la clase
5. Funciones y clases friend
6. Mejorando el uso de la clase
7. El destructor
8. El constructor de copia
9. Llamadas a constructores y destructores

Los constructores de la clase Lista

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

Comenzamos considerando el constructor por defecto: creará una lista vacía, sin elementos.

Constructores: constructor por defecto I

```
Lista::Lista(){  
    // Se asigna 0 a todos los datos miembro  
    longitud=0;  
    contenido=0;  
}
```

Supongamos que se desea permitir la creación de listas de un determinado tamaño, donde todas las celdas estarán inicializadas con el mismo valor. La declaración será:

```
Lista(double valor, int longitud)
```

Constructores: constructor auxiliar I

```
Lista::Lista(double valor, int longitud) {  
    // Se asigna el valor de longitud  
    this->longitud=longitud;  
  
    // Se asigna valor 0 a los punteros auxiliares  
    Celda * pCelda, *pCeldaAnterior=0;  
  
    // Creacion de celdas  
    for(int i=0; i < longitud; i++){  
        // Creacion de nueva celda y asignacion de valor  
        pCelda=new Celda();  
        pCelda->asignarValor(valor);  
    }  
}
```


Constructores: constructor auxiliar II

```
// Se enlaza
if (pCeldaAnterior == 0){
    contenido=pCelda;
}
else{
    pCeldaAnterior->asignarSiguiente(pCelda);
}

// Se avanza anterior, para apuntar a la creada
pCeldaAnterior=pCelda;
}
}
```

Supongamos que ahora interesa disponer de un constructor que permita crear listas para representar rangos de valores: por ejemplo, desde 1 hasta 10, con incremento de 1. La declaración del constructor sería:

```
Lista(int desde, int hasta, int incremento=1)
```

Constructores: constructor auxiliar I

La implementación es ahora:

```
Lista:: Lista(int desde, int hasta, int incremento){
    double valor;
    // Se determina la longitud
    longitud=(hasta-desde)/incremento+1;

    // Se asigna valor 0 a los punteros auxiliares
    Celda * pCelda, *pCeldaAnterior=0;

    // Creacion de celdas
    for(int i=0; i < longitud; i++){
        // Se determina el valor
        valor=desde+i*incremento;
```

Constructores: constructor auxiliar II

```
// Creacion de nueva celda y asignacion de valor
pCelda=new Celda();
pCelda->asignarValor(valor);

// Se enlaza
if (pCeldaAnterior == 0){
    contenido=pCelda;
}
else{
    pCeldaAnterior->asignarSiguiente(pCelda);
}

// Se avanza anterior, para que apunte a la creada
pCeldaAnterior=pCelda;
}
}
```

Constructores: constructor auxiliar I

Se observa en ambos constructores un bloque de código muy similar: encargado de crear las celdas necesarias y asignarles el valor que se desea.

¿Qué hacer?

Se agrega a la clase un método auxiliar (y privado) para hacer esta tarea:

```
Celda * reservarCeldas(int numero, double valor, int incremento=0)
```

Constructores: constructor auxiliar I

```
Celda * Lista::reservarCeldas(int numero, double valor,
                             int incremento){
    // Se asigna valor 0 a pCeldaAnterior
    Celda * pCelda, *pPrimera, *pCeldaAnterior=0;

    // Creacion de celdas
    for(int i=0; i < numero; i++){
        // Creacion de nueva celda y asignacion de valor
        pCelda=new Celda();
        pCelda->asignarValor(valor+i*incremento);
    }
}
```

Constructores: constructor auxiliar II

```
// Se enlaza
if (pCeldaAnterior == 0){
    pPrimera=pCelda;
}
else{
    // Se enlace con la anterior
    pCeldaAnterior->asignarSiguiente(pCelda);
}

// Se avanza anterior, para que apunte a la recién creada
pCeldaAnterior=pCelda;
}

// Se devuelve el puntero a la primera celda
return pPrimera;
}
```

Haciendo uso de este nuevo método podemos simplificar los constructores:

```
Lista::Lista(double valor, int longitud) {  
    // Se asigna el valor de longitud  
    this->longitud=longitud;  
  
    // Se reservan las celdas asignando el valor  
    contenido=0;  
    if (longitud != 0){  
        contenido=reservarCeldas(longitud, valor);  
    }  
}
```


Constructores: constructor auxiliar I

```
Lista:: Lista(int desde, int hasta, int incremento){  
    // Se determina la longitud  
    longitud=(hasta-desde)/incremento+1;  
  
    // Se reservan y asignan las celdas necesarias  
    contenido=0;  
    if (longitud != 0){  
        contenido=reservarCeldas(longitud, desde, incremento);  
    }  
}
```

Índice

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
- 4. Los métodos de la clase**
5. Funciones y clases friend
6. Mejorando el uso de la clase
7. El destructor
8. El constructor de copia
9. Llamadas a constructores y destructores

4. Los métodos de la clase

4.1 Consideraciones generales

4.2 Métodos de la interfaz básica

4.3 Métodos de la interfaz adicional

Los métodos a incluir en cualquier clase se agrupan en **interfaz básica** e **interfaz adicional**.

Con respecto a los métodos de la interfaz básica:

- deberían ser **pocos**, ya que definen la funcionalidad básica
- deberían definir una interfaz **completa**
- suelen utilizar directamente los datos miembro de la clase

En relación a la interfaz adicional:

- pueden ser métodos de la clase o funciones externas
- facilitan el uso del tipo de dato abstracto
- no deberían extenderse demasiado
- aunque sean métodos, no es conveniente que accedan directamente a los datos miembro de la clase, ya que un cambio en la representación del tipo de datos supondría cambiar todos los métodos adicionales

4. Los métodos de la clase

4.1 Consideraciones generales

4.2 Métodos de la interfaz básica

4.3 Métodos de la interfaz adicional

Métodos de la interfaz básica

Los métodos de la interfaz básica deben permitir acceder a los datos miembro de la clase y obtener una representación completa de su estado (de la información que contiene):

- **obtenerLongitud**: permite conocer el número de elementos en la lista
- **obtenerElemento**: da acceso al coeficiente asociado a un determinado término
- **asignarElemento**: permite almacenar un nuevo elemento en la lista
- **insertarElemento**: inserta un elemento en una posición determinada
- **borrarElemento**: elimina el elemento de una posición concreta

Algunos de ellos no modifican el objeto sobre el que se llaman y la implementación mostrará esta circunstancia

La implementación del método `obtenerLongitud` es:

```
inline int obtenerLongitud() const{  
    return longitud;  
}
```

Al ser tan breve se ha declarado como `inline` (por esta razón la implementación puede hacerse en el propio archivo de cabecera). Por no modificar el valor de los datos miembro (el estado del objeto) se ha declarado `const`

El método para obtener el elemento asociado a un determinado índice tiene la siguiente declaración:

```
double obtenerElemento(int indice) const
```

Métodos de la interfaz básica I

Su implementación podría ser la siguiente:

```
double Lista::obtenerElemento(int indice) const{
    // Se comprueba si el indice es correcto
    assert(indice >= 0 && indice < longitud);

    // En caso de ser indice valido, avanzamos hasta llegar a el
    Celda * pCelda=contenido;
    for(int i=1; i <= indice; i++){
        pCelda=pCelda->obtenerSiguiente();
    }

    // Devolvemos el valor almacenado en la celda
    return pCelda->obtenerValor();
}
```

El método de acceso a un valor de la lista se declara de la siguiente forma:

```
void asignarElemento(int indice, double valor)
```

Este método modifica el estado del objeto y no se declara como **const**

Y su implementación:

```
void Lista::asignarElemento(int indice, double valor){  
    // Se comprueba si el indice es correcto  
    assert(indice >= 0 && indice < longitud);  
  
    // En caso de ser valido, avanzamos hasta llegar a el  
    Celda * pCelda=contenido;  
    for(int i=1; i <= indice; i++){  
        pCelda=pCelda->obtenerSiguiente();  
    }  
  
    // Se asigna el valor sobre pCelda  
    pCelda->asignarValor(valor);  
}
```

Métodos de la interfaz básica I

De nuevo, existe un bloque de sentencias común en los dos métodos vistos anteriormente. Creamos nuevo método auxiliar:

```
Celda * Lista::obtenerCelda(int indice) const{
    Celda *pCelda=0;
    // En caso de error, se devuelve null
    if (contenido != 0 && indice >= 0 && indice < longitud){
        // En caso de ser indice valido, avanzamos hasta alcanzarlo
        pCelda=contenido;
        for(int i=1; i <= indice; i++){
            pCelda=pCelda->obtenerSiguiente();
        }
    }
    // Devolvemos el puntero a la Celda
    return pCelda;
}
```

Métodos de la interfaz básica I

Haciendo uso de este método auxiliar podemos simplificar los métodos de obtención y asignación de un elemento:

```
int Lista::obtenerElemento(int indice) const{  
    // Se recupera la celda de interes  
    Celda *pCelda=obtenerCelda(indice);  
  
    // Devolvemos el valor almacenado en la celda  
    assert(pCelda != null);  
    return pCelda->obtenerValor();  
}
```

```
void Lista::asignarElemento(int indice, double valor){  
    // Se recupera la celda de interes  
    Celda *pCelda=obtenerCelda(indice);  
  
    // Se asigna el valor sobre pCelda  
    if (pCelda != 0){  
        pCelda->asignarValor(valor);  
    }  
}
```

El método de inserción de un elemento se declara de la siguiente forma:

```
void insertarElemento(int indice, double valor);
```

Este método modifica el estado del objeto y no se declara como **const**

En cuanto a su implementación:

```
void Lista::insertarElemento(int indice, double valor){
    Celda *pCelda, *pUltima;

    // Si el indice es valido, se asigna el elemento
    if (indice < longitud){
        asignarElemento(indice, valor);
    }
    else{
        // Hay que ampliar la lista, creando tantos elementos
        // como sea necesario
        while(longitud <= indice){
            // Se reservan celdas adicionales, de 10 en 10
            pCelda=reservarCeldas(10, 0);
        }
    }
}
```

```
        // Estas deben engancharse al final
        pUltima=obtenerCelda(longitud-1);
        pUltima->asignarSiguiente(pCelda);

        // Se incrementa el valor de longitud
        longitud=longitud+10;
    }

    // Una vez creadas las celdas necesarias, se asigna
    // el valor correspondiente
    asignarElemento(indice, valor);
}
}
```

El método de borrado de un elemento se declara así:

```
bool borrarElemento(int indice);
```

Este método modifica el estado del objeto y no se declara como `const`

En cuanto a su implementación:

```
bool Lista::borrarElemento(int indice){
    bool resultado=false;

    // Se recupera el elemento a borrar
    Celda *pBorrar=obtenerCelda(indice);

    // Se continua si la celda a borrar es valida
    if (pBorrar != 0){
        // Si es la primera, la siguiente se enlace a contenido
        if (pBorrar == contenido){
            contenido=pBorrar->obtenerSiguiente();
        }
    }
```

Métodos de la interfaz básica II

```
else{  
    // Se busca la anterior  
    Celda *pPrevia=obtenerCelda(indice-1);  
  
    // Se engancha con la siguiente  
    pPrevia->asignarSiguiente(pBorrar->obtenerSiguiente());  
}  
  
// Se borra la celda  
delete pBorrar;  
  
// Se fija el valor de resultado  
resultado=true;  
  
// Se reduce el valor de longitud  
longitud--;
```

```
}  
  
    // Se devuelve resultado  
    return resultado;  
}
```

4. Los métodos de la clase

4.1 Consideraciones generales

4.2 Métodos de la interfaz básica

4.3 Métodos de la interfaz adicional

Métodos de la interfaz adicional

- **mostrar**: muestra por pantalla el contenido del objeto. Al no tratarse de un método de la interfaz básica, se evita que acceda directamente a los datos miembro
- **concatenar**: se agrega al objeto el contenido de la lista pasada como argumento

El método que muestra por pantalla el contenido de la lista será:

```
void Lista::mostrar(){  
    // Se recorre toda la lista  
    Celda *pCelda=obtenerCelda(0);  
  
    // Avance por la lista  
    while(pCelda != 0){  
        cout << pCelda->obtenerValor() << " -> ";  
        pCelda=pCelda->obtenerSiguiente();  
    }  
    cout << "NULL" << endl;  
}
```

Métodos de la interfaz adicional I

Igual ocurre con el método concatenar: al formar parte de la interfaz adicional no debe usar de forma directa los datos miembro de la clase.

```
void Lista::concatenar(const Lista &otra){  
    // Se agregan nuevas celdas a la lista, las necesarias como  
    // para contener todas las de otra  
    Celda *pCelda=reservarCeldas(otra.longitud, 0);  
  
    // Sobre esta celda se copian los valores de otra  
    Celda *pPrimera=pCelda;  
    for(int i=0; i < otra.longitud; i++){  
        pPrimera->asignarValor(otra.obtenerElemento(i));  
  
        // Se adelanta pPrimera  
        pPrimera=pPrimera->obtenerSiguiente();  
    }  
}
```

```
// Ahora se concatenan  
Celda *pUltima=obtenerCelda(longitud-1);  
pUltima->asignarSiguiete(pCelda);  
  
// Se aumenta el valor de longitud  
longitud+=otra.longitud;  
}
```

Índice

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
4. Los métodos de la clase
- 5. Funciones y clases friend**
6. Mejorando el uso de la clase
7. El destructor
8. El constructor de copia
9. Llamadas a constructores y destructores

Funciones y clases amigas (friend)

Puede hacerse que una clase (o función) sea amiga de otra clase (mediante la palabra reservada **friend**). Así podrá acceder a su parte privada.

Deben usarse en contadas ocasiones, ya que se **rompe el encapsulamiento** que proporcionan las clases

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

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

Funciones y clases amigas (friend): Ejemplo

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

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

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

Índice

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
4. Los métodos de la clase
5. Funciones y clases friend
- 6. Mejorando el uso de la clase**
7. El destructor
8. El constructor de copia
9. Llamadas a constructores y destructores

Los constructores de la clase Lista I

Conviene reducir al máximo el número de constructores disponibles en la clase. En este ejemplo, podemos usar uno de los auxiliares, mediante valores por defecto, para que haga el papel de constructor por defecto:

```
Lista(double valor=0, int longitud=0)
.....
Lista::Lista(double valor, int longitud) {
    // Se asigna el valor de longitud
    this->longitud=longitud;

    // Se reservan las celdas asignando el valor
    contenido=0;
    if (longitud != 0){
        contenido=reservarCeldas(longitud, valor);
    }
}
```


Mejorando el uso de la clase Lista: liberación de memoria

Las operaciones sobre la lista (algunas) precisan la reserva de espacio de memoria dinámica, mediante el operador `new`.

Para el funcionamiento correcto de la clase conviene dotarla de un `destructor` para liberar aquella memoria que no sea necesaria

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
4. Los métodos de la clase
5. Funciones y clases friend
6. Mejorando el uso de la clase
- 7. El destructor**
8. El constructor de copia
9. Llamadas a constructores y destructores

Solución:

- un método especial denominado **destructor**
- el destructor es **único** en cada clase, no lleva parámetros y no devuelve nada
- se ejecuta de forma **automática**, al destruir los objetos de la clase:
 - los objetos locales a una función o trozo de código, se destruyen al acabar la función o trozo de código
 - los objetos variable global, justo antes de acabar el programa

El destructor de la clase Lista I

```
Lista::~~Lista() {  
    // Se van obteniendo todas las celdas y se van borrando  
    Celda *pCelda=contenido, *pSiguiente;  
  
    // Recorrido sobre lista  
    while(pCelda != 0){  
        pSiguiente=pCelda->obtenerSiguiente();  
        // Se borra pCelda  
        delete pCelda;  
        // Se adelante  
        pCelda=pSiguiente;  
    }  
    // Ahora se pone todo a cero  
    longitud=0;  
    contenido=0;  
}
```

Índice

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
4. Los métodos de la clase
5. Funciones y clases friend
6. Mejorando el uso de la clase
7. El destructor
- 8. El constructor de copia**
9. Llamadas a constructores y destructores

8. El constructor de copia

8.1 El constructor de copia por defecto

8.2 Creación de un constructor de copia

El constructor de copia por defecto I

Aunque no implementemos el constructor por defecto, `C++` nos proporciona uno, con las siguientes características:

- este constructor hace una copia de cada dato miembro usando el constructor de copia de cada uno de ellos (para los datos de tipo primitivo es inmediato)
- pero, ¿qué ocurre con los datos miembro que precisan de la reserva de memoria dinámica? En este caso no hay reserva de espacio para el nuevo objeto: este se limita a compartir el espacio reservado por el objeto original

8. El constructor de copia

8.1 El constructor de copia por defecto

8.2 Creación de un constructor de copia

Creación de un constructor de copia

La solución a este problema pasa por crear un constructor de copia que:

- haga una copia correcta de un objeto de la clase en otro objeto
- 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 en construcción (**this**)
- se llama automáticamente al hacer un paso por valor para copiar el parámetro actual en el parámetro formal

Creación de un constructor de copia I

```
Lista::Lista(const Lista &otra){
    // Se reservan las celdas necesarias
    contenido=reservarCeldas(otra.obtenerLongitud(),0);

    // Se hace una copia de los valores en otro
    Celda *pPrimera=contenido;
    for(int i=0; i < otra.longitud; i++){
        pPrimera->asignarValor(otra.obtenerElemento(i));

        // Se adelanta pPrimera
        pPrimera=pPrimera->obtenerSiguiente();
    }

    // Se copia el valor de longitud
    longitud=otra.longitud;
}
```

¿Cuándo se llama al constructor de copia?

Situaciones en que se produce la llamada al constructor de copia:

- cuando se pasa un parámetro por valor al llamar a una función o método
- puede llamarse de forma explícita (como creamos objetos en pila o en montón)
- cuando una función devuelve un objeto por valor, mediante la sentencia (**return**)

Índice

1. Introducción
2. Clases con datos dinámicos
3. Los constructores
4. Los métodos de la clase
5. Funciones y clases friend
6. Mejorando el uso de la clase
7. El destructor
8. El constructor de copia
9. Llamadas a constructores y destructores

Llamadas a constructores y destructores I

Suele ser habitual que una clase tenga datos miembro de otras clases. Conviene analizar en este caso el orden que se sigue al crear y destruir objetos de estas clases **compuestas**.

El orden de construcción y destrucción de objetos sigue las siguientes reglas:

- un constructor de una clase:
 - llama al constructor por defecto de cada miembro
 - ejecuta el cuerpo del constructor.
- el destructor de una clase:
 - ejecuta el cuerpo del destructor de la clase del objeto
 - luego llama al destructor de cada dato miembro