Metodología de la Programación

Tema 3. Clases en C++ (ampliación)

Departamento de Ciencias de la Computación e I.A.

Curso 2016-17

¡Esta es una licencia de Cultura Libre!

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 constructures y destructores

Introducción: recordemos

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

Una instancia de una clase es un objeto.

Una clase encapsula:

- **Datos** (datos miembros)
  - Todos los objetos de una misma clase continen esos datos
  - Determinan el estado de dicho objeto
- **Métodos**
  - Todos los objetos de la misma clase comparten estas funciones
  - Determinan el comportamiento de los objetos

Introducción: recordemos

Ocultación de información:

En la declaración de una clase es necesario especificar su ámbito mediante los modificadores de acceso.

- **private**: Sólo se permite el acceso desde los métodos de la clase (por defecto)
- **public**: Los miembros públicos son visibles dentro y fuera del objeto
- **protected**: Se permite su uso en los métodos de la clase y en los dde las clases derivadas mediante herencia.

Introducción: recordemos

Encapsulamiento y ocultación de información permite:

- Usuarios no hagan mal uso de los datos propios de un objeto.
- Separación efectiva representación-implementación.

Gráfico de acceso al objeto.

Introducción: recordemos

Los métodos públicos:

- Cálculos sobre el objeto
- Acceso (lectura/escritura) a los campos del objeto
- Inicialización del objeto (constructor)
- Destrucción del objeto (destructor)

Los métodos privados: funciones auxiliares

- Tareas de petición y liberación de memoria
- Métodos que hacen tareas críticas y no se desean tener acceso desde fuera de la clase

Introducción: tipos de datos abstractos

Tipo de dato abstracto:

- un tipo de dato abstracto (T.D.A.) es una colección de datos (posiblemente de tipos distintos) y un conjunto de operaciones de interés sobre ellos, definidos mediante una *especificación que es independiente de cualquier implementación* (es decir, está especificado a un alto nivel de abstracción)
- no incorpora detalles de implementación

Introducción: tipos de datos abstractos

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;
}
```

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 constructures y destructores

La clase Lista

A modo de ejemplo que guía la exposición del tema construiremos una clase Lista. 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

La clase Lista

En la clase Lista los datos miembro serán:

- longitud de la lista
- puntero a la primera celda

La clase Lista

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

La clase Lista I

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;
}
Celda(double valor){
   this->valor=valor;
   siguiente=0;
}

inline double obtenerValor() const{
   return valor;
}
```

La clase Lista II

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

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

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

La clase Lista I

Para la clase Lista:

```
#ifndef LISTA_H
#define LISTA_H

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

#endif /* LISTA_H */
```

La clase Lista

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 constructures y destructores

Los constructores de la clase Lista I

Método que se ejecuta cuando se crea un objeto de una clase.

No se hace una llamada explícita, se ejecuta automáticamente cuando se crea el objeto.

```
//Manera mas simple
SecuenciaEnteros miSecuencia;

//Con memoria dinamica
SecuenciaEnteros *p = new SecuenciaEnteros;
```

Los constructores de la clase Lista II

Los constructores se encargan de inicializar de forma conveniente los datos miembro, para asegurar un estado válido.

Si el objeto requiere memoria dinámica, deben además reservar la memoria necesaria.

Pueden existir varios constructores para una clase, **sobrecarga**.

Normalmente son métodos **public**.

Tiene el mismo nombre de la clase y no devuelve nada.

Constructores: constructor por defecto I

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

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

Constructores: constructor auxiliar

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;
    }
}
```

Constructores: constructor auxiliar I

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 recien creada
        pCeldaAnterior=pCelda;
    }

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

Constructores: constructor auxiliar I

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);
    }
}
```