Universidad de Murcia

Material didáctico para la asignatura Sistemas Inteligentes de 3º de Grado en Informática

AUTORES:

- José Manuel Cadenas Figueredo
- María del Carmen Garrido Carrera
- Raquel Martínez España
- Santiago Paredes Moreno

Capítulo 2

Definición de clases

2.1 Introducción

El lenguaje C++ tiene la capacidad de permitir realizar una programación estructurada y también nos proporciona herramientas para realizar una programación orientada a objetos. Una de las herramientas clave que nos proporciona C++ orientado a objetos son las clases. Una clase no es más que una especificación a seguir para construir objetos. Un objeto lo podemos definir como una entidad autónoma que tiene una funcionalidad bien definida. Haciendo una analogía con el lenguaje de programación C, podemos ver una clase como una estructura en C, teniendo en cuenta que las clases de C++ nos proporcionan ciertas ventajas, sencillas pero bastante potentes.

2.2 Definición de clases

Continuando con la analogía del lenguaje C, para definir una clase lo que debemos de hacer es cambiar la palabra reservada `struct` por `class`. Un ejemplo sencillo para empezar a ver la definición de las clases, sería el siguiente:

```
1 class Editorial{
2     char * nombre;
3     char * direccion;
4     int identificador;
5     int nPublicaciones;
6     void informacion();
7     void addPublicacion(char * titulo);
8 };
```

Este código define la clase `Editorial` que está compuesta de los atributos `nombre`, `direccion`, `identificador` y número de publicaciones (`nPublicaciones`). Además, en la definición de la clase se indican las cabeceras de los métodos de la clase.

Una de las características de la definición de clases es que se puede especificar dentro de una clase diferentes especificaciones de acceso. Una especificación de acceso indica cuál es el nivel de visibilidad de los atributos o métodos que hay definidos. Las especificaciones de acceso que se pueden indicar son `private`, `public` y `protected`.

La especificación de acceso `private` indica que los atributos o funciones definidos en tal especificación sólo son accesibles desde dentro de la clase, es decir, que ninguna función externa a la clase o función de clase derivada tendrá acceso. La especificación `public` indica que los atributos y funciones definidos en esta especificación serán accesibles desde cualquier parte donde el objeto sea accesible. Por último la especificación de acceso `protected`, se refiere a que los atributos y funciones de la clase se comportarán como de acceso `public` para las clases derivadas y se comportarán como `private` para las funciones externas. Para definir la especificación de acceso simplemente hay que indicar la palabra reservada seguida de "`:`". Cuando no se especifica nada en una clase, por defecto se considera que la especificación de acceso es `private`.

Un ejemplo de cómo definir las especificaciones de acceso es el siguiente:

```
1  class Editorial{
2
3  private:
4      char * nombre;
5      char * direccion;
6      int identificador;
7      int nPublicaciones;
8
9  public:
10     void informacion();
11     void addPublicacion(char * titulo);
12
13 };
```

Un aspecto a comentar es que las funciones las podemos implementar dentro o fuera de la clase. En el código mostrado arriba solamente se están declarando las funciones y atributos, por lo que las funciones van a ser implementadas fuera de la clase. Para

identificar que se están implementando las funciones de una clase, debemos de utilizar el operador de ámbito `::` indicando antes de tal operador a qué clase pertenecen. Así por ejemplo para implementar la función `informacion` fuera de la clase deberíamos de implementarla de la siguiente manera:

```
1 void Editorial::informacion(){
2   cout << "El nombre de la editorial es " << nombre << endl;
3 }
```

Un aspecto a tener en cuenta es que no se reserva memoria para un objeto de una clase hasta que este objeto es creado. Es el constructor de la clase el encargado de crear el objeto y de reservar la memoria necesaria para inicializar aquellos campos que sean requeridos. De la misma forma, cuando un objeto ya no es necesario utilizarlo hay que llamar al destructor para liberar aquella memoria que se reservó en el constructor de tal objeto. Vamos a ver de forma más detallada algunos aspectos acerca de los constructores y destructores de una clase.

2.3 Constructores

Los constructores, como hemos comentado, son funciones que nos sirven para crear el objeto e inicializar los valores pertinentes reservando memoria en caso de que sea necesario, es decir, mediante la llamada a un constructor inicializamos los valores de un objeto al mismo tiempo que creamos el objeto. Los constructores son funciones especiales debido a las siguientes razones:

- No tienen tipo de retorno, ni retornan ningún valor.
- No se heredan.
- El nombre de la función debe de ser exactamente el mismo que el de la clase.
- Un constructor siempre debe de ser una función pública, ya que es la que crea el objeto. No tiene sentido un constructor `private` o `protected`, ya que un constructor, la mayoría de las veces, será llamado desde funciones exteriores de la clase y al no poder heredarse, no tiene sentido que sea público para las clases derivadas y no para el resto de funciones.

Si en una clase no se define un constructor, el compilador crea uno por defecto sin parámetros para crear el objeto, pero este constructor no inicializa absolutamente nada, por lo que los atributos que deban de ser inicializados, contendrán una información incorrecta. Un aspecto importante es que si se ha definido un constructor con argumentos, cuando se llame a ese constructor habrá que pasarle los argumentos pertinentes de forma obligatoria. Siguiendo con el ejemplo que teníamos anteriormente de la editorial, veamos cómo sería el constructor de tal clase:

```
1  class Editorial{
2
3  private:
4      char * nombre;
5      char * direccion;
6      int identificador;
7      int nPublicaciones;
8
9  public:
10     Editorial(char * name, char * address, int id, int npubli);
11     void informacion();
12     void addPublicacion(char * titulo);
13 };
14
15 Editorial::Editorial (char * name, char * address, int id, int npubli){
16     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
17     strcpy(nombre,name);
18     direccion = new char[strlen(address)+1];
19     strcpy(direccion,address);
20     identificador = id;
21     nPublicaciones = npubli;
22 }
```

Como podemos observar dentro de la clase hemos declarado el constructor y fuera de la clase lo hemos implementado indicando mediante el operador de ámbito `::` que pertenece a la clase `Editorial`. Dentro de la implementación del constructor hemos reservado la memoria correspondiente para los atributos `nombre`, `direccion` y hemos inicializado los valores.

Se pueden definir tantos constructores como sea necesario, es decir, el lenguaje C++ permite la sobrecarga de constructores, así si hay circunstancias donde no es necesario inicializar atributos cuando se crea el objeto, pero hay otras situaciones donde sí,

definimos los constructores modificando el número de parámetros que reciben. Continuando con el ejemplo de la editorial, vamos a sobrecargar el constructor creando otro constructor más con diferentes parámetros:

```
1  class Editorial{
2
3  private:
4      char * nombre;
5      char * direccion;
6      int identificador;
7      int nPublicaciones;
8
9  public:
10     Editorial(char * name, char * address, int id, int npubli);
11     Editorial(char * name, char * address, int id);
12     void informacion();
13     void addPublicacion(char * titulo);
14 };
15
16 Editorial::Editorial (char * name, char * address, int id, int npubli){
17     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
18     strcpy(nombre,name);
19     direccion = new char[strlen(address)+1];
20     strcpy(direccion,address);
21     identificador = id;
22     nPublicaciones = npubli;
23 }
24
25 Editorial::Editorial(char * name, char * address, int id){
26     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
27     strcpy(nombre,name);
28     direccion = new char[strlen(address)+1];
29     strcpy(direccion,address);
30     identificador = id;
31     nPublicaciones = 0;
32 }
```

En el ejemplo vemos cómo hemos definido dentro de la clase dos constructores, con diferentes parámetros y después fuera de la clase los hemos implementado al igual que hemos realizado anteriormente utilizando el operador de ámbito `::`.

Un constructor importante es el llamado constructor de copia. Este constructor es utilizado para, como su nombre indica, crear una copia del objeto que recibe como parámetro. Este tipo de constructor sólo recibe un parámetro que es una referencia al objeto de la misma clase. Siguiendo con el ejemplo de la clase editorial, veamos cómo funciona el constructor de copia.

```
1  class Editorial{
2
3  private:
4      char * nombre;
5      char * direccion;
6      int identificador;
7      int nPublicaciones;
8
9  public:
10     Editorial(char * name, char * address, int id, int npubli);
11     Editorial(char * name, char * address, int id);
12     Editorial(const Editorial &e);
13     void informacion();
14     void addPublicacion(char * titulo);
15 };
16
17 Editorial::Editorial (char * name, char * address, int id, int npubli){
18     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
19     strcpy(nombre,name);
20     direccion = new char[strlen(address)+1];
21     strcpy(direccion,address);
22     identificador = id;
23     nPublicaciones = npubli;
24 }
25
26 Editorial::Editorial(char * name, char * address, int id){
27     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
28     strcpy(nombre,name);
29     direccion = new char[strlen(address)+1];
30     strcpy(direccion,address);
31     identificador = id;
32     nPublicaciones = 0;
33 }
34
35 Editorial::Editorial(const Editorial &e){
36     nombre = (char *) malloc((strlen(e.nombre)+1) * sizeof(char));
```

```
37      direccion = new char[strlen(e.direccion)+1];
38      strcpy(nombre,e.nombre);
39      strcpy(direccion,e.direccion);
40      identificador = e.identificador;
41      nPublicaciones = e.nPublicaciones;
42  }
```

Como se puede observar el constructor de copia lo que hace es hacer una copia en profundidad del objeto que se recibe como parámetro. Como se aprecia no se hace una asignación de punteros sino que se copia el contenido de la memoria explícitamente con `strcpy`. Esto es importante porque si se hace una copia de puntero como tal, al modificar el nuevo objeto estaríamos modificando también el objeto que se ha pasado para copiar. Es importante destacar que durante el ejemplo que hemos detallado se han especificado dos formas de reservar memoria en C++, utilizando la función `malloc`, heredado del lenguaje C y creando un objeto tipo `char *` con el operador `new` e indicándole la cantidad de espacio en memoria a reservar.

El constructor de copia es llamado no sólo cuando se quiere crear una copia de un objeto sino que el compilador en ciertas situaciones también lo utiliza. Una de esas situaciones es cuando se iguala un objeto a otro a la misma vez que se inicializa. Por ejemplo:

```
1  Editorial e1("Thomson","Avenida de Madrid",3,25634);
2  Editorial e2 = e1;
```

En el ejemplo hemos declarado el objeto `e1` de la clase `Editorial` y hemos llamado al constructor con sus parámetros y luego hemos declarado otro objeto `e2` de tipo `Editorial` y lo hemos inicializado igualándolo al objeto `e1`, por lo que internamente el compilador al analizar esta sentencia lo que hace es llamar al constructor de copia.

2.4 Destructores

Al igual que los constructores, los destructores son funciones especiales en el lenguaje C++. El destructor es una función que se llama bien directamente o bien de forma automática al querer eliminar un objeto, es decir, desde una función se puede destruir un objeto que ya no sea necesario, por ejemplo, un objeto utilizado como auxiliar. Pero

además cuando se sale de una función donde se ha creado un objeto, al abandonar el ámbito, el objeto local se destruye.

Si no se declara un destructor en una clase, el compilador crea uno por defecto pero este destructor, al igual que ocurre con el constructor por defecto, no tendrá sentido si disponemos de atributos de tipo puntero declarados dentro de la clase, puesto que la memoria reservada para estos punteros no será liberada por el destructor por defecto. Así que es recomendable declarar e implementar un destructor en toda clase que tenga algún atributo que utilice la memoria de forma dinámica, por ejemplo, un puntero, un vector, etc. Las características de un destructor son bastante similares a las de los constructores. Las más destacables son:

- No tienen ningún tipo de retorno, la función no devuelve nada.
- No tiene parámetros ni pueden ser heredados.
- Deben de ser funciones públicas para que puedan ser llamadas desde cualquier parte donde se tenga acceso al objeto.
- Se definen con el mismo nombre de la clase, pero delante del nombre se debe de poner el símbolo ~.
- Al contrario que los constructores, no pueden ser funciones sobrecargadas, ya que al no tener parámetros ni retornos es lógico que no se puedan sobrecargar.

Siguiendo con el ejemplo de la editorial, vamos a ver cómo definimos el destructor y cómo eliminamos los atributos tipo puntero que se declaran en la clase.

```
1  class Editorial{
2
3  private:
4      char * nombre;
5      char * direccion;
6      int identificador;
7      int nPublicaciones;
8
9  public:
10     Editorial(char * name, char * address, int id, int npubli);
11     Editorial(char * name, char * address, int id);
12     Editorial(const Editorial &e);
```

```
13      ~Editorial();
14      void informacion();
15      void addPublicacion(char * titulo);
16  };
17
18  Editorial::Editorial (char * name, char * address, int id, int npubli){
19      nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
20      strcpy(nombre,name);
21      direccion = new char[strlen(address)+1];
22      strcpy(direccion,address);
23      identificador = id;
24      nPublicaciones = npubli;
25  }
26
27  Editorial::Editorial(char * name, char * address,int id){
28      nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
29      strcpy(nombre,name);
30      direccion = new char[strlen(address)+1];
31      strcpy(direccion,address);
32      identificador = id;
33      nPublicaciones = 0;
34  }
35
36  Editorial::Editorial(const Editorial &e){
37      nombre = (char *) malloc((strlen(e.nombre)+1) * sizeof(char));
38      direccion = new char[strlen(e.direccion)+1];
39      strcpy(nombre,e.nombre);
40      strcpy(direccion,e.direccion);
41      identificador = e.identificador;
42      nPublicaciones = e.nPublicaciones;
43  }
44
45  Editorial::~Editorial(){
46      delete[] direccion;
47      free(nombre);
48  }
```

Como podemos apreciar al implementar el destructor se ha liberado la memoria de dos formas distintas, una de ellas utilizando la función `free` y la otra utilizando el operador `delete`. Esto es debido a que para el atributo `direccion` hemos reservado memoria mediante el operador `new` y para el atributo `nombre` hemos utilizado la función `malloc`.

Tras ver los elementos principales de las clases en el lenguaje C++, vamos a ver un pequeño ejemplo concreto de una clase y de cómo se puede modular por ficheros la definición de la clase y de sus métodos.

2.5 Ejemplo

En este apartado vamos a ver un ejemplo completo de la definición de una clase modulada en ficheros.

Fichero `editorial.h`: En este fichero hemos definido la clase y declaramos sus funciones y atributos.

```
1  class Editorial{
2
3  private:
4      char * nombre;
5      char * direccion;
6      int identificador;
7      int nPublicaciones;
8
9  public:
10     Editorial(char * name, char * address, int id, int npubli);
11     Editorial(char * name, char * address, int id);
12     Editorial(const Editorial &e);
13     ~Editorial();
14     void informacion();
15     void addPublicacion(char * titulo);
16 };
```

Fichero `editorial.cpp`: En este fichero se implementan las funciones de la clase. Para ello hacemos uso del operador de ámbito `::`.

```
1 #include<iostream>
2 #include "editorial.h"
3
4 using namespace std;
5
6 Editorial::Editorial (char * name, char * address, int id, int npubli){
7     nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
8     strcpy(nombre,name);
9     direccion = new char[strlen(address)+1];
```

```
10      strcpy(direccion,address);
11      identificador = id;
12      nPublicaciones = npubli;
13  }
14
15  Editorial::Editorial(char * name, char * address,int id){
16      nombre = (char *) malloc((strlen(name)+1) * sizeof(char));
17      strcpy(nombre,name);
18      direccion = new char[strlen(address)+1];
19      strcpy(direccion,address);
20      identificador = id;
21      nPublicaciones = 0;
22  }
23
24  Editorial::Editorial(const Editorial &e){
25      nombre = (char *) malloc((strlen(e.nombre)+1) * sizeof(char));
26      direccion = new char[strlen(e.direccion)+1];
27      strcpy(nombre,e.nombre);
28      strcpy(direccion,e.direccion);
29      identificador = e.identificador;
30      nPublicaciones = e.nPublicaciones;
31  }
32
33  Editorial::~Editorial(){
34      delete[] direccion;
35      free(nombre);
36  }
37
38  void Editorial::informacion(){
39      cout << "El nombre de la editorial es " << nombre << endl;
40  }
41
42  void Editorial::addPublicacion(char * titulo){
43      cout << "Se añade la publicacion con titulo " << titulo << endl;
44      nPublicaciones++;
45  }
```

Fichero main.cpp: En este fichero declaramos la función `main` donde creamos objetos de la clase `Editorial` y llamamos a las funciones que hemos definido.

```
1 #include<iostream>
2 #include "editorial.h"
3
```

```
4  int main(){
5
6      Editorial miEditorial("Thomson","Calle Mayor",1,25);
7      Editorial miEditorialaux("Norman","Calle Australia",2);
8      Editorial miEditorialcopia(miEditorial);
9
10     miEditorialcopia.addPublicacion("Programacion en C++");
11     miEditorialcopia.informacion();
12     miEditorialaux.informacion();
13
14     system("pause");
15     return 0;
16 }
```