uc3m | Universidad Carlos III de Madrid

Informática Industrial I

Grado en Ingeniería en Electrónica Industrial y Automática

Álvaro Castro González

Nicolas Burus

Mohamed Abderrahim

José Carlos Castillo Montoya

Práctica 2

Clases I.
Definición, Constructores, Destructores, …

A lo largo de los últimos años, han colaborado en impartir las prácticas presenciales, y por tanto proporcionar comentarios para mejorar algunos aspectos de este documento los profesores siguientes:

Ioannis Douratsos
Fares Abu-Dakka
Juan Carlos González Victores
Raúl Perula
Avinash Ranganath

Práctica 2 - Clases

1. Clases

1.1. Definición

Lo primero que se va a ver es como se estructura un fichero `main.cpp` en C++.

```
////////////////////////////////////////////////////
// main.cpp
////////////////////////////////////////////////////

#include <iostream>
#include “point.h”

using namespace std;
using namespace space;

int main(int argc, char **argv) {
      Point P;
      cout << P.getX() << endl;
      cout << P.getY() << endl;

      P.set(10, 10);
      cout << P.getX() << endl;
      cout << P.getY() << endl;

      P.display();
}
```

En este programa, lo primero que se hace es incluir la biblioteca estándar de C++ y el fichero de cabecera `point.h`, que contendrá la declaración de la clase que se va a crear posteriormente. Nótese que la biblioteca estándar no se incluye de igual manera que en C, por ejemplo, en el que sería `iostream.h`. En C++, la mayoría de los ficheros de inclusión estándar, acaban sin `.h` y a veces comienzan con la letra “`c`”.

Algunos ejemplos de ficheros de C que pasan a C++ son:

```
#include <stdlib.h>     →    #include <cstdlib>
#include <assert.h>     →    #include <cassert>
```

Después, se pueden encontrar las directivas:

```
using namespace std;
using namespace space;
```

Estas indican al compilador que se van a usar los espacios de nombres `std` (estándar) y `space` (que se crea por el usuario más adelante). En C++, las clases se pueden encapsular en **espacios de nombres**. De esta forma se organizan mejor para proyectos grandes. El espacio de nombres `std` define el espacio de nombres estándar de C++. Si no se indica explícitamente que se va a usar, el compilador dará un error al tratar de usar por ejemplo `cout`, ya que esta es una clase de este espacio de nombres. En caso de no especificar el espacio de nombre y para que no de error al utilizar alguna de estas funciones, se puede llamarlo de forma explícita utilizando `std::cout`.

En la función `main`, observar la siguiente declaración:

```
Point P;
```

Esto está indicando al compilador que debe crear una variable de tipo `Point`. Es decir, que se va a crear una variable de la clase `Point`, o también se dice que se ha creado un objeto de la clase `Point`.

Como ya se ha visto en teoría, se llama a los métodos de una clase usando el operador "`.`" (punto). Por eso, la expresión `P.getX()` es una llamada a la función miembro `getX()` del objeto `P`.

Si se considera el código siguiente:

```
Point P, P2;
cout << P.getX() << endl;
cout << P2.getX() << endl;
```

Se estarían creando dos objetos distintos, ambos de la clase `Point`. Y `P.getX()` sería una llamada a la función miembro del objeto, que sería distinto a llamar a `p2.getX()`. La diferencia, como se verá más adelante, es que `P.getX()` usará los valores del objeto `P`, y `P2.getX()` usara los valores de `P2`.

En este momento ya se puede ver cómo se va a definir una clase. Una clase se define en dos ficheros, un fichero `.h` y un fichero `.cpp`. El fichero `.h` contiene la *definición o declaración* de la clase, mientras que el `.cpp` contiene la *implementación* de la clase.

Las clases se pueden definir dentro de un espacio de nombres "`namespace`". Los espacios de nombres comienzan por *letra minúscula* (**por convenio**). El nombre de las clases comienza con *mayúscula* (**por convenio**). La clase `Point` representará un punto en el espacio. Esta clase permitirá manejar un punto mediante las funciones miembro o métodos de la clase.

Las clases se definen usando la palabra clave `class`. Seguidamente se define la *parte pública*, es

decir, aquella que podrá ser usada por otras clases o funciones. Más adelante se define la *privada*, donde se colocan las variables o funciones que solo podrán accederse desde funciones miembro de la misma clase.

A continuación se muestra el fichero `point.h`.

```
////////////////////////////////////////////////
// point.h
////////////////////////////////////////////////

#ifndef _POINT_H_
#define _POINT_H_

#include <iostream>

using namespace std;

namespace space {
      /**\brief This class represents a point
      */
      class Point {
            public:
                  int getX();
                  int getY();
                  void set(int x, int y);
                  void display();
            private:
                  int _x, _y;
      };
}

#endif
```

Dentro de la clase se han definido dos variables privadas y cuatro métodos públicos. El primer y segundo método sirven para obtener los valores de las coordenadas del punto, el segundo método sirve para especificar los valores del punto, y el tercer método sirve para ver los valores de las coordenadas del punto. Finalmente, en la parte privada, se han declarado dos enteros que representan los valores del punto de la clase. Las variables privadas se declaran comenzando por *subrayado* "**_**" (**por convenio**).

Para ocultar los detalles de implementación de una clase y así permitir cambiar su implementación sin tener que cambiar el resto del código que la utiliza, C++ introduce los **especificadores de acceso**. Estos permiten definir qué miembros de una clase son accesibles desde fuera de ésta y cuáles no. Hay tres especificadores de acceso: `public`, `private` y `protected`. El primero, `public`, significa que el miembro es accesible desde fuera de la definición de la clase, mientras que los miembros `private` y `protected` sólo se pueden acceder desde dentro de las funciones de la clase o las clases habilitadas para ello. Por defecto, todos los miembros de una clase son **privados**.

Si se intenta acceder a un atributo o método privado, el compilador dará un error y no se podrá crear el ejecutable. En el siguiente código se puede probar como fallaría la compilación del `main.cpp`.

```
int main() {
	Point P;
	P._x = 10; // imposible! ERROR!
	P.display();
}
```

Otro aspecto muy importante de la programación es la documentación. Se deben documentar todos los ficheros fuente que se realicen. Los comentarios tendrán formato **Doxygen** como el que se muestra a continuación:

```
/**
*/
```

Para este formato de documentación existen herramientas que nos crearán una documentación HTML del código y que se podrá consultar en cualquier navegador.

Ahora, se va a crear el `.cpp`.

```
////////////////////////////////////////////////
// point.cpp
////////////////////////////////////////////////

#include "point.h"

namespace space {
	////////////////////////////////////
	//
	////////////////////////////////////
	int Point::getX() {
		return _x;
	}

	////////////////////////////////////
	//
	////////////////////////////////////
	int Point::getY() {
		return _y;
	}

	////////////////////////////////////
	//
	////////////////////////////////////
	void Point::set(int x, int y) {
		_x = x;
		_y = y;
	}

	////////////////////////////////////
	//
```

```
      ////////////////////////////////////
      void Point::display() {
            cout << _x << “, “ << _y << endl;
      }
}
```

Finalmente, compilar haciendo uso de QtCreator y ejecutar el programa.

1.2. Constructores y destructores

1.2.1. Constructores

Probablemente, ya se habrán dado cuenta que cuando se instancia un objeto de una clase, los valores de los atributos de la clase tienen valor inicial “basura”. Sin embargo, es posible alterar este comportamiento de la clase añadiendo constructores. El constructor es una función que se invoca al crear un objeto de la clase, pero para ello ha de estar definida.

Constructor vacío

Crear primero el constructor vacio. Añadir el siguiente código a la definición de la clase justo debajo de la palabra clave “`public`”.

```
/** Empty constructor
*/
Point();
```

Esto es un constructor, que es una función miembro que no devuelve nada y que se llama **igual que la clase**. Fíjese muy bien que no solo no devuelve nada, sino que tan siquiera dice **`void`**. Al ser una función miembro puede tener parámetros, pero en este caso se encuentra vacío de parámetros. Los constructores siempre son las primeras funciones de la clase y deben ser **públicos**. Ahora ver la implementación en el `.cpp`.

```
////////////////////////
//
////////////////////////
Point::Point() {
      _x = 0;
      _y = 0;
}
```

Lo que se hace es inicializar las variables de la clase a valor 0, por ejemplo, para que no sea *“basura”* al crearse el objeto. Finalmente, volver a compilar y ejecutar. Como se observa, la salida ahora es 0.

Sobrecarga de constructores

En C++ es posible crear **varias** funciones miembro con el **mismo nombre** aunque con diferentes argumentos, a esto se le llama **sobrecarga**. Será el compilador el que se encargue de decidir a qué

función se debe llamar dependiendo de los argumentos que se usen.

Se va a probar la sobrecarga añadiendo dos constructores típicos que son: el **constructor parametrizado** y el **constructor de copia**. El primero servirá para asignar un valor a las variables en la creación de la clase y el segundo servirá para crear un objeto que sea copia de otro. Añadir el siguiente código al `.h`.

```
/** Parametrized constructor
*/
Point(int x, int y);

/** Copy constructor
*/
Point(const Point & P);
```

Y ahora habrá que completar el `.cpp`.

```
//////////////////////////
//
//////////////////////////
Point::Point(int x, int y) {
	_x = x;
	_y = y;
}

//////////////////////////
//
//////////////////////////
Point::Point(const Point & P) {
	_x = P._x;
	_y = P._y;
}
```

Obsérvese que el primer constructor recibe como argumento valores enteros y los asigna a las variables. El segundo requiere un poco mas de explicación. Se denomina constructor de copia y lo que hace es que hace que el objeto creado sea una copia del que se pasa. El argumento (único) es una ***referencia a un objeto de la misma clase***.

```
const Point & P
```

La palabra clave `const` sirve para indicar que el elemento pasado no debe ser modificado en el interior de la función. Después, se observa el símbolo `&`. En C hay dos formas básicas de pasar argumentos, **por valor** y **por referencia**. En el primer caso, se pasaba un elemento y dentro de la función se hacia una copia con la que se trabajaba. Por referencia (puntero), se pasaba la dirección de memoria y se podía modificar el valor directamente.

Cuando se quiere pasar como parámetro a una clase en C++, se tiene la misma idea, paso por valor

o por referencia. Sin embargo, el paso por valor **no** es recomendable cuando se pasa a una clase. Esto es porque las clases pueden contener una gran cantidad de variables y el realizar una copia podría llegar a ser muy costoso e ineficiente. Por tanto, es preferible hacer el paso de un objeto siempre **por referencia**. Para ello, en C++ se define el operador `&`. Cuando se pasa `&`, se está indicando que se puede modificar el valor de la variable dentro de la función. Las referencias se verán con detalle en la siguiente práctica. Ahora se puede ver un ejemplo simple.

```
main() {
	int v = 10;
	modify(v);
	cout << v << endl;
}

void modify(int & v) {
	v = 0;
}
```

En el caso del constructor de copia, se está pasando por referencia el objeto (para evitar una copia de todos sus datos), pero a la vez se le está diciendo que se haga constante. Es decir, que no sea posible modificar sus valores en el interior. Por tanto, el siguiente código sería **incorrecto**:

```
//////////////////////////
//
//////////////////////////
Point::Point(const Point & P) {
	P._x = 100;
	P._y = 101;
}
```

Ya que se está modificando las variables internas del objeto pasado P, que se supone que es constante dentro de la función.

1.2.2. Destructores

El destructor es la función a la que se llama cuando se debe destruir el objeto. Es decir, la función a la que se llama cuando ya no se va a usar el objeto nunca más. Si se define en el `.h`.

```
/** Destructor
*/
~Point();
```

Ahora se tiene algo nuevo `~Point()`. Esto será el destructor de la clase. El objeto tendrá que liberar la memoria en caso de que se hubiese creado. Lo bueno de los destructores es que es una función a la que llama el compilador **automáticamente** cuando detecta que el objeto no se va a usar más. No se tiene que llamar al destructor, el compilador lo hace por el usuario. En el caso del ejemplo de la clase `Point`, el destructor no hará nada de momento, pero cuando se vea la reserva de

memoria *dinámica* en la siguiente práctica se verá que será muy útil.

El código que iría en el .cpp sería:

```
//////////////////////////
//
//////////////////////////
Point::~Point() {
}
```

1.2.3. Operadores

Otra de las ventajas de las clases es que se pueden definir operadores sobre las clases (=, +, -, *, /). Se va a definir el operador de asignación. Añadir en la parte publica de la clase `Point` la siguiente definición:

```
/** Assign operator
*/
Point & operator=(const Point & P);
```

Y en el `.cpp`:

```
//////////////////////////
//
//////////////////////////
Point & Point::operator=(const Point & P) {
      _x = P._x;
      _y = P._y;

      return *this;
}
```

Los operadores son funciones miembro que permiten definir operaciones habituales. Todos los operadores devuelven una copia del objeto actual con el ***objetivo de poder realizar un encadenamiento.***

El valor de retorno es `Point &`, es decir, una referencia a un objeto de la clase. Si se observa la implementación, una vez se realiza la copia, se retorna `*this`. La palabra reservada `this`, es puntero al **objeto actual**. El puntero `this`, usado dentro de una función de la clase `Point`, es un puntero al objeto mismo, por tanto, en este caso es un puntero de tipo `Point *`. Si `this` es un `Point *,` entonces `*this` es el propio objeto, es decir `Point`. Entonces, lo que se está devolviendo es el propio objeto. Ver un ejemplo:

```
int main(int argc, char **argv) {
      Point P(10, 10); // parametrized constructor
      Point P2;
      cout << P2.getX() << endl;
      cout << P2.getY() << endl;
      P2 = P;
      P2.display();
}
```

En este ejemplo, `P` es asignado a `P2`. Por tanto, `P2` será una copia de `P`. Se puede ver ahora entonces:

```
int main(int argc, char **argv) {
      Point P(10);
      Point P2, P3;
      P2.display();
      P3 = P2 = P;
      P3.display();
}
```

En este ejemplo, `P` es asignado a `P2`, que a su vez es asignado a `P3`. Al escribir `P3 = P2 = P`, el compilador primero realizará `P2 = P`. El resultado de esta operación es `P2`, que será asignado a `P3`. Es decir, el compilador descompondrá las operaciones en:

```
P2 = P;
P3 = P2;
```

Esto es posible gracias a que el resultado de la asignación es un `Point &`.

Ejercicios

- Implementar las clases necesarias para realizar una base de datos de personas. Para ello, se debe definir:
    - Una clase `Person` en un fichero `Person.h`, y la implementación de sus funciones miembro en un fichero `Person.cpp`.
        - Deberá contener dos atributos privados, `_name,` de tipo `string` y `_age` de tipo entero.
        - Deberá contener las siguiente funciones miembro públicas:
            - Constructor vacío, parametrizado y de copia. Realizarán las inicializaciones pertinentes.
            - Destructor de la clase.
            - Métodos `getName` y `setName`: obtener y poner el nombre de la persona.
            - Métodos `getAge` y `setAge`: obtener y poner la edad de la persona.
            - Operador de asignación.
- Realizar un fichero `main.cpp` en el que la función `main` deberá realizar lo siguiente:
    - Instanciar un objeto de tipo `Person` mediante su constructor parametrizado y que se muestren por pantalla los datos.
    - Instanciar otro objeto de tipo `Person` mediante su constructor de copia, teniendo como parámetro el objeto anterior y que se muestren los datos de ambos objetos por pantalla.
    - Instanciar un último objeto de tipo `Person` mediante su constructor vacio y llamar a sus funciones `setAge` y `setName`. Mostrar los datos por pantalla.
    - Instanciar dos objetos más de tipo `Person` y hacer que valgan lo mismo que el objeto anterior utilizando el operador de asignación.
    - Por último, mostrar todos los nombres de las personas que contengan una ‘s’. Mostrar todos los nombres de las personas que tengan una edad superior a 18 años.

Notas

1. Al menos debe haber un resultado.
2. Se espera que los alumnos completen el trabajo por su cuenta si no logran terminarlo durante el horario de práctica presencial.