Universidad Carlos III de Madrid

Práctica de Informática Industrial I

Grado en Ingeniería Electrónica Industrial y Automática

2016-2017

Práctica 3

Clases II. Métodos, Reserva dinámica y Referencias

Profesores:

Mohamed Abderrahim
Álvaro Castro González
Ioannis Douratsos
José Carlos Castillo
Lia Garcia
Juan Carlos González Victores

Práctica 3 – Clases II. Métodos, Reserva dinámica y Referencias.

1. Métodos

1.1.1. Métodos constantes

En C++, se puede usar el modificador `const` para definir una **función constante**. Observar en el siguiente ejemplo la función `getVal()`.

```
class MyC {
      public:
            MyC() {_val = 10;}
            ~MyC() {}
            void setVal(int v) {_val = v;}
            int getVal() const {return _val;}
            int getValNoConst() {return _val;}
      private:
            int _val;
};
```

Al declarar una función como constante, le estamos indicando al compilador que la función no modificará en su interior el contenido del objeto. Es decir, es una función que accede a los valores, pero solo para ver su valor, posiblemente calcular algo usándolos y a lo mejor retornar algo. Pero **no modificar** el estado del objeto, es decir, ninguna de sus variables internas.

¿Para qué sirve esto? Observar del siguiente ejemplo.

```
class MyCC {
      public:
            MyCC() {}
            ~MyCC() {}
            void readFromMyC(const MyC & M) {_val = M.getVal();}
      private:
            float _val;
};
```

La clase `MyCC`, en la función miembro `readFromMyC`, se hace que `_val` valga lo mismo que vale `getVal()`. Darse cuenta que en la declaración se pone `const MyC & M`, es decir, que `M` es una variable constante. Lo que quiere decir que no se debe poder modificar `M` dentro de la función.

Al ser `M` de tipo constante, el compilador comprobará que no se hacen llamadas que descarten ese cualificador (el cualificador constante). Es decir, solo permitirá llamadas a funciones que sean de tipo constante. Por tanto, si la variable `getVal()` de `MyC` no se hubiese definido como constante, no se hubiese podido usar en `readFromMyC`. O dicho de otro modo, sólo se podrá llamar a funciones constantes de objetos que se comportan como constantes. Con lo que el siguiente código daría error de compilación:

```
class MyCC {
	public:
		MyCC() {}
		~MyCC() {}
		void readFromMyC(const MyC & M) {_val = M.getValNoConst();}
	private:
		float _val;
};
```

Pero esto tiene otra limitación. Se ha comentado que al definir una función como constante, no se puede modificar el valor del objeto. Por tanto, dentro de una función constante, sólo se podrán hacer llamadas a funciones constantes.

Por ejemplo, el siguiente código daría error de compilación porque `foo()` se declara como una función constante pero dentro se hace una llamada a una función **no** constante. Aunque en realidad la función `getValueNoConst()` no modifica el objeto `_myc`, el compilador no lo sabe a priori. El compilador "se cura en salud" y dice, "como `getValueNoConst()` no es contante, podría ocurrir que se modificase el objeto. Así que no lo voy a permitir".

```
class MyCCC {
	public:
		MyCCC() {}
		~MyCCC() {}
		int foo() const {return  _myc.getValNoConst()*10;}
	private:
		MyC _myc;
};
```

¿Cuál sería la solución al problema? Pues llamar a una función constante. El siguiente código si sería correcto:

```
class MyCCC {
	public:
		MyCCC() {}
		~MyCCC() {}
		int foo() const {return  _myc.getVal()*10;}
	private:
		MyC _myc;
};
```

2. Memoria dinámica y Referencias

2.1. Reserva y liberación de memoria

Para la reserva de memoria dinámica se usa la directiva `new`. Y para la liberación de la misma se utiliza `delete`. Incluir lo siguiente en el `.h` de la clase Point como variable pública:

```
int size;
int *v;
```

Ahora, añadir al constructor vacío lo siguiente:

```
size = 2;
v = new int[size];
for (int i = 0; i < size; i++) {
      v[i] = 0;
}
```

Como se puede observar, se crea un vector de tipo entero de 10 elementos. Y en el constructor vacío de la clase se reserva memoria y se inicializan sus valores.

La sintaxis general para la reserva de memoria en C++ es:

```
tipo_dato *var = new tipo_dato[numero_elementos];
```

Lo bueno de `new` es que el calcula el tamaño de la memoria por el usuario. Probar el siguiente código de ejemplo:

```
#include <iostream>

using namespace std;

int main() {
      int *v = new int[10]; // vector entero de 10 elementos
      v[0] = v[1] = 1;
      cout << v[0] << ", " << v[1] << endl;
      delete[] v;
}
```

Además, este mismo ejemplo se podría aplicar dentro de una clase. Si se vuelve a tomar la clase `Point` y se añade una variable pública `int *v`; en el constructor de la clase se podrá inicializar reservando memoria para dicho vector.

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

class Point {
      public:
            int *v;
            int size;
}

//////////////////////////
.cpp
//////////////////////////

Point::Point() {
    size = 10;
    v = new int[size];
    for (int i = 0; i < size; i++) {
       v[i] = 0;
    }
}
```

Creación dinámica de objetos

Además de poderse reservar memoria para los tipos básicos, también se puede reservar memoria dinámica para los objetos. En este caso sería de la siguiente forma:

```
main() {
      Point *P = new Point;
}
```

Con esto, se está creando de forma dinámica un objeto de tipo `Point`. El operador `new` se encargará de llamar al constructor, en este caso al constructor vacío. Aunque también se podría haber escrito lo siguiente para llamar a cualquier otro constructor, en este caso al parametrizado.

```
main() {
      Point *P1 = new Point(10, 10);
      P1->display();
}
```

Ahora bien, cuando se crea un objeto de forma dinámica hay cambios a la hora de llamar a sus funciones miembro. En lugar de usar el operador punto (`.`), como lo que se tiene es un puntero a un objeto (`Point *`), se usará el operador flecha (`->`). Esta es una de las novedades de C++ y que ya se verá como dará una gran potencia de reusabilidad.

Destrucción de objetos dinámicos

Cuando se acaba de usar el objeto dinámico, no se debe de olvidar de **destruirlo**. Para ello, se usará la sentencia `delete`, que se encargará de llamar automáticamente al destructor, en caso de estar definido. Es importante indicar que si el destructor no está definido no se le podrá llamar. Si ahora se pone en el `.cpp`, siguiendo con el ejemplo anterior:

```
Point::~Point() {
      delete[] v;
}
```

Se liberará la memoria reservada en el constructor dejándola libre para otros usos. Hacer esto es **muy importante** para no sobrecargar el sistema.

Ver el siguiente ejemplo completo.

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

using namespace std;
using namespace space;

int main() {
    // memoria dinamica objetos
    Point *P = new Point;
    Point *P1 = new Point(10, 10);
    P1->display();
    delete P;
    delete P1;

    Point *P2 = new Point;
    for (int i = 0; i < P2->size; i++) {
        cout << "v[" << i << "] = " << P2->v[i] << endl;
    }
}
```

En el ejemplo, se crean varios objetos de la clase `Point` usando `new`. Después, se muestran sus valores y después se destruyen los objetos. Tener en cuenta que para liberar la memoria reservada para el objeto no hay que poner los corchetes (`[]`) como en el vector del ejemplo anterior. Posteriormente se crea otro objeto de forma dinámica y se accede a los elementos del vector dinámico creado en el interior de la clase.

Para avanzados

Si se quiere reservar memoria para una matriz debe tenerse en cuenta ciertos aspectos a la hora de la reserva y la liberación de la memoria. Para crear una matriz dinámica de NxM elementos sería:

```
double **matrix = new double*[n];
for (int i = 0; i < n; i++) {
      matrix[i] = new double[m];
}
```

Y para liberar su memoria se haría al revés:

```
for (int i = 0; i < n; i++) {
      delete[] matrix[i];
}
delete[] matrix;
```

2.2. Referencias de objetos

Para ver como se usan las referencias de objetos el ejemplo típico es mediante el paso de objetos dinámicos a funciones.

Una vez se tiene creado el objeto de forma dinámica, ¿cómo se pasa a otra función?. A continuación se puede ver un ejemplo de uso.

```
void modify(Point *P, int v) {
      P->setV(v); // implementar el metodo setV
}

main() {
      Point *P = new Point(10, 10);
      modify(P, 11);
      cout << P->getV() << endl; // implementar el metodo getV
      delete F;
}
```

En el ejemplo, la función recibe un puntero a un objeto de la clase y lo modifica en el interior. Otra forma alternativa de escribir la función hubiese sido.

```
void modify(Point *P, int v) {
      (*P).setV(v);
}
```

Mediante `(*P)` se accede al contenido del objeto y no a la referencia, con lo que se puede usar el operador punto (`.`) en lugar del operador flecha (`->`).

Pero se puede ir un poco más allá, se podría tener la siguiente función:

```
void modify(Point & P, int v) {
      P.setV(v);
}
```

En este caso se está usando la notación de **paso por referencia** de C++. Como se ve, es posible intercambiar entre las notaciones de C y C++. Para poder llamar a la función, se tendría que haber escrito:

```
main() {
      Point *P = new Point(10, 10);
      modify(*P, 11);
      cout << P->getV() << endl;
}
```

Con el que se obtendría el mismo resultado.

Ejercicios

- Utilizando la librería estándar que se ha visto en las prácticas anteriores, escribir un programa en C++ que lea un array de *n* enteros por teclado y muestre la suma de todos ellos. El tamaño del array (*n*) debe ser dinámico y se pedirá al usuario por el teclado al principio del programa.
- Completar la implementación de las clases necesarias para realizar la base de datos de personas. Para ello, se debe definir:
    - Una clase `PersonList` en un fichero `PersonList.h`, y su implementación en un fichero `PersonList.cpp`.
        - Deberá contener los siguientes atributos privados:
            - Vector de tipo `Person`: `_v`.
            - Número de personas: `_n`.
        - Deberá contener los siguientes métodos públicos:
            - Constructor vacío, parametrizado y de copia. Realizarán las inicializaciones pertinentes y las reservas de memoria.
            - Destructor de la clase. Liberará la memoria reservada.
            - Método `readData`: lee el número de personas a introducir en la base de datos y seguidamente los datos de cada una obteniéndolos por teclado.
            - Método `display`: muestra la lista de personas incluidas en la base de datos.
            - Operador `[]`: que devuelva el elemento `N` de tipo `Person` solicitado.
- Realizar un fichero `main.cpp` en el que la función `main` deberá realizar lo siguiente:
    - Instanciar un objeto de tipo `PersonList` y llamar a su método `readData`, pedirá al usuario que introduzca por teclado el número de personas a introducir, se reservará la memoria de manera dinámica para esas personas y se pedirá por teclado los datos de cada una de ellas, seguidamente se llamará al método `display`, el cual mostrará por pantalla el contenido de la base de datos con todas las personas introducidas.