Universidad Carlos III de Madrid

Industrial Informatics I Lab Session

Bachelor in Industrial Electronics and Automation

2016-2017

Lab 3

Teaching staff:
Juan Carlos González Victores
Mohamed Abderrahim

Lab 3 – Classes II: Methods, Dynamic Allocations and References

1. Methods

1.1.1. Constant Methods

In C++, you can use the `const` modifier to define a **constant function**. Look at the function `getVal()` in the following example.

```
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;
};
```

By declaring a function as a constant, we are telling the compiler that the function does not modify the content of an object from within it. That is, this function can access values, but only for displaying it or using it for calculation or even for sending it as return value, but would **not change** the state of the object, i.e. none of the internal variables of the class it belongs to would be changed.

But what is it useful for? Look at the following example.

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

In class `MyCC`, through the member function `readFromMyC`, the value of the variable `_val` is assigned to the same value as contained in the variable `_val` of class `MyC`. Observe the parameter declaration `const MyC & M`, i.e. `M` is a constant variable. This means that the internal variables of object `M` cannot be modified from within the function `readFromMyC()`.

When an object is passed as a constant to a function, the compiler checks that each of its member function called from within the current function is also constant. Therefore, if the function `getVal()` of `MyC` is not defined as a `const`, then it cannot be called from the within the function `readFromMyC`. Or putting it another way, you can only make calls to `const` functions of objects that behave as constants. So, the code on the following snipped would produce compilation error:

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

But there exists a limitation. It is said that when a function is defined as a constant, you cannot change the value of the object. Therefore, within a constant function, you can only make calls to other functions that are constant.

For example, the following code would produce compilation error because the member function `foo()` is declared as a constant, and a call to a non-constant function is made from within it. Although the called function `getValueNoConst()` does not modify the object `_myc`, the compiler does not know this a priori. The compiler, in this case, takes the "safety first" approach and says, "since `getValueNoConst()` is not constant, it may or may not modify the object, and so it is not permitted".

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

So, what would then be the solution? Make a call to a constant function. The following code demonstrates the correct way of doing it:

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

2. Dynamic Memory and References

2.1. Memory allocation and deallocation

Dynamic memory allocation is done using the directive `new`, and the same can be released using the directive `delete`. Include the following in the `.h` file of the Point class, as a `public` variable:

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

Now, add the following in the empty constructor:

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

As you can see, it creates a vector of integer of 2 items, and in the empty constructor of the class, it allocates memory and initialize its value.

The general syntax for memory allocation in C++ is:

```
data_type *var = new data_type[number_of_elements];
```

The good thing about `new` is that it can allocate memory based on the user's requirement. Try the code in the following snippet:

```
#include <iostream>

using namespace std;

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

Additionally, the same example could be applied within a class. If we go back to the `Point` class and add a `public` variable `int *v`, then the memory of the vector can be allocated and initialised from within the constructor of the class. The code in the following snipped demonstrates the same.

```
///////////////////////////
.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;
    }
}
```

Dynamic Object Creation

Besides being able to allocate memory for the basic types, you can also dynamically allocate memory for objects. Following is an example of the same:

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

With this, it is dynamically creating an object of type `Point`. The `new` operator would be the one responsible for making a call to the respective constructor, in this case, to the empty constructor. However, it could also be written in the following way, so that a call to a parameterized constructor is being made.

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

Now, when an object is created dynamically, there is a change in the way a call to its member function is done. Instead of using the dot operator (`.`), we now need to use the arrow operator (`->`), since we now have a pointer to an object (`Point *`). This is one of the new features of C++.

Destruction of dynamic objects

Whenever you create an object dynamically, do not forget to **destroy** it. To do this, you can use the directive `delete`, which automatically calls the destructor, if defines. It is important to note that if the destroyer is not defined, then it cannot be called. Now include this in the `.cpp` file of `Point` class, as shown in the following example:

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

This releases the allocated memory, which makes it available for other programs to use. Doing this is **very important** so as to avoid overloading the system.

See the complete example below.

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

using namespace std;
using namespace space;

int main() {
    // dynamic memory objects
    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;
    }
}
```

In the above example, two objects of the class `Point` are created using `new`. One of the object's values is displayed, and then both the objects are destroyed. Note that to release the allocated memory of the objects, brackets (`[]`) are not used, as was the case in the previous vector example. Later, another object is created and then its array elements, which are dynamically created within the class, are accessed and displayed.

Advanced

If you want to allocate memory for an array, then certain aspects must be considered when reserving and releasing memory. A dynamic array of size NxM elements can be created by:

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

And the reverse for freeing their memory:

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

2.2. Object references

A typical example to see how to use an object references is by passing a dynamic object to a function.

Once you have created an object dynamically, how could you pass it as an argument to another function? This is demonstrated in the following example:

```
void modify(Point *P, int v) {
      P->setV(v); // implement the method setV
}

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

In the above example, the function receives, as a parameter, a pointer to an object, and then it modifies the content of the object from within the called function. An alternative way of writing the same function is as follows.

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

By `(*P)`, you can access the content of the object, rather than the reference, so now you can use the dot operator (`.`) instead of the arrow operator (`->`).

But if you can go a little further, you could have the following function:

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

In the above example, we are using the notation of **passing by reference**, as in C++. As shown, it is possible to switch between the notations of C and C++. Function call for the same is done in the following way:

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

With this, you would have the same result as before.

Exercise

- Using standard library, as done during the previous lab sessions, write a program in C++ to read an array of n integers from the keyboard and display its sum. The size of the array (*n*) should be determined dynamic by the user, via the keyboard, at the start of the program.
- Complete the implementation of the needed classes for the *people database* program. To do this, define:
    - A class `PersonList` in file `PersonList.h`, and its implementation in file `PersonList.cpp`.
        - It must contain the following private attributes:
            - Vector of type `Person`: `_v`.
            - Number of persons: `_n`.
        - Must contain the following public methods:
            - Empty, parameterized and copy constructors. Perform the appropriate memory allocations and initializations.
            - Destructor of the class. Free the reserved memory.
            - Method `readData`: Read the number of people to be entered into the database and then read each person's data via the keyboard.
            - Method `display`: Display the list of people in the database.
            - Operator `[]`: Return the requested `N`th element of type `Person`, from the database.
- Create the `main.cpp` file, in which the `main` function performs the following:
    - Instantiate an object of type `PersonList` and call its method `readData`, which asks the user to enter, via keyboard, the number of people to be entered into the database. Dynamically reserve memory for the number of people to be entered into the database, and then ask the user to enter, via keyboard, data of each person. Then call the method `display`, which should show on screen, the complete content of the database, including data of each person entered into the database.