Home Personal projects About me

My C++ & Qt blog – Cesc M.

Software engineer & C++ enthusiast

Orthodox canonical class form

This post is more about the basics of C++ and the last things C++11 brings to the orthodox canonical class form (OCCF). Some times the basics of a programming language are assumed by everybody but there is always a the time when someone asks why. Implement the complete OCCF is something that not always is required. In fact when you work with Qt framework and specially with widgets or inheritance from QObject is something you usually don’t care. And that’s an error.

The orthodox canonical class form before C++11

In C++98 and C++03 the OCCF had four different methods that the C++ compiler is willing to generate:

- Default constructor

- Copy constructor
- Destructor
- Copy assignment operator

A standard class should look like the following code:

```
class A final
{
    public:
        A ();
        A (const A &a);
        ~A ();
        A & operator = (const A &a);
};
```

occf.h hosted with ❤ by **GitHub** **view raw**

As you may see, this is simple and it should be nothing to worry about. In general, when we work with objects (not pointers to an object) we can assume that there is no need to overwrite all methods. When we copy an object, all its data is copied to the other using the copy constructors of their data members. That's fine and most of the time it's also enough. The problem arises when we want to work with pointers to objects, members that are pointers to objects, o more complex data structures (smart pointers, etc).

A problematic example

Imagine the following situation:

```
class B final
{
   public:
      B () = default;

```

```
  private:
    int b = 0;
};

class A final
{
   public:
      A () = default;
      ~A () { delete member; }
  private:
      B *member = new B ();
};
```

problematic.h hosted with ❤ by **GitHub** **view raw**

In the class B we can assume that we don't need to deallocate memory for the integer so we discard the destructor. The copy is really cheap so we also discard copy constructor or copy assignment operator. It means that the compiler will generate those methods for us copying data members.

Now, let's evaluate the class A. We have a member that is a pointer to a class B. In the constructor we allocate the memory and then we deallocate it in the destructor, that's fine. If we are not copying or assigning objects that's perfect but we all now that we like to copy, assign and delete objects all the time. In fact, we like to create pointers to objects and deallocate them whenever it's possible! Well, I'm joking. Maybe we don't *like* to do it all the time but we do need to so many times…

The problem implementation

Taking the code written above. We start writing and the code flows and we do:

```
A a;

```

```
 3  if (...)
 4  {
 5     A b = a;
 6     a lot of things with b
 7  }
 8  else
 9  {
10     A *c = new A (a);
11     doStuff (c);
12
13     delete c;
14  }
15
16  printMethod (a);
```

killer.cpp hosted with ❤ by **GitHub** **view raw**

We assume that all the methods we call are defined and don't do any weird stuff like deallocating memory or so. Even in that case, we have the problem in lines 7 and 13. I written both different approaches because sometimes we forget that going out of scope implies a call to the destructor of the class (line 7). Other times we explicitly allocate and deallocate memory (so we do with the destructor) like in the line 13.

In these cases, the copy constructor and the copy assignment operator generated by the compiler makes the *member* of the class A to share the address between all the objects. The reason is because C++ copies or assigns the values and in a pointer is the address it points to. So, when we are deleting the object in lines 7 or 13 we are in fact deallocating the memory of the *member* field of object a!

The solution: implement the orthodox canonical class form

In order to avoid such a nasty behavior we must define the copy methods in the class A. It means that we have to implement the full orthodox canonical class form: 4 methods before C++11.

```
class A final
{
    public:
        A () = default;
        A (const A &a) { member = new B (a.member); }
        A & operator = (const A &a)
        {
            if (this != &a)
            {
                delete member;
                member = new B (a.member);
            }

            return *this;
        }
        ~A () { delete member; }

    private:
        B *member = new B ();
};
```

full_occf.h hosted with ❤ by GitHub view raw

Doing as the code above we avoid pointers that points to the same address in multiple objects. I think it is necessary to point that in the copy assignment operator we need to check that the address we're giving by parameter is not our own address. If we don't do it, we'd be leaking memory or destroying our own object.

The orthodox canonical class form in C++11 and later

In C++11 the orthodox canonical class form has changed by including two new member functions:

- Move constructor
- Move assignment operator

Since it implies a lot of new things I prefer to explain it in its own post.

April 8, 2019 Add Comment

In C++

#C++ #OFFC #OOP

Leave a Reply

Comment

Name *

*Email **

Website

☐ *Save my name, email, and website in this browser for the next time I comment.*

Post Comment