OOP and Inheritance in C++

Martin Nestorov

January 8, 2018

Contents

The ++ in Cpp

- C with classes
  - Constructors
  - Destructors
  - Operators
  - Designs
- Inheritance
  - Inheritance models
  - Multiple Inheritance
  - Virtual functions
- Templates
  - Understanding typename
  - Meta programming
- Resource Management
- Good practices
- STL

When C++ first came out it wasn't all that different from C. Apart from having standard object orientation functionalities and inheritance capabilities, the two languages were more or less the same. But overtime C and C++ grew to become very different. In the previous tutorial we looked into how to structure out program in terms of files and how it executes. Now we will dive into the world of OOP in C++, we will revise what was though in the C++ course and will try to build upon that material by giving examples, by running code, and talking about practices for writing better C++.

First we will cover classes. What the classes implement in the background, what to do with constructors, destructors, and operators. We will look over some designs on how to build our classes for maximum utility and efficiency. Then we will go over inheritance and polymorphism, with discussions over the different models, how to use multiple inheritance, what virtual tables are, how to use virtual functions, and so on.

Classes and Objects

```
class Person
{
private: // (Used by default if not specified explicitly) No one can see in here
    int age;

    std::string name;
    std::string job;

    static const Doctor *d;

    const void wave();

protected: // Can be seen only by class that is a child
           // Usually we don't have a use case for protected
    bool is_married;

public: // Everyone can see this part
    Person();
    Person(std::string name);
    ~Person();

    std::string get_name();
```

```
    std::string get_job();
    int get_age();

    double calculate_salary();

    bool get_occupation() { return is_married; } // implicit inline function
};
```

This is a basic class. We can see the scopes of the class, what we can have as variables and functions. A quick note here - although we have 3 scopes, the ones we use the most are `private` and `public`, while `protected` is used in seldom cases, so we need not worry about it that much.

Initialization lists

```
class Library
{
public:
    ABEntry(const std::string& name, const std::list<Books>& books);
private:
    std::string theName;
    std::list<Books> books;
    int addressNum;
};

Library::Library(const std::string& name, const std::list<Books>& books)
// Initialization start righ here before the opening bracket
{
    theName = name; // these are all assignments
    theBooks = books;
    addressNum = 0;
} // This is OK but can be done better
```

```
Library::Library(const std::string& name, const std::list<Books>& books)
: theName(name), // These are all now initialized
  theBooks(books),
  addressNum(0)
{ } // Empty constructor body
```

When we are initializing this object, we would write `Person p1`, but what would happen with its data members. Will they be initialized to zero or not? We can never be sure about this and sometimes they might be zero, sometimes they might be left **uninitialized**. If we take the risk and read something that is uninitialized, then we will get an undefined behavior. There are rules to remember when something is initialized to zero and when something is left uninitialized, but this all depends if you are writing the C part of C++ or the STL part and things just get way to complex. That is why we can take the safe approach and just **always** initialize our object when we create them. If we are just working with normal variables in a function, this would be like writing `int x = 0; double b; std::cin >> b; char* x = "pointer intialization"` and so on. For object, however, this tasks is left to the constructor. The rule is simple - make sure that everything is initialized when an object is created.

In the example we can see that the Library constructor makes 3 assignments to the data members. In C++, the **initialization** takes place **right before** we get into the constructor body! This means that in order to make sure that the variables are initialized, we have to put them into an **initialization list** (how intuitive!). There are several advantages to this approach. First we make sure that we won't have uninitialized data members and will avoid undefined behavior. Secondly, because we are assigning objects (the `string` and `list` members) we are avoiding calling their assignment constructors. In the previous code, where we assigned the variables, first we had to initialize the `string` and `list` variables, then we had to assign them the new values by calling a copy assignment operator. That's way too much work. But with the initialization list, not such thing needs to happen. The name and the phone are **copy-constructed** from the passed parameters. This means that only a copy constructor of the `string` and `list` is called in order to initialize the data members, and this operations (the copy constructor) is much more efficient than the previous code.

If we want to initialize the data members to nothing or we don't have any parameters to pass to the constructor, we can just call the initialization list and leave the brackets empty.

There is one little aspect we must note here and that is - we must write the order of our initialization list in the same order we declared our data members in our class. This is just hoe C++ works, we must first initialize `theName` and then `theBooks` and the `addressNum`. If we don't do that we will get an error and undefined behavior.

Static non-local objects defined in different translation units

```
// in the filesystem.cpp file
```

```
class FileSystem
{
public:
    std::size_t numDisks() const;
};
extern FileSystem tfs;

// in our .cpp file
class Directory
{
public:
    Directory(params);
};

Directory::Directory(params)
{
    std::size_t disks = tfs.numDisks(); // we use the tfs object,
                                        // but we cannot be sure
                                        // that it is initialized
}

Directory tempDir(params);
```

Solution

```
FileSystem& tfs()         // this replaces the tfs object; it could be
{                         // static in the FileSystem class
    static FileSystem fs; // define and initialize a local static object
    return fs;            // return a reference to it
}
class Directory { ... };                // as before
Directory::Directory( params )          // as before, except references
{                                       // to tfs are
    std::size_t disks = tfs().numDisks(); // now to tfs()
}

Directory& tempDir()              // this replaces the tempDir object; it
{                                 // could be static in the Directory class
    static Directory td( params ); // define/initialize local static object
    return td;                    // return reference to it
```

```
}
```

One final thing that is worth mentioning is the case when we have to initialize non-local static objects from different translation units. What that jumble of words means is that - if we have a static object (an object that has a duration from the beginning of its initialization till the end of the program) and if that object is non-local (it's in the global scope, namespace scope, or in the scope of an other class) and if it's in a translation unit (basically it's a single source file) how do we initialize it? The problem here is that if we use such a static non-local object, we can never ever be sure when it is initialized properly and we are down the road of undefined behavior. The way to overcome such situations is by writing a function that makes that static non-local object a static local object (basically implementing the **Singleton** design pattern. . . almost). We just need to write a small function in our code, where we define and initialize a static local object of what we need and return a reference to it. This way we make sure that every time the function to get the object is called, the object will **always** be initialized.

Constructors

```
// Empty class
class A {};

// Empty class again
class A
{
public:
    A() {...}              // constructor
    A(const A& rhs) {...} // copy constructor
    ~A() {...}             // destructor

    A& operator=(const A& rhs) {...} // copy assingment operator
};
```

All of the classes we write have at least several constructors in them which will be either user defined or put in the class by the compiler. Sometimes we don't have to explicitly tell C++ what kind of constructors we want, because our program doesn't requite such functionality, but sometimes we have to implement our own constructors and operators (**operator overloading**).

It's a good thing to know that the compiler is implementing instead of us when we are not writing the constructors ourselves.

Let's take this example for instance. Here we have a class that has nothing in it and we have a class that has four constructors implemented. There is virtually no difference between writing the empty class without constructors and the one with the 3 ctors and one operator implemented. The 3 cotrs and 1 operator that we will always have by default are - a default constructor, default destructor, copy constructor, and a copy assignment operator. In general there isn't much difference between what you will write as an implementation and what the compiler will write for there things, but keep in mind that in some specific cases you will be forced to write your own constructors. This will be the case when you have some constants and/or some references in your class. Why references? Because when you are copying an instance and you have a pointer in it, you would want to copy what the pointer points to (its content) and not the pointer only. And the case for `const` is that you would want to keep **const correctness** when copying. This is not a problem if our class has normal data members.

```
template<typename T>
class NamedObject
{
public:
    NamedObject(std::string& name, const T& value);
private:
    std::string& nameValue;
    const T objectValue;
};
```

In this case when we have a reference, we just have to make sure that we are implementing a **deep** copy when we are writing the constructor or operator. But if our class has a `const` then we are in trouble. Because we cannot change the value of the initial object we want to re-assign, we cannot implement a proper copy assignment operator. In the case with the constant data member, a default copy assignment operator will not be implemented, and if we were to implement one of our own, we would not copy the `const` data members.

References

- Copy constructors, assignment operators, and exception safe assign-

ment

- The Problem with const Data Members
- Copy assignment