OOP and Inheritance in C++

Martin Nestorov

January 8, 2018

Contents

The ++ in Cpp

- C with classes
  - Classes and Objects
  - Constructors and Operators
  - Designs
- Inheritance
  - Inheritance models
  - Multiple Inheritance
  - Virtual functions and destructors
- 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.

C with classes

Classes and Objects

```
 0  class Person
 1  {
 2  private: // (Used by default if not specified explicitly) No one can see in here
 3      int age;
 4
 5      std::string name;
 6      std::string job;
 7
 8      static const Doctor *d;
 9
10      const void wave();
11
12  protected: // Can be seen only by class that is a child
13             // Usually we don't have a use case for protected
14      bool is_married;
15
16  public: // Everyone can see this part
17      Person();
18      Person(std::string name);
19      ~Person();
20
21      std::string get_name();
22      std::string get_job();
23      int get_age();
```

```
24
25      double calculate_salary();
26
27      bool get_occupation() { return is_married; } // implicit inline function
28  };
```

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

```
 0  class Library
 1  {
 2  public:
 3      ABEntry(const std::string& name, const std::list<Books>& books);
 4  private:
 5      std::string theName;
 6      std::list<Books> books;
 7      int addressNum;
 8  };
 9
10  Library::Library(const std::string& name, const std::list<Books>& books)
11  // Initialization start righ here before the opening bracket
12  {
13      theName = name; // these are all assignments
14      theBooks = books;
15      addressNum = 0;
16  } // This is OK but can be done better
```

```
0  Library::Library(const std::string& name, const std::list<Books>& books)
1  : theName(name), // These are all now initialized
2    theBooks(books),
3    addressNum(0)
4  { } // 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

```
0  // in the filesystem.cpp file
1  class FileSystem
2  {
```

```
 3  public:
 4      std::size_t numDisks() const;
 5  };
 6  extern FileSystem tfs;
 7
 8  // in our .cpp file
 9  class Directory
10  {
11  public:
12      Directory(params);
13  };
14
15  Directory::Directory(params)
16  {
17      std::size_t disks = tfs.numDisks(); // we use the tfs object,
18                                          // but we cannot be sure
19                                          // that it is initialized
20  }
21
22  Directory tempDir(params);
```

The solution:

```
 0  FileSystem& tfs()         // this replaces the tfs object; it could be
 1  {                         // static in the FileSystem class
 2      static FileSystem fs; // define and initialize a local static object
 3      return fs;            // return a reference to it
 4  }
 5  class Directory { ... };                  // as before
 6  Directory::Directory( params )            // as before, except references
 7  {                                         // to tfs are
 8      std::size_t disks = tfs().numDisks(); // now to tfs()
 9  }
10
11  Directory& tempDir()               // this replaces the tempDir object; it
12  {                                  // could be static in the Directory class
13      static Directory td( params ); // define/initialize local static object
14      return td;                     // return reference to it
15  }
```

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 and Operators

```
 0  // Empty class
 1  class A {};
 2
 3  // Empty class again
 4  class A
 5  {
 6  public:
 7      A() {...}              // constructor
 8      A(const A& rhs) {...} // copy constructor
 9      ~A() {...}             // destructor
10
11      A& operator=(const A& rhs) {...} // copy assingment operator
12  };
```

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.

```
0  template<typename T>
1  class NamedObject
2  {
3  public:
4      NamedObject(std::string& name, const T& value);
5  private:
6      std::string& nameValue;
7      const T objectValue;
8  };
```

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.

Designs

Write OOP classes

```
0  class AccessLevels
1  {
2  public:
```

```
 3      int get_read_only() const { return readOnly; }
 4      void set_read_write(int value) { readWrite = value; }
 5      int get_read_write() const { return readWrite; }
 6      void set_write_only(int value) { writeOnly = value; }
 7  private:
 8      int no_access;    // no access to this
 9                       // (maybe it's used as an internal constant)
10      int read_only;    // only read access
11      int read_write;   // read and write access
12      int write_only;   // write only access
13  };
```

Although C++ gives us a lot of flexibility with what we can put into a class and under what type of scope, there have been several great designs that usually make our work much easier and allows us to quickly create our classes. This doesn't mean that we have to neglect what we are actually writing, but for most of the time we can follow these conventions.

First of - declare your data members as private (not even protected). Okay so why do this? First off, by making our data members private we are limiting the access to them, we can only **get** them and **change** them only through functions. This allows us for finer control over our data (REGAIN CONTROL OVER YOU DATA) and will also follow some consistency when writing classes. We won't need to bag our heads whether we are chaining something or we are writing to something or whatever. The general rule is - every data member should be hidden, or - show only as much as you need. This means that some of our data members won't even need `getters` and `setters`, thus reducing our chances of bugs and lines of code. The last, and biggest, benefit to hide your data is **encapsulation**. This brings a lot of flexibility of implementation to the table and we should strive to make our classes as flexible and encapsulated as possible. A rule we can remember about this is - the more a class is used, the more it needs to be encapsulated. The same rules apply for `protected` data members. It might seem that they are encapsulated on first glance, but think about this - if we are to remove a **protected** data member, we are going to break and **undefined** amount of inherited classes, which is always more than what we will break if we just had that member as **private**. Seldom are the situations where `protected` will save the day.

Treat class design as type design

- How should objects of your new type be created and destroyed?
- How should object initialization differ from object assignment?
- What are the restrictions on legal values for your new type?
- What operators and functions make sense for the new type?
- What standard functions should be disallowed?
- Who should have access to the members of your new type?
- Is a new type really what you need?

One of the things that will happen when you are writing your FDS project is that you won't have an idea what to do in the beginning and how to start. Hence you will hesitate how to build your classes and what to do with them.

Writing classes isn't easy, because in C++ we are not creating only a class, but we are also creating a **user type**. This means that we have to take care of what that type must to do and what it must **not** do. That's why we can ask ourselves the following question in order to get a better grip of the classes we are writing. In consequence we will get a better understanding of the program we are writing.

How should objects of your new type be created and destroyed? - How this is done influences the design of your class's constructors and destructor, as well as its memory allocation and deallocation functions.

How should object intialization differ from object assignment? - The answer to this question determines the behavior of and the differences between your constructors and your assignment operators. It's important not to confuse initialization with assignment, because they correspond to different function calls.

What are the restrictions on legal values for your new type? - Usually, only some combinations of values for a class's data members are valid. Those combinations determine the invariants your class will have to maintain. The invariants determine the error checking you'll have to do inside your member functions, especially your constructors, assignment operators, and "setter" functions. It may also affect the exceptions your functions throw and, on the off chance you use them, your functions' exception specifications.

What operators and functions make sense for the new type? - The answer to this question determines which functions you'll declare for your class. Some functions will be member functions, but some will not.

What standard functions should be disallowed? - Those are the ones you'll need to declare private.

Who should have access to the members of your new type? - This question helps you determine which members are public, which are protected, and which are private. It also helps you determine which classes and/or functions should be friends, as well as whether it makes sense to nest one class inside another.

Is a new type really what you need? If you're defining a new derived class only so you can add functionality to an existing class, perhaps you'd better achieve your goals by simply defining one or more non-member functions or templates.

Prefer pass-by-reference-to-const to pass-by-value

References

- Copy constructors, assignment operators, and exception safe assignment
- The Problem with const Data Members
- Copy assignment