C++ a more in depth look into the language

Martin Nestorov

January 8, 2018

Contents

Who am I?

- Martin Nestorov
- Junior 2nd sem.
- I like to type . . . a lot
- email: mdn150@aubg.edu
- Twitter: @mnestorov

Why I chose to make these small tutorials

- People seem to have some problems with the transition from C++ to FDS.
- I want to help!
- I want to also learn!

Over the course of the past several years I found a certain love for C++, but instead of just reading about the language and doing some small experiments with it, I wanted to share my knowledge and to get better at it. Seeing how some students struggle with C++ in the beginning courses, I decided that I can help out. That way not only am I learning more in-depth concepts, but also I am making the lives of students easier (hopefully).

How to look at C++

C++ as **four** sub-languages

- C
- OOP
- STL
- Templates

There is an old joke about C++ - "How do you shoot yourself in the foot? In C you just shoot yourself in the foot. In C++ you accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me, over there."

C++ can sometimes be vary annoying, because we have absolutely no idea how it works and what it does. The fact that having very similar syntax for wildly different things doesn't help either. So instead of just banging our heads until bleeding sets in, we can try to understand a little bit better what is happening with this language and (hopefully) try to minimize **undefined** behavior. In fact C++ is that special language where if you don't know that you are doing, the language re-enforces that by acting in a way where it doesn't know what it might do. That's pretty scary. That's why I will try to minimize this effect with these tutorials and help you out to make better C++ programs and to introduce effective methods or writing C++ code.

C++ can be generally divided into four sub-categories or sub-languages. It's a daunting task to go over all of these aspects and trying to teach everything, mainly because I can't! But instead I will try to cover some topics that haven't been covered in depth in the C++ lectures and are somewhat elude to the students. Now our list seems like this

What we are going to cover

- File structure
- OOP
- Inheritance
- Templates

File structure

- Advantages:
  - Better code organization
  - Faster compiling
  - Separation of **implementation** from **interface**
- Disadvantages:
  - More complicated structure

It has come to my attention that many people, while writing their homeworks, neglect the advantages of structuring their **source** files. This is most noticeable when they are writing the first 3 FDS homeworks or their FDS project. Being able to properly manage your project into several files and to navigate between them in an optimal and cohesive way will minimize your C++ suffering (and coding experience in general).

Now most of you already know this, but there are several great advantages to separating your **implementation** (cpp) files from your **interface** (h) files. One advantage is that this will bring code clarity to your program. Instead of navigating through one big **400** line `Source.cpp` file, and wondering where that one definition of that object is and then going back to the `main()` method to fix one line, you will have a nicely structured system. Every class will have it's own implementation and interface file, thus minimizing code searching and optimizing code writing! But that's not all. This separation will allow you to compile your programs even faster. That is, every **source** file is compiled on its own. When you make a small change in your original 400 line monstrosity, you have to wait for all of that code to **re-compile**, but if you have the files separated, you just have to re-compile only the changes files, making waiting time much smaller (naturally this effect can be seen when compiling large programs, while small ones won't have such noticeable impact on time). And lastly, we also gain the benefit of implementation from interface separation.

But what we are sacrificing for these benefits is that we are increasing our project complexity. This shouldn't scare us that much, but we do have to keep this in mind - **We have to limit ourselves not to over layer our program**.

Since most of you are working with Visual Studio, here is how your structure looks like now: Picture 1

And here is how you would want it to look: Picture 2

Now I am not going to tell you how to add files and where to put them, I will just show you how to properly implement header and source files and how to avoid collisions and also how to stop writing those damn "xfz.h" files.

First things first. What's the difference between headers and sources? Apart from the obvious name difference, header files are considered to be the **interface** part of the source code, that is, the place where we explain what the program does, but **not** how it does it. The source (cpp) files are the **implementation** part, this is where we explain **how** we implement the interface. Keeping them separate allows us to keep a clean directory and structure and to separate our logic from our implementation.

How to work with headers?

```
 0  #include <iostream> //header we always include
 1  #include "awesomeclass1.h"
 2  #include "awesomeclass2.h"
 3  #include "awesomeclass3.h" // our own class headers
 4
 5  int main()
 6  {
 7      awesomeclass1 ac;
 8      awesomeclass2 ac2;
 9      awesomeclass3 ac3;
10
11      ac.do_something(ac2);
12
13      ac3 = ac;
14
15      return 0;
16  }
```

So what are header files and what do they do?

Header files, usually ending with the `*.h` file extension (can be also `*.hxx`, `*.hpp`, or nothing at all), are code files that are **included** in our source files. We can look at header files as code that only serves as the interface to our classes. That is, it holds a declaration which we later on implement and/or include in our source files. Although we will talk about header files and how we `#include` them later on, what we can say now is that for every header file that is included, the contents of that file are put in the place of the `#include` keyword. What we can expect header files to do is to hold **declarations**

inside of them. They just tell us what we are going to implement, but not **how** we are going to implement it (that's the job of the source files).

NOTE: that for the `iostream` we used angle brackets and for our classes we used quotes. This just means that the header file **iostream** comes with the compiler and that the linker should look in the system directories for the definition. The quotes indicate that we are providing the header file and that the linker should look at the build directory.

Header guards

```
 0  // x.h
 1  class x { . . . };
 2
 3  // a.h
 4  class a { X x . . . };
 5
 6  // b.h
 7  class b { X x . . . };
 8
 9  // source.cpp
10  #include "a.h" // include x.h for the first time
11  #include "b.h" // error! include x.h again!
```

```
0  // x.h
1  #ifndef __X_H_INCLUDED__
2  #define __X_H_INCLUDED__
3
4  class x { . . . };
5
6  #endif
```

Because we are going to be using a lot of headers in our program, we would need to include all of them. But because these files are actually copied when we **include** them, we would have several problems. One would be the code duplication. Another one will be the multiple declarations of classes and functions. To overcome this problem we would need to **guard** our header files in order not to use them more than once in our program. That is why we use the directives `#ifndef`, `#define`, and `#endif`. When we wrap our header file around with these directives, we ensure that the preprocessor will not copy a single file more than once into the source file.

Declarations vs Definitions

```
 0  // declarations
 1  extern int a;
 2  extern char b;
 3  class foo; // extern not allowed for user types
 4  double sum(double, int);  // no need for extern
 5
 6  // definitions
 7  int a; // the implementation of a
 8  char c; // the implementation of c
 9  double div(double a, int x) { return a / x; }
10  class foo { . . . };
```

I want to take some time now to revisit two old concepts before we continue - namely I want to talk about **declarations** and **definitions**. Here we will mention the role of the **compiler** and **linker** briefly, but we will cover them later on. Now a **declaration** introduces an identifier and its type, be that type of a variable, object, or a function. Basically we are saying to the compiler that there is something with this name and this type, but we are not specifying **how** this specific thing is implemented. This is what the **compiler** needs in order to make a reference to that declaration. We can imagine the situation where we have a function which we want to use in multiple files in our program, it's just a very useful function, but we don't want to provide the implementation of the function in every source file. That's why we just declare the function, the compiler sees the declaration, makes a reference to it without the need to know its implementation, and then when we build the final executable, the linker links every reference of the declared function to its implementation. If you remember from the C++ course, a function prototype is a declaration.

A **definition** in C++ is when we a define something in its complete form. This means that we need to both declare something and define it at the same time. We actually implement that specific identifier we have declared. When we write functions and give them a function body at the same time, that is a form of definition. When we write `int x;` we are both declaring and defining the variable `x`. This means that we are telling the compiler that there is a variable `x` of type `int` that is in the global scope in the current source file. It's important to note that giving this variable a value is not necessary in order to define it. But in the examples we have this specific keyword `extern` that is associated with declarations. What `extern` does is that it marks the variable and tells the compiler that this is only a

declaration and that a definition will be found somewhere else. In general we use `extern` when we want to declare global variables in header files and **define** them in some source file.

We can **declare** something as much as we like in as many files as we want, but we can only **define** something only **once**. Often when we get errors telling us that something is undefined (this is at linker time), what that means is that we have something declared, but we have not defined its implementation anywhere and we get a **missing symbols error**. On the other hand when we have multiple definitions of the same thing, the linker is confused as to which specific definition we want and we get a **duplicate symbols error**.

Okay now let us go back to header files.

How and when to use headers

`class A` ==> (uses) ==> `class B`

- do nothing if: A makes no references at all to B
- do nothing if: The only reference to B is in a friend declaration
- forward declare B if: A contains a B pointer or reference: `B* b;`
- forward declare B if: one or more functions has a B **object/pointer/reference** as a parameter, or as a return type: `B MyFunction(B myb);`
- `#include "b.h"` if: B is a parent class of A
- `#include "b.h"` if: A contains a B object: `B b;`

```
 0  #ifndef __MY_CLASS_H_INCLUDED__
 1  #define __MY_CLASS_H_INCLUDED__
 2
 3  // Forward declare dependencies
 4  class F;
 5  class B;
 6
 7  // Include dependencies
 8  #include <map>
 9  #include "c.h"
10
11  // Our current class
```

```
12  class MyClass : public c
13  {
14  public:
15    std::map<int, char> matrix;  // a map object that is required by our class
16    F* f;                        // F pointer, so forward declare F
17    void Func(B& b);             // B reference, so forward declare B
18
19    friend class MyFriend;       // friend declaration is not a dependency
20                                 // don't do anything about MyFriend
21  }
22  #endif
```

At this point, after we are revised what declarations and definitions are, we can talk about how to use headers and in which situations this is a viable option. The very short answer is - use header files when you have a class, a grouping of functions that do similar things, or a grouping of global variables. In general this approach is not bad and will frailly well for you, but we shouldn't stop here. That's just the surface of it, to dig deeper we have to understand in which situations we can use header files and when to use forward declarations.

Let's imagine that we will be writing a lot of classes, where there will be a lot of dependencies. So when we talk about header files and using multiple classes we are also talking about dependencies between classes (this may also mean just a grouping of functions, but for simplicity reasons, we will refer to a header file as being a single class that does one thing). As an example, a derived class will always be dependent of its parent class, which means that we must be aware of the parent class at compile time, so we can use the derived class (such a case will be when we have a **public** inheritance between two classes). There are two general forms of dependencies - one is, that there will be things that can be **forward declared**, and the other is, that things need to be `#include`-ed. Let's take an example with the following. Image we have class `A`, which uses class `B`. This makes A dependent on B. To decide if we need to **forward declare** or to include A depends on what the interaction between the two classes is.

Some explanation is required here. First of we are protecting our class with the header guards. Then we are forward declaring two classes `F` and `B`. This means that we are telling the compiler that there is a user type class called **F** and **B** and that the implementation of the classes are somewhere else. This allows us to access the methods and variables of **F** and **B** without needing to specify their implementation. Forward declarations can be useful,

because they can reduce compilation time. If we just want to access and use one or two functions from a class, instead of including the whole class (and copying its contents to the current file) we can just forward declare it and save ourselves some time.

In general we want to remember this simple rule - prefer forward declarations to includes, in order to save compilation time and circular dependency issues. The benefits to these rules is that we are encapsulating the classes we use as much as possible, because the classes that use our classes don't need to know how they work. This means that if we restrict our use of includes and think about how the dependencies of the classes interact, we are writing encapsulated and decoupled code.

Circular Dependencies

```
0  // b.h
1  #include "a.h"
2  class B { A* a; };
3
4  // a.h
5  #include "b.h"
6  class B { B* b; };
```

I mentioned at some point about circular dependencies, but they are something that is very common when we are writing OOP code. In this example we see a circular dependency. If we were to just include both header files, we would be stuck in loop, trying to figure out which class to include first, giving us a compilation error. In order to overcome this (and keep in mind that there is nothing wrong with such circular dependencies) we can just forward declare one of the two classes and include the other so we can break from this cycle. Now before you ask "what about the situation where we have explicit object declarations instead of pointers to objects?", let's just say that such a situation is **impossible**. Not that you can't write such a program, but there is an inherit design flaw to such code, it's virtually impossible to make such a code work, and there is a clear flaw in logic if you find yourself in such a situation. The other question you might have is - well when should we use pointers to objects? This totally depends on your program. For instance, if we have a `class Car` and a `class Wheel`, it's logical for the **Car** to include the **Wheel** class, but the **Wheel** class only needs a pointer to the Car object (it doesn't need the whole Car).

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

```
12      int read_write;   // read and write access
13      int write_only;   // write only access
14  };
```

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.

In order to access are already `private` variables, we must rely on getters and setters. These are small and fast functions that do what their name suggests. It's usually a good practice to make you getters `const`, this way you are ensuring yourself that the variable won't change upon retrieval. As a reminder - not all data members need such functions to them!

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

```
 0  class Window
 1  {
 2  public:
 3      std::string name() const;     // return name of window
 4      virtual void display() const; // draw window and contents
 5  };
 6
 7  class WindowWithScrollBars: public Window
 8  {
 9  public:
10      virtual void display() const;
11  };
12
13  // the bad functoin
14  void printNameAndDisplay(Window w) // Bad! Parameter might be sliced!
15  {
16      std::cout << w.name();
17      w.display();
18  }
19
20  // the correct function
21  void printNameAndDisplay(const Window& w) // Correct! Parameter won't be sliced!
22  {
23  std::cout << w.name();
24  w.display();
25  }
26
27  // the call
28  WindowWithScrollBars wwsb;
29  printNameAndDisplay(wwsb);
```

Most of you are aware of what **pass-by-value** and **pass-by-reference** is, but here we will just discuss a few more advantages to using passing of references to `const`. One thing that happens when we are passing objects (and more specifically, **inherited** / **derived** objects from a base class) by value is that we are at the risk of **slicing** off the derived part of the object in the function call. What this means is that if we have a `class A` which inherits `class B`, and we pass an object of `type A` to a function by value (the function parameter accepts `type B`), the **A** part of that object will not be passed, and only the **B** part will remain. Okay maybe that sounds too complicated, let's revise. By passing something by value we are copying it and the function is using the copy. In the case of inherited classes, when we are passing a derived object by value the **copy constructor** of the class is called to make a copy - naturally. But when the object has a base class, the base class is copied first. If the parameter of the function is of the base class type, only the base class during the copying will be left, and the derived part will be omitted. That is why it's a better idea to pass-by-reference-to-const.

Under the compiler, **references** are passed as pointers, so we are technically just passing a pointer. That is why if we have a build-in type like `int` it is more efficient to pass by value. Also we can say that the STL **iterators** and **function objects** are also good to pass-by-value, because they have been specifically build to be more optimal to copy than to be passed as a reference (as a pointer).

Inheritance

Templates

References

File Structure

- How to use header files
- Header files
- Why do we have header and cpp files
- Declaration vs Definition
- Difference between declaration and definition
- The PIMPL idiom

- Forward declarations

Classes and Objects

- Copy constructors, assignment operators, and exception safe assignment
- The Problem with const Data Members
- Copy assignment
- Functions / Function Objects