



ROBERT HOFFMAN

C++

**THE ULTIMATE GUIDE
TO MASTER C
PROGRAMMING FAST**

C++

The Ultimate Guide to Master C Programming Fast
(c plus plus, C++ for beginners, programming
computer, how to program)

ROBERT HOFFMAN

CONTENTS

[Introduction](#)

[Chapter 1: Installing Cygwin: Linux-Like Development](#)

[Chapter 2: Installing Notepad++: Going from Text to Code](#)

[Chapter 3: Setting up the Workspace: File Structure Is Key](#)

[Chapter 4: Programming Time: Salute the World](#)

[Chapter 5: Compile Time: Getting an Executable](#)

[Chapter 6: Getting complex: Variables and Functions](#)

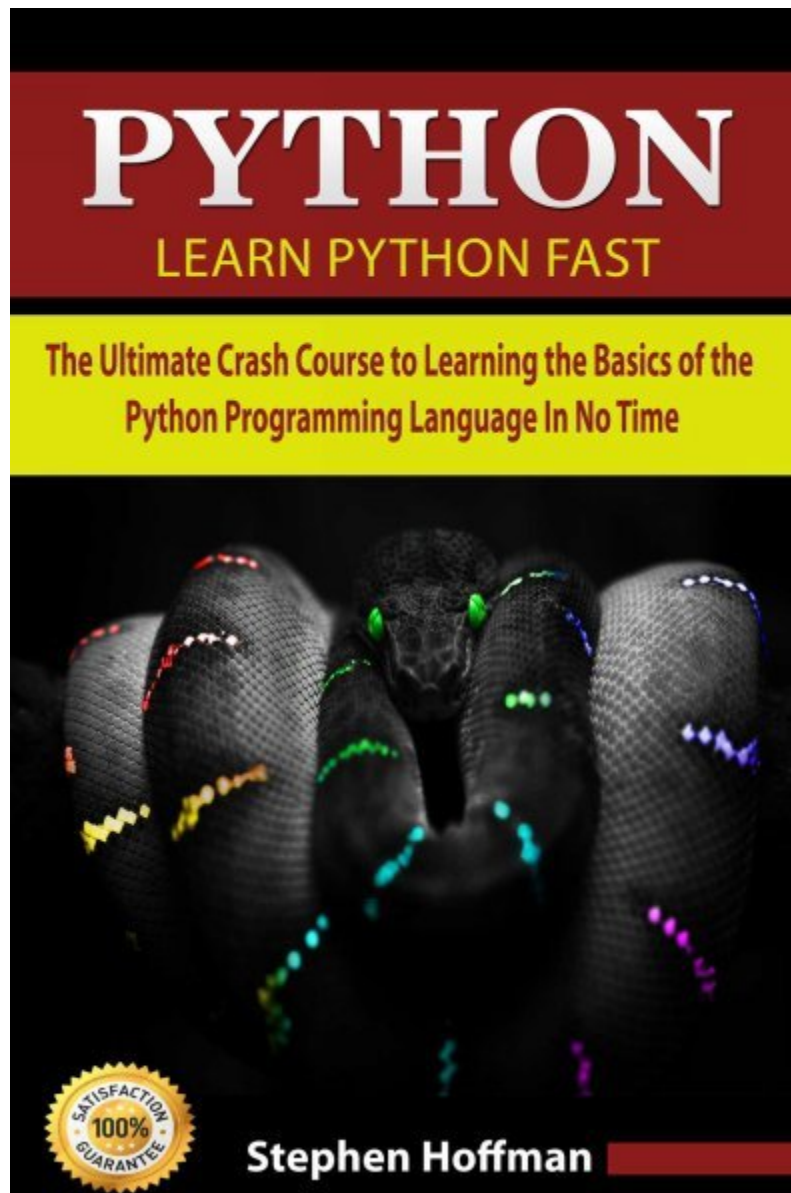
[Conclusion](#)

I think next books will also be interesting for you:

[C++](#)



[Python](#)



[Javascript](#)

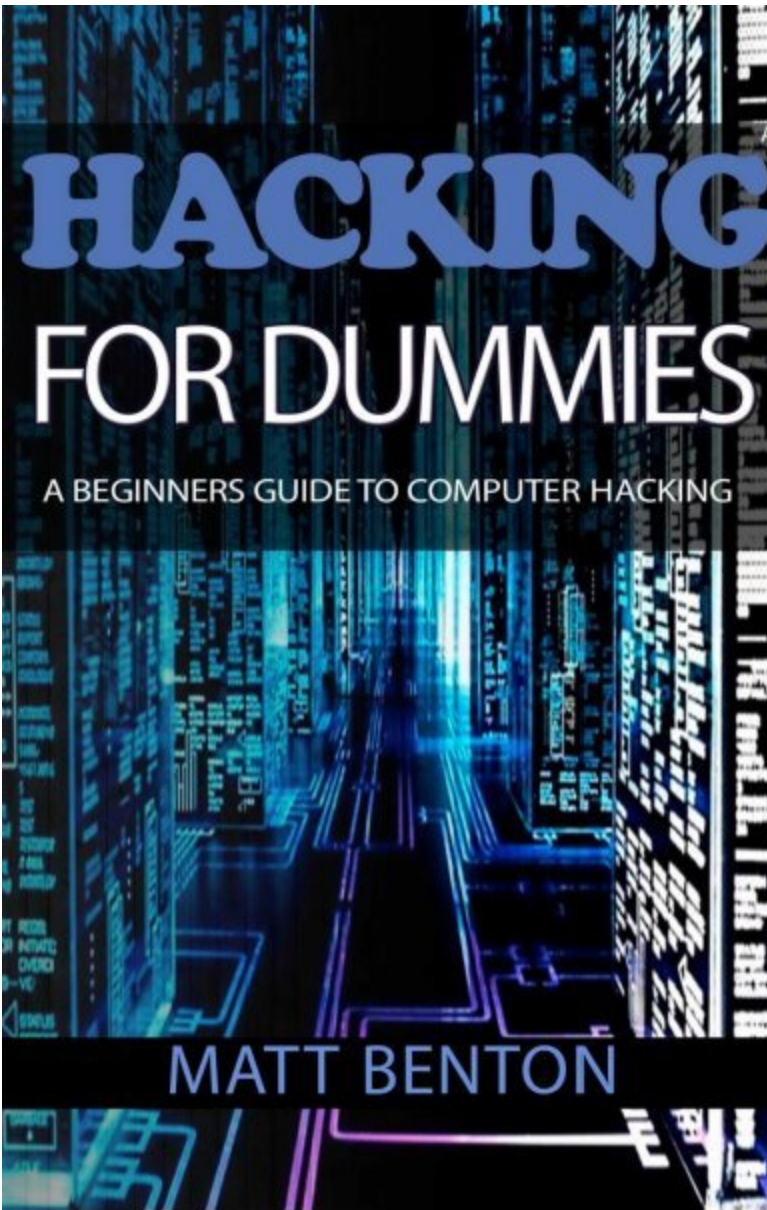
JAVASCRIPT

Ultimate Guide For Javascript Programming

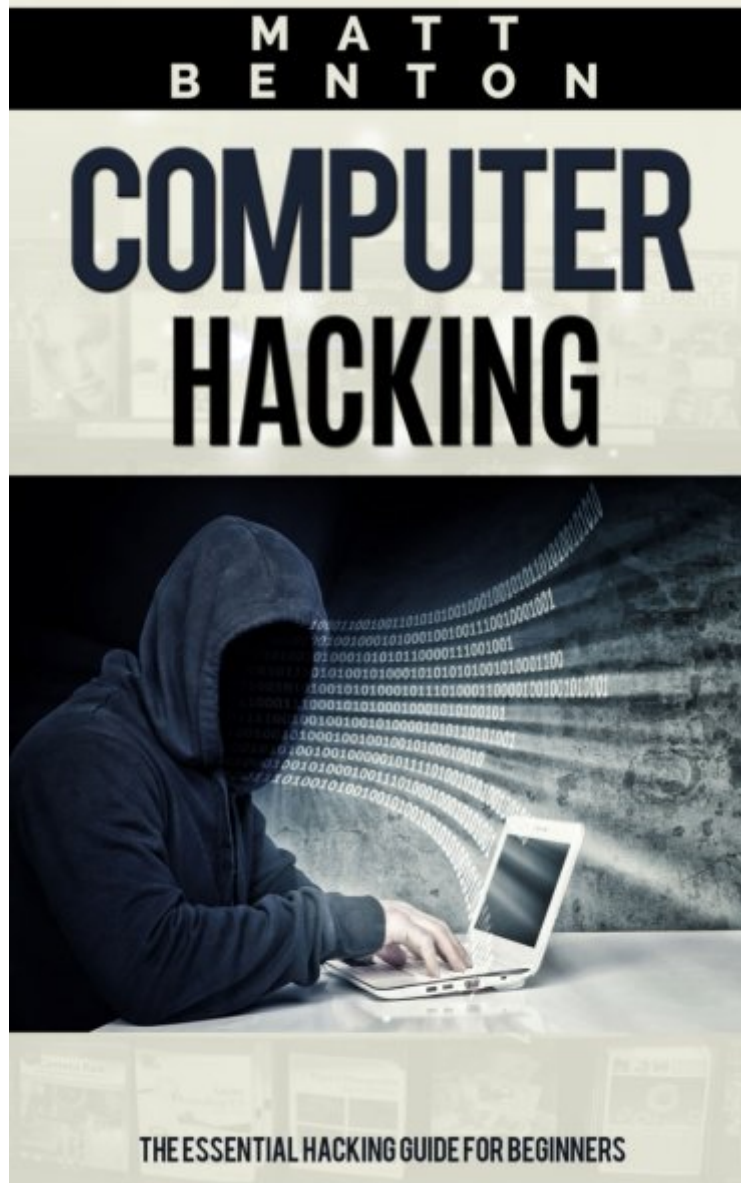


STANLEY HOFFMAN

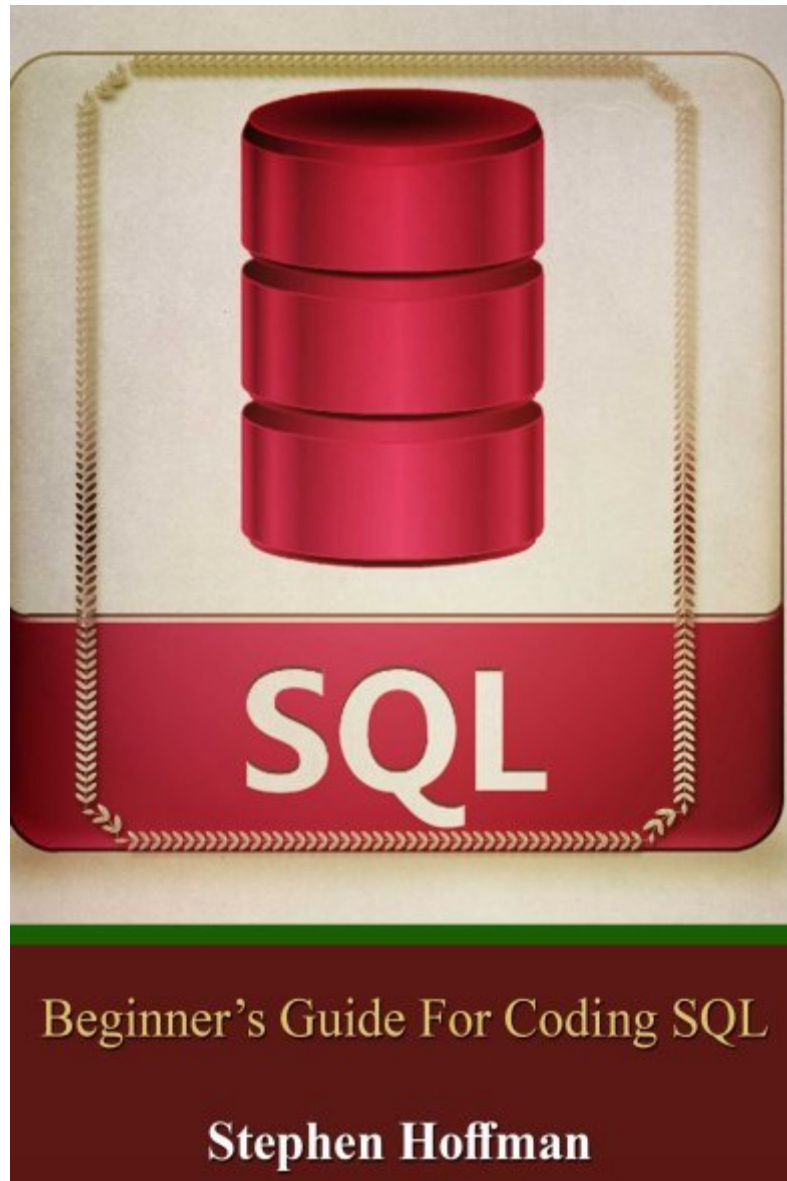
[Hacking for Dummies](#)



[Computer Hacking](#)



SQL



Introduction

Many of the applications of C++ involve more complex organizations of software pieces, including graphics and scientific computations. This is not to say that Java or other languages lack the ability to tackle these kinds of problems. C++ does have a history in these fields, with its ancestor being the C programming language. The best way to view it is that C++ added some object-oriented principles (OOP) to the C language in order to become more modern at the time.

To be more specific, C++ has a very identical syntax to C in general, but shares the OOP concepts that Java also uses. This includes classes, polymorphism, encapsulation, inheritance, and a number of other concepts. The gist is that C++ provides an easy to understand platform for both new and old programmers to communicate easily about.

This guide isn't meant to bog you down in terminology though. Once you learn any one programming language, the next will be easier so long as it's in the same family. Meaning, C++ will help you learn Java which will help you learn C# and so on. The best way to choose a programming language is to find a feel for how you might like to program. This could mean writing speech like statements as in QBASIC, or writing esoteric coding magic as in Ook-Ook. In this case, we need a language that will easily introduce the initial aspects of programming while still being flexible enough for more advanced topics; therefore, we choose the C++ programming language.

Now that we're up to speed on the general view that our programming language holds, we can start to tackle some of the preliminary objectives of setting up our programming environment. This will include an editor, a compiler and a file directory structure. The editor is where you will actually write your code. The compiler will take your code and make an executable application that will run on your computer. The only purpose of the file directory structure is to minimize confusion as to where your files are being saved while you're coding. Shall we begin?

Chapter 1: Installing Cygwin: Linux-Like Development

The first step is to get the compiler. The reason is that this will force some constraints on how your program will run after the compiler process it. An example of this is that there are some compiler specific macros, such as `__sync_synchronize` from the gcc compiler that will add some extra instructions that you don't have to add yourself.

Compiler concepts aside, there are some other issues with using different compilers with the same program. Compiler warnings will also cause some unneeded confusion whenever you're trying to find problems with your code. The selection of a compiler can also be limited because of consensus over one particular compiler causing a stagnation of further research.

Another perk to using the Cygwin package is that it follows a similar format to the default Linux compiler. The specific compiler we'll be using from the Cygwin package is the g++ compiler which will have a few quirks about it, but it'll help to get us intimate with the code well enough that we don't just hit a magical button to get executable code, like some other integrated development environments (IDE).

There's plenty of research being done about compilers in general. Concepts that are applied include formal languages, automata, and various other more advanced topics. If you have further interest as to how compilers work under the hood, there are plenty of resources available online as well as some easy to understand explanations of the most popular compilers.

[Click this link to grab Cygwin.](#)

Once that's installed, there are few other things that you can do now. The Linux operating system uses the bash different scripting language whereas Windows uses batch. The difference between bash and batch are fairly numerous, but both share some general functionality. Mentioning this, Cygwin actually adds in some of the commands that bash has, but Windows makes a little more complex.

The **pwd** command is going to be useful later, since it will print the working directory that the command prompt, or terminal, is using as its place to execute commands.

Chapter 2: Installing Notepad++: Going from Text to Code

The next thing that we'll need to download is the code editor. The difference between a code editor and a simple text editor is primarily apparent in the fact that the former will provide some helpful syntax highlighting, among other features.

Syntax of a language encompasses the keywords, grammatical structure, and whitespace parsing as well as other pieces. The purpose of syntax in coding is fairly identical to syntax applied to language in general. It provides an agreed upon structure of the form of a statement.

The keywords of a language are special words which are unique to programming. These include loop structures such as for, while, and do-while; data types are also included such as float, double, string, and so on.

The grammatical structure important in coding is primarily the placement of separators. More specifically, the way that a language would want us to form functions, what to call the entry function, and whether or not a statement requires a terminator. An example of the latter is using semicolon basically as a period in C++ and other languages.

The last mention was primarily for a language called Python. It uses tabs as an indicator for whether or not the next line is part of a function, loop, or class block. It's got a likeably different flavor than C++ and might be worth a look into after wrapping our heads around the basic programming ideas we still need to cover.

[Click this link here to grab Notepad++.](#)

The best part is that this one code editor will highlight not just C++, but plenty of other languages as well. Linux also uses a hybrid text and code editor called gedit, which is another cool point about Notepad++. Funny that we're basically just trying to make Windows more like Linux so we can program, eh?

Chapter 3: Setting up the Workspace: File Structure Is Key

The final setup step is here. This one some would say is optional; however, think about this quick scenario. Say that there's a really large project that you're tasked to help develop. There are other coders with you. They each have their own way of doing things, which can already spell some problems early on. One of them asks you for some help with his piece. He tries to explain it quickly, but you're already lost as you try to go inside of folder after folder of software. An interface here and a realization there, with some utilities in another folder way over here. See the problem?

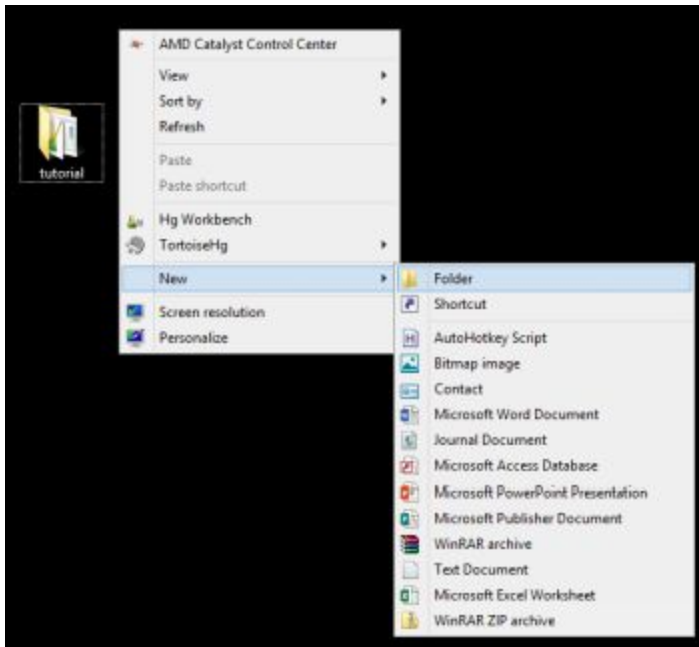
Now then. This simple example will seem to defeat the purpose of needing a strict directory structure, but the next to last chapter will have you adding a few pieces of code to your program to validate the need for a comprehensible directory structure. The most we're going to do involves 2 folders.

For the primary example, we need to make a folder called "tutorials." If you already know how to do this, then please continue to chapter 4. Otherwise, here's step-by-step how you make a new folder using Windows, though most other operating systems will follow a somewhat similar format.

The first step is to go to your Desktop. This is the place where all of your icons reside. At the bottom is the Taskbar which has the Start button and a clock on the bottom right side. Place your cursor over the Desktop and right click. You should see something similar to the image at right.

After you've selected this, the next thing we want to do is to rename the folder so we can find it easier later. To do so, all you need to do is right click the folder and click rename. You may want to left click the folder to ensure that the folder is selected before right clicking it. The picture below will show what the context menu may look like.

Though this may seem simple, there are other ways to create folders using



nothing but command line actions. In a system which doesn't have a graphical user interface (GUI), like most operating systems, then ensuring that you're putting files and folders in the correct place is of the utmost importance. The code can be as great as it wants to be, but if you or the compiler can't find it then it's as if it doesn't even exist.

One last comment on file structure is that creating it is one thing. Adhering to it throughout development is what most coders have problems doing. This means always knowing what's where at all times. If you can explain how your code is structured to someone with little to no knowledge of the system and they can find any folder or , better yet, file in the directory then we've done it right.

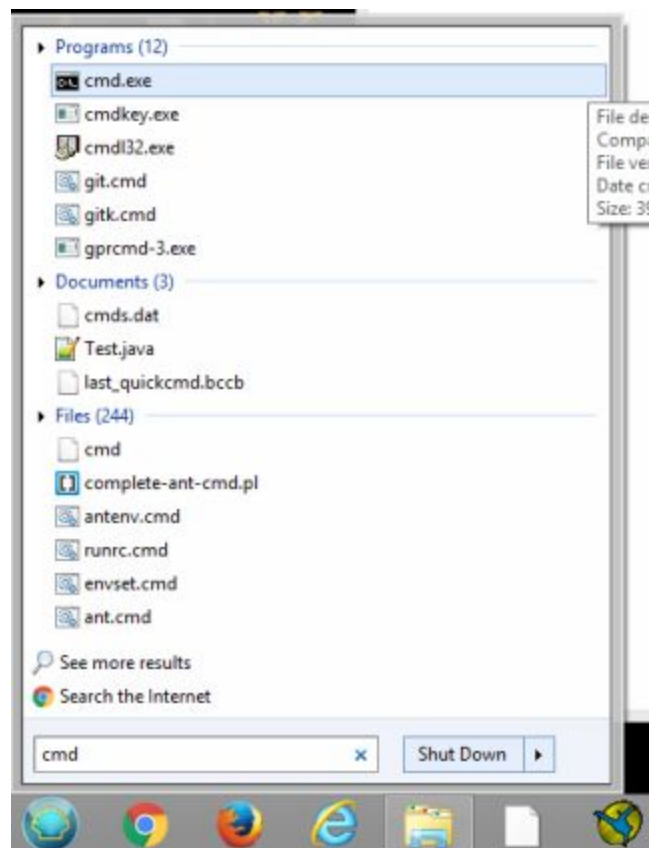
Chapter 4: Programming Time: Salute the World

Breathe in, and breathe out. Now we can program. First thing we should do is to get our command prompt into the correct working directory, which is where we put our “tutorial” folder. Let’s get started.

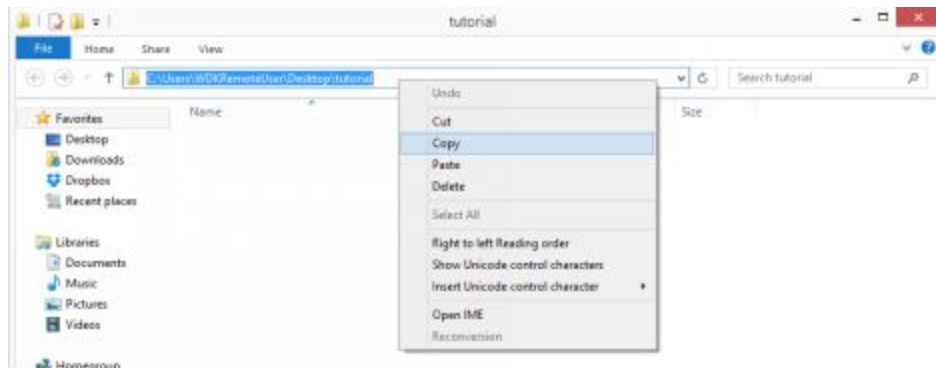
Click the Start button on the bottom Taskbar. Type in “cmd” in the search box at the bottom, or click the “Run...” option on the right side. Another way to get the Run prompt is to press and hold the Windows key and then press the “r” key. You should see the following set of screens.

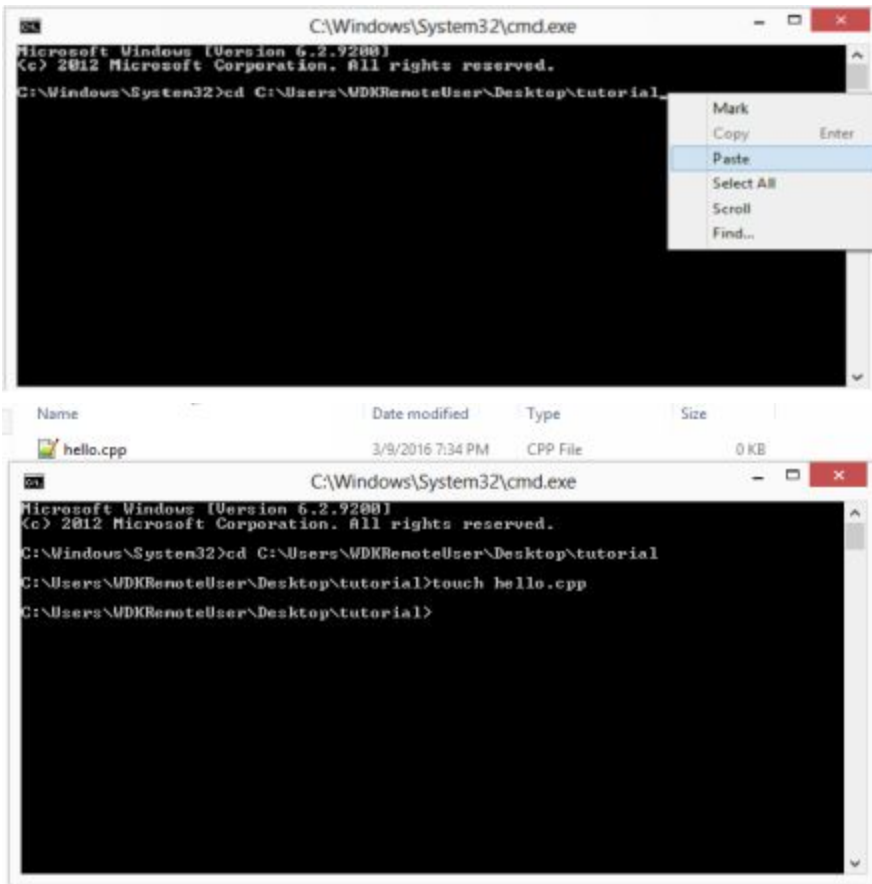
The **cd** command will change your directory to the one specified with the words following the command. To get the directory of the “tutorial” file, double click it and find the address bar at the top of the window which pops up. Click the address bar and then press [Ctrl] + [C] to copy the address from it. Then right click on the command prompt and paste the code to go to that directory. The next 2 images will show all of this in action.

The next few steps will be done in the command line.



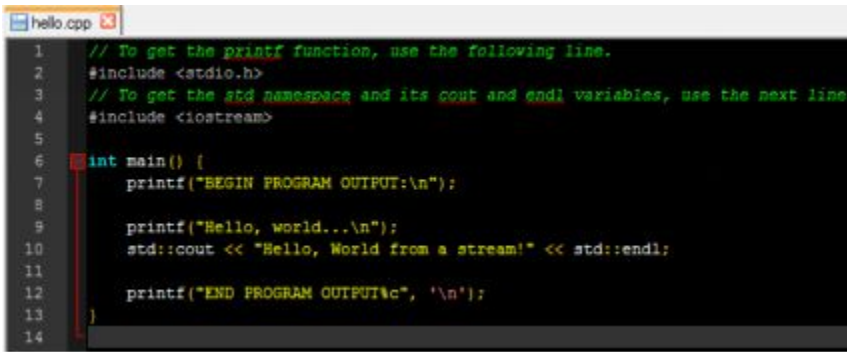
This is to familiarize you with how a terminal will look and give you a better feel for how code goes through the computer. The first command you'll do will create the file called "hello.cpp" which is where we plan to put our code. Type in **touch hello.cpp** after you've made it inside the tutorial folder. This should have created a file with the name you've specified inside of the current working directory. In case you don't recall, you can type in **pwd** in the command prompt to print your current working directory. Print here means that it'll just place the resulting text to your screen. The last image here shows it at work.





Here's where we can finally start our coding. You've successfully made a C++ source file, even though it may be a little less than full for now. To open it, you can do a couple of methods. The most obvious is double click it; however, you may have other applications which may interfere and open the file inside of them instead. To avoid this, right click the newly created file and pick the option "Open with" and click the "Choose default program..." option. Then find where Notepad++ is installed. Normally, it should be inside of C:\Program Files (x86)\Notepad++ on 32-bit systems whereas it'll be in the same place except without the (x86) on a 64-bit system.

The code given below is explained in the following paragraphs.



```
1 // To get the printf function, use the following line.
2 #include <stdio.h>
3 // To get the std namespace and its cout and endl variables, use the next line.
4 #include <iostream>
5
6 int main() {
7     printf("BEGIN PROGRAM OUTPUT:\n");
8
9     printf("Hello, world...\n");
10    std::cout << "Hello, World from a stream!" << std::endl;
11
12    printf("END PROGRAM OUTPUT%c", '\n');
13 }
14
```

The first four lines are preprocessor commands. In essence, we can look at these as library references so we can borrow common functions such as those which the green comment lines mention. This is pretty common in most C/C++ programs, and is only a small taste of what the preprocessor can do, which we'll touch on later.

The next line is a function header. In most modern languages, the compiler will try to find the function called **main** and make that the first function that runs. This doesn't mean that we have to put all of code there; in fact, that's bad practice as we'll see later when we have more than one function. In this case for now, we define a typical main function which returns an integer data type, shortened to **int**, that'll output a number we specify. Also note that it doesn't have any inputs, or arguments, inside of the parentheses. The curly brace begins the function definition explained in the following paragraph.

As for the body of the program, we have two major pieces that we need to go over. The first piece is the **printf** function which accepts a data type known as a string and an optional list of variable size of various data types to put into the main string. Breaking this down, a string is just a collection of characters. Characters in C++ are wholly represented in the American Standard Code of Information Interchange (ASCII) table available through a simple Google search. In essence, anything that appears between two quotation marks can be a string, barring special formatting using backslashes. The function will print the string exactly as it appears to the screen.

The last instance of the **printf** function includes some of the interesting formatting aforementioned. The "%c" pattern tells the function that we

want place a character in that space, whose value we get from the first variable in the list that is the optional second argument in the function. The character we want to put into the string is the ‘\n’ character. This character is the new line character which will force the following string to appear on the next line of the screen. We’re basically just telling the function to replace the “%c” with ‘\n’ in this case. Again, in a more complex system this will be extremely useful, especially for actual formatting of the string that you try to append into the main string.

For the final part of the main function body, we will cover the beauty of how C++ does standard input and output differently from the C language. In this case, we see the **std** namespace being used in the token **std::cout** and **std::endl**. A namespace is simply a way to collect related pieces of data in order to easily access them with as little overhead as possible. The specific variables being used here refer to the standard out stream and the end line string respectively.

The interesting piece is the “<<” part. This tells the program to essentially push the data that it receives from any kind of function, literal, or variable that is of the string data type. Think of it as pushing all the information down a stream toward **cout**.

The terminating curly brace will end the function definition and complete our first program. It’s a good idea to go ahead and put a new line after the last line of your code. Not only does it help for human readability, but past compilers would look for a new line in order to determine where to end parsing the end of the code. Relics of the past, it seems. We will next go over compiling to make sure that the code actually works.

Chapter 5: Compile Time: Getting an Executable

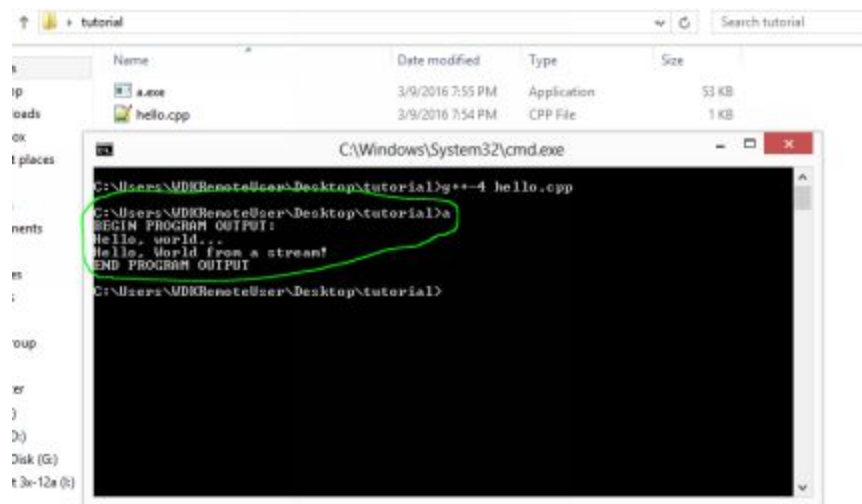
Again, compiling goes fairly deeply into a particular field of software and deserves the amount of attention it gets. Without compilers, then high-level languages couldn't really exist since they need a way to be translated into machine understandable code. For now, let's just focus on what we need to fundamentally understand to change our program to an executable file.

Hopefully you haven't closed your command prompt, because we're about to go right back to it. Make sure the command prompt is in the correct working directory as well. An easy way to do this is to run either the **dir** or **ls** commands, which print to the screen what files are in your current working directory for Windows or Linux respectively. The only file you should see is "hello.cpp" for now.

The actual command to compile varies from machine to machine. The general form is **g++** usually, with an occasional "-#" directly after it. The reason this discrepancy can happen is due to conflicting versions of the compiler being used. This can happen most likely due to various IDE's being installed on the same machine. Another useful point is the **-o** flag which requires some text after it which will be the name of the executable that gets named. The default name that the compiler will use is simply "a.exe" in Windows.

If you followed the tutorial thus far, then there shouldn't be any extra text that gets printed out after the compiler runs and creates the executable file. If there's something wrong with the code that is trying to be compile, there can be some abnormally abstract error messages that will try to tell you what's wrong with the code. The most useful piece of the information that it gives you are the line numbers and the files in which they appear. They appear in the very first part of the line the error appears which follows the following pattern usually: "<filename> : <line number> : <column number>."

Here's an image of what it should look like. The point about version discrepancies is also shown here, since the laptop used here needs the version shown. I also ran the program right after the successful compilation. The part that is encircled green is the actual output.



The screenshot shows a Windows File Explorer window with the address bar set to 'tutorial'. The file list contains two items:

Name	Date modified	Type	Size
a.exe	3/9/2016 7:55 PM	Application	53 KB
hello.cpp	3/9/2016 7:54 PM	C++ File	1 KB

Below the File Explorer is a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The command prompt shows the following text:

```
C:\Users\MDKRemoteUser\Desktop\tutorial>g++-4 hello.cpp
C:\Users\MDKRemoteUser\Desktop\tutorial>
BEGIN PROGRAM OUTPUT:
Hello, world...
Hello, World from a stream!
END PROGRAM OUTPUT
C:\Users\MDKRemoteUser\Desktop\tutorial>
```

The output text is circled in green.

The compiler understands when we need to go to another file to get some function definitions, so that way we just need to pass the compiler only the name of that one file. Now that we've gotten the green light on how our program runs, let's move on and add a little more while we're at it.

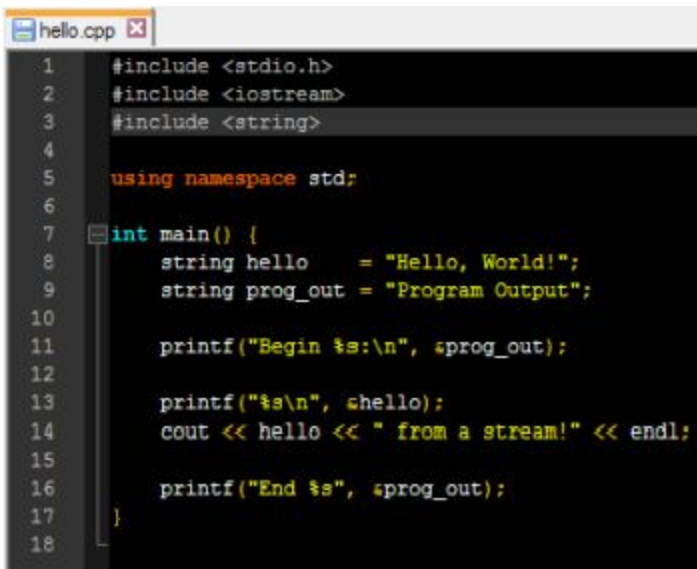
Chapter 6: Getting complex: Variables and Functions

The next few concepts are included as a bit of a set of bonus lessons in order to facilitate learning. Essentially, the next few important steps are using variables and call some functions. These are staple items when starting out programming.

Let's tackle variables first. This way we can keep everything inside of main for now. A noticeable problem with the code currently is that the strings "Hello World" and "Program Output" repeat fairly often. Let's put these two strings inside of some smaller reusable variables. Let's call them **hello** and **prog_out**.

Let me take a second to talk about variable naming. If you have three variables which stand for a three-dimensional coordinate system they could be **x**, **y**, and **z** which might be fine in this case; however, there could be multiple systems involved as in a graphics engine with the model, camera, and world coordinate systems requiring their versions of each variable. The question here is how would you name each set of variables. One way could be just to tag it with the first letter of each system that needs to be made like **x_m** for the model system's x-coordinate. The problem with that is if you came back after a long time of not dealing with the code, then you may forget what the "m" means, since it could be model or main view. The best , and sometimes longest, method to name a variable like this is to just put the entire word afterwards like **x_model**. This would make for some easy reading then.

Now that we've gotten through the bit about naming, we can put variables to work for our current code. The picture below will somewhat sum things up, but in case you need it in words I explain it afterwards as well.

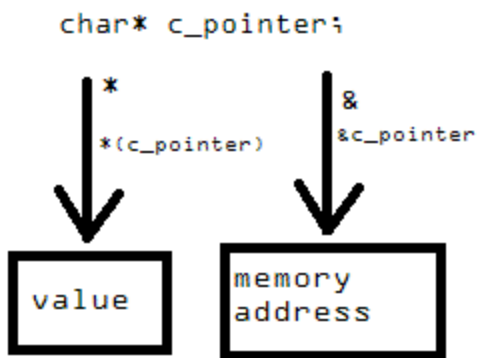


```
1  #include <stdio.h>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  int main() {
8      string hello    = "Hello, World!";
9      string prog_out = "Program Output";
10
11     printf("Begin %s:\n", &prog_out);
12
13     printf("%s\n", &hello);
14     cout << hello << " from a stream!" << endl;
15
16     printf("End %s", &prog_out);
17 }
18
```

The first thing that's noticeable is the extra include statement at the top. This includes the string data type into our program for use in our function. The next thing is the **using namespace** part that's outside of the function definition. This is along the lines of making smaller code using variable names, since now that I'm declaring that I'm using the **std** namespace all the variables involved with it are automatically brought over into our program.

The next two lines in the function that are new are the actual variable declarations. This is the typical way that variables are declared of any data type. Strings are a unique case in that they are really just a simpler way to understand the idea of a `char*` data type, which essentially says that this variable points to a character.

Pointers are a pivotal concept of C/C++ which actually shows up even in our current code. The **&** which appears before our variables that show up in our function calls basically tell the function not to use the variable itself, but to use the value that is the variable is referencing. The following picture should clear up some confusion which most fledgling coders have problems initially understanding.



The gist is that if you have a variable that is a pointer to any certain data type, then you can turn that pointer into a value using a preceding asterisk or you can use a preceding ampersand to get the address of the variable to pass it to other functions and such. That's a really simplified introduction to pointers, just so you know.

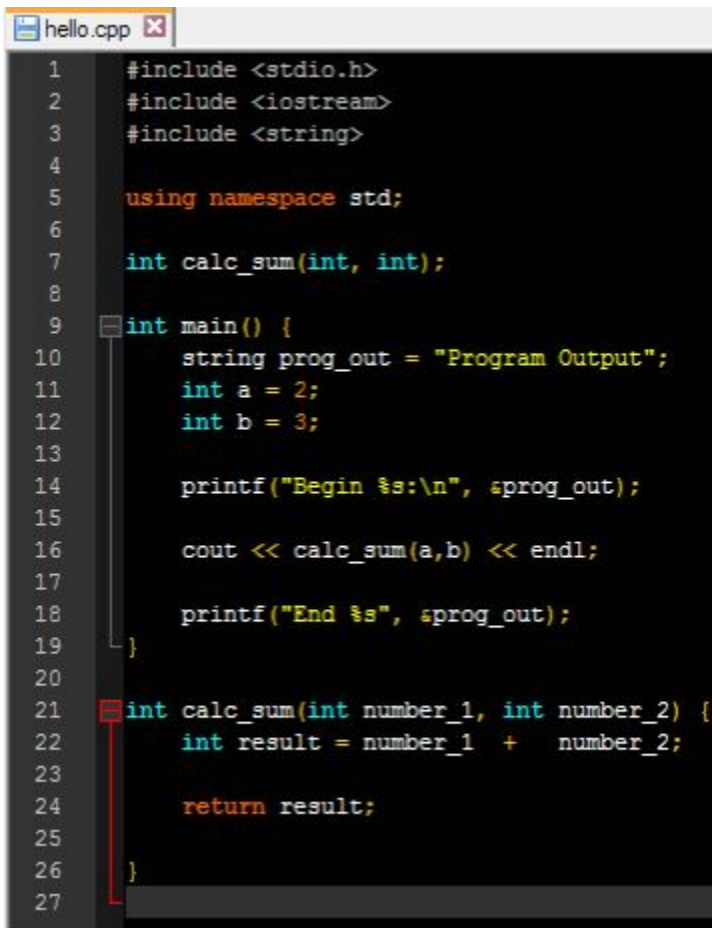
Now then, getting back to the code. There should be ampersands in front of the variable mentions in our functions because they require a pointer to the string as opposed to a value. This enables the function to pass the correct string and print out the same output as last time.

The next thing we're going to add into our current program is the idea of writing more than one function. This might not be necessary for our simple example, so let's add in some simple computation that will show the power of functions. The idea will be to add two numbers together and return their sum. For a challenge, try to add in an **if** statement to check if the value is negative and return zero if it is such. The solution will be in the conclusion.

As for the function we're going to add, let's take out the middle part of our program and replace it with the function we want to add. For namesake, let's call it **calc_sum** to keep things simple.

The picture below shows one way to add it. The idea is that we want to keep as much as the old program as we can when we add in new functionality. We'll just say that coders like to save time. The differing piece is whether we want the function to be responsible for printing its output to

the screen or not. This also affects its return value being either **void** or **int** in this case. A quick synopsis of the code is given after the screenshot.



```
1  #include <stdio.h>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  int calc_sum(int, int);
8
9  int main() {
10     string prog_out = "Program Output";
11     int a = 2;
12     int b = 3;
13
14     printf("Begin %s:\n", prog_out);
15
16     cout << calc_sum(a,b) << endl;
17
18     printf("End %s", prog_out);
19 }
20
21 int calc_sum(int number_1, int number_2) {
22     int result = number_1 + number_2;
23
24     return result;
25 }
26
27
```

Notice that there's a one-liner before the main function again. This line is called a function **prototype** which tells the compiler that we're going to define a function later on in our program that we want to use. In this case, it's called **calc_sum** and it takes two integer values as arguments.

The definition itself is fairly simple in that we just need to return the addition of two integers. It reads similarly to how we've done it in math, except that we now return it back to our caller. To make this even simpler is to directly return the operation of the adding the two values together, essentially just removing the result variable all together. There's some interesting bickering that can happen whenever returning directly is a good idea or not, but that just means it's also a good learning opportunity.

The fact that we can simply just put the function into the standard output stream is pretty nice to me. Integer conversion to a string data type is implicit in C++ which can also be kind of nice as well.

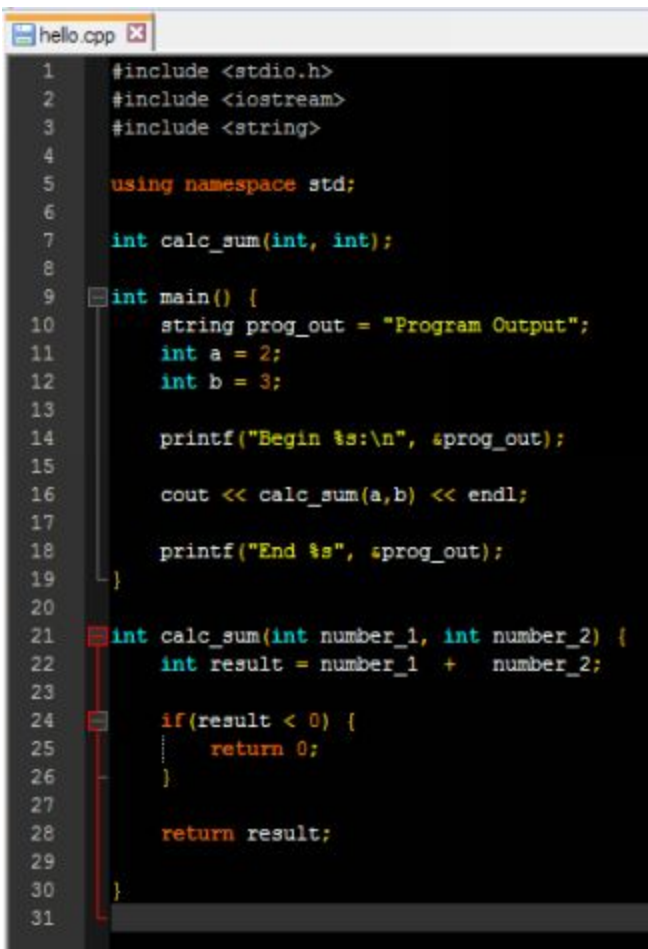
Conclusion

We've successfully made a running program. Not only that, but we've even made a good dent into some introductory concepts inside of C++ which will also help you jumpstart your programming knowledge. The best part of learning about coding is that any point of entry will help to increase your momentum into the field as a whole.

Before getting a motivational twinge, let's step back and realize that we didn't even get into virtual functions, friend classes, and so many other cool concepts. Coding is definitely my own personal hobby. The funny thing is that I'm also trying to make a career of it as well. The best way I can explain it is that just banging out some programs is a nice thing to do for a hobby, but when it comes to making a career out of it, it's an entirely different animal. The amount of information available is only expanding, so people in the software industry are having to learn constantly.

As for the solution to the **if** statement mentioned earlier, the picture on the next page should sum it up as a whole. Of course, there are plenty of other methods, such as using a **switch** statement or a **ternary operator**, that could've just as easily been implemented. Again, it's all subjective. Interpretation is a big thing of how to write software, though the compiler will tell what's right from wrong.

Again, some important factors to notice are the fact that the result variable was used to check the result. If we wanted to use the direct return method, then conditional checking becomes a little less the fun and not but a bit more difficult. Programming is always a malleable piece of equipment.



```

1  #include <stdio.h>
2  #include <iostream>
3  #include <string>
4
5  using namespace std;
6
7  int calc_sum(int, int);
8
9  int main() {
10     string prog_out = "Program Output";
11     int a = 2;
12     int b = 3;
13
14     printf("Begin %s:\n", &prog_out);
15
16     cout << calc_sum(a,b) << endl;
17
18     printf("End %s", &prog_out);
19 }
20
21 int calc_sum(int number_1, int number_2) {
22     int result = number_1 + number_2;
23
24     if(result < 0) {
25         return 0;
26     }
27
28     return result;
29 }
30
31

```

As a final few words, I can give a couple of references as to where you may want to look to continue your software journey. One of the most influential books is "The Art of Programming" by Donald Knuth. If you ever take a course in a university, then the professor will most likely mention the book or the author at some point during the class. The key pieces of software that are mentioned by this book and others include algorithms, modeling, process, and plenty of other necessary actions which don't necessarily involve coding in essence.

Speaking of modeling, it's an interesting idea that some people will literally discuss for hours on end how a system should look based on some prior analysis before even writing a line of code. Some may shout about the evils of bureaucracy when they hear about something like this, but the truth of the matter is that designing software is key to minimizing the hazards of coding in and of itself.

Even algorithm design has its hands in software. The mathematics and analytic thinking required to adequately discover just the right algorithm to solve a problem can only be analogized as the talent of an artists and the beauty of their work, as well as the fact that the latter is fairly subjective once again. This branches a bit over into the sister field of computer science, but it still affects software with more efficient algorithms just waiting to be found around the corner.

The last tidbit that I mentioned was that software has special processes that it can go through. This involves some management principles as well, but mostly affects larger teams as well which is a sufficient condition for such principles. The easiest way to understand process in a nutshell is to think about how you personally code. Okay, this may've been you're first time, but I'm fairly confident that you coded along with this tutorial. You may've also just went straight to the bottom and tried to get the last piece of code to see what you'll be learning. Either way, that shows at least some of how you program. The next question is: how will you program in the future?

The answer to that is in the code...

Thank you for reading. I hope you enjoy it. I ask you to leave your honest feedback.