C++ CRASH COURSE

The Ultimate Beginner's Course to Learning C++ Programming in Under 12 Hours

EPROGRAMY

C++
CRASH COURSE

The Ultimate Beginner's Course to Learning C++ Programming in Under 12 Hours

By Eprogramy

Table of Contents

Introduction
Welcome to Your New Programming Language

So, you've decided to learn C++ Programming? Well, congratulations and welcome to your new Programming Language! You're going to love it!

In Eprogramy we believe that the foundation of a good education is to provide the tools able to open the doors of the future. It is indisputable that the world has entered an era in which we all have everything at the fingertips. Social needs have changed. Thus, today communication has changed. Communication opens doors, so it is our challenge to learn how to communicate. In Eprogramy we want you to learn how to communicate with a new language: The programming language. This language allows us to use our PC or notebook to create.

Maybe the road is difficult and hard, but we will give the reader the tools to make learning enjoyable and fruitful. We'll show all aspects necessary to learn how to program. From the ABC´s to the solution of the most common programming problems and much, much more. Always with the help of numerous examples that will contribute to a better understanding

We know what we do and believe that you are fully capable of incorporating our teachings.

The doors of the future are here. Let's go step by step together.

Let´s get started!

Eprogramy Team

Chapter 1
C++ Programming Language

History of C++

The history of C++ is pretty much simple and interesting, in 1979 when Bjarne Stroustrup was working to complete a thesis for his PhD. If you have had a little background of working with C programming language, you will notice that it doesn't have classes; or putting it technically C cannot work with OOP. Bjarne started working on "C with classes". The main goal of doing so was so that he can add Object-oriented programming concepts to C language. C language is well praised for its simplicity and speed while execution because the code directly compiles to metal.

The first compiler introduced was Cfront that would compile the "C with classes" code into native C code. But, it was not used for a longer time and was abandoned and left, because new features and elements were not able to be added to it.

C++ came into existence when "C with classes" was renamed to C++ in 1983; fun fact is that you will soon learn that ++ operator in C means increment, to increase the value of the object. A lot of new stuff was added to the language, classes, operator overloading, inheritance and some more features that were not available in C language. Thus, making it better than the old C language. Bjarne Stroustrup released a reference to the C++ library as, "*C++ Programming Language*" and C++ was released as a commercial product; programming language.

Since then, most of the programming features and add-ons have been added to C++ to make it more robust and agile. Exceptions and other similar techniques have been added to the language to check when there are errors in the code execution. More efficiency has been added to overcome the load of the feature of OOP over C language.

Most companies and products have implemented C++ as a standard, Borland or Turbo are one of them. They have released their own versions of C++ standardized as Borland C++ or Turbo C++ etc. Not only them but most of other companies have also used C++ to write their software in order to access the efficiency and robust control over memory. Adobe has developed their Photoshop in C++ for better performance.

C++ was built exclusively as an object-oriented programming language, which doesn't mean much right now, but it will later in the guide. Object-Oriented programming allows for efficient, organized and powerful code to be created and will be seen throughout this guide.

I hope that introduction would guide you enough to understand the importance of C++ in programming languages. Now let us learn how to develop applications in C++.

What is C++?

C++ is a low-level programming language and was designed to be a platform-independent language. Although many people call C++ similar in syntax to C. There are many small variations in syntax between C++ and C. A very common difference is their OOP element, C++ supports OOP whereas C does not.

C++ does not support multiple inheritances from different parent classes. The reason for not including multiple inheritances was to avoid having messy code and making the language simpler to work with.

C++ is different from other programming languages because of its simplicity and powerful nature. That combination makes the C++ programming language great to use. Since it is a low-level programming language, it provides more support for programming the software applications that have more access to hardware resources and allows working with memory-management. It is a great feature if you want to handle all of the objects in the memory to remove or access new memory locations and to remove the objects from the memory when they are no longer required. If you are a game developer, then you would know the importance of memory-management. High-level programming languages do not support such functionalities.

Most of the frameworks that use C++ programming language allow you to use another functionality known as, “Garbage collection”. This is enabled in most frameworks, for Microsoft and Windows dependent applications that take care of garbage collection from memory allowing you to focus on programming, and not on memory-management.

Considering that C# and C++ are similar in some level, and that C# and Java are similar in some level, it is also worth noting that if you know some C# or Java, you will be able to easily pick up on how C++ works, apart from a few features only that are available only in low-level programming languages. If you are a beginner, this guide will help you get started so don’t worry.

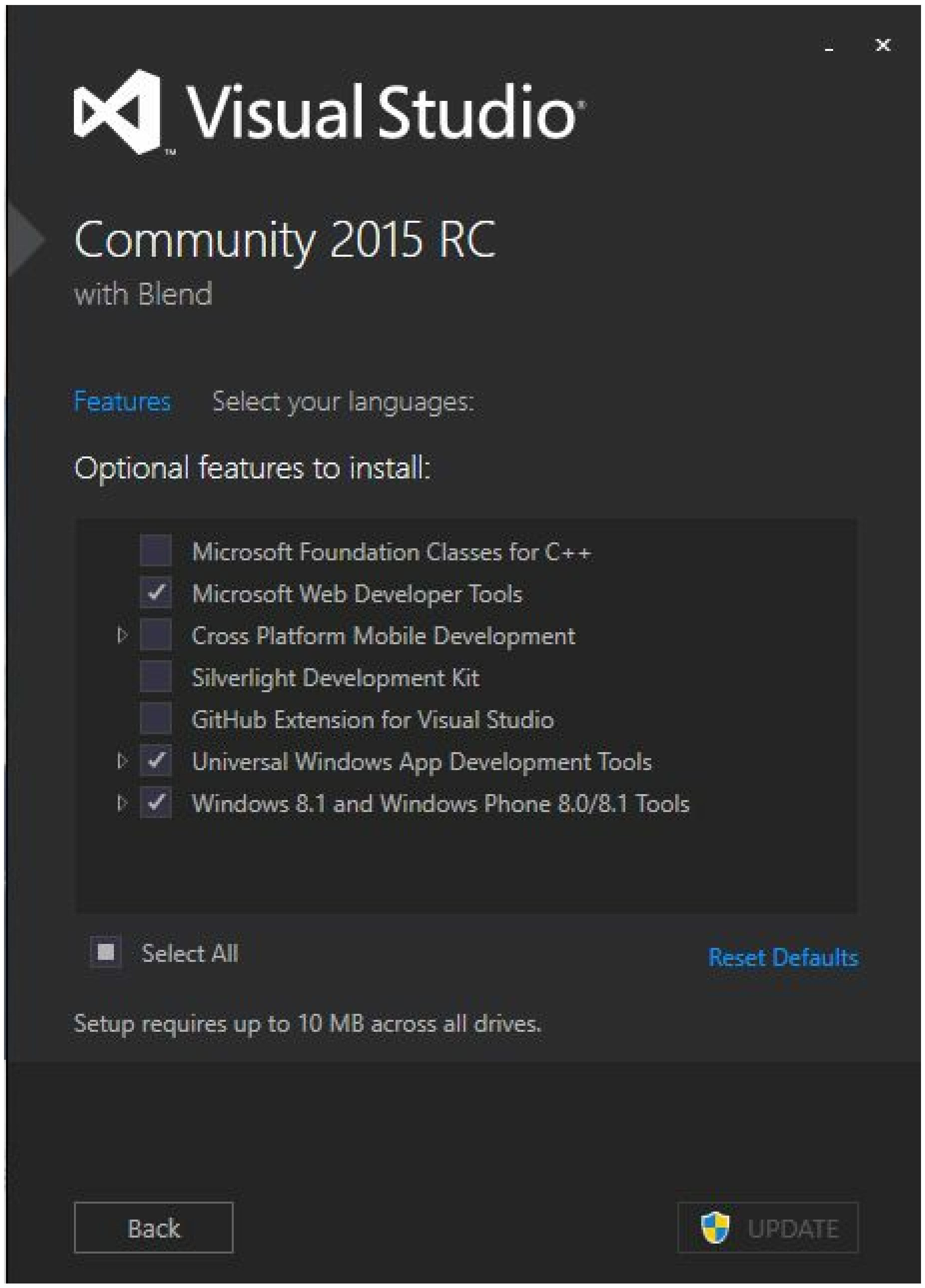
Whether you have programmed before in another language, want to learn more about C++ or are a beginner programmer starting with C++, this guide has got you covered.

Chapter 2
Setting Environment for C++

In order to set up the programming environment for C++ on to a machine, you must download the following:

- Visual Studio 2015 (Or Visual Studio 2013) Community Edition
  Good thing is that it comes free of charge and contains all the features of a professional edition of Visual Studio.

- Microsoft Foundation Classes for C++
  You can Modify your current installation and select Microsoft Foundation Classes for C++ and update the product.



Optional features to install:

- [ ] Microsoft Foundation Classes for C++
- [✓] Microsoft Web Developer Tools
- [ ] Cross Platform Mobile Development
- [ ] Silverlight Development Kit
- [ ] GitHub Extension for Visual Studio
- [✓] Universal Windows App Development Tools
- [✓] Windows 8.1 and Windows Phone 8.0/8.1 Tools

Select All

Setup requires up to 10 MB across all drives.

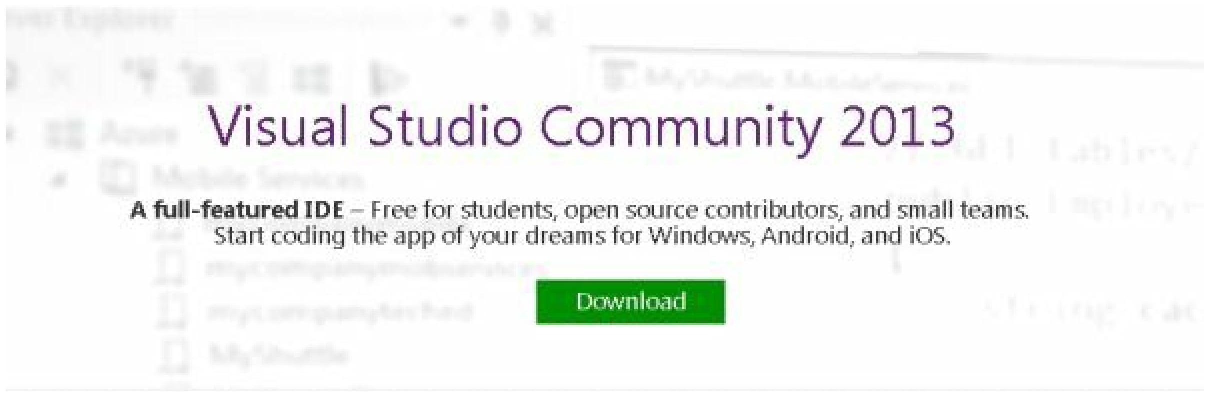
You should remove the packages that you do not need and select the ones that you need. *Microsoft Foundation Classes for C++* should be checked and any other package that you want.

The download of these tools will put you on your way to becoming a C++ programmer. An IDE (integrated development environment) is a packaged application program used by programmers because it contains necessary tools in order to process and execute code. An IDE contains a code editor, a compiler, a debugger, and a graphical user interface (GUI) that can be used to trigger multiple commands and actions, such as building the application, compiling the source code to an executable or running some different tests on the source code for debugging purposes. There are many different type of IDE's but the most commonly used one is Visual Studio for C++ as it is a standard IDE from Microsoft and provides a set of functions and features for developing Windows or Microsoft based products in C++ languages.

In this guide, it would be recommended to use Visual Studio because it is the standard to be used for C++ on Microsoft products and other cross-platform application development. In order to download Visual Studio, please use the following link:

https://www.visualstudio.com/products/visual-studio-community-vs

Once you have reached this link, you will have to find this:



Visual Studio Community 2013

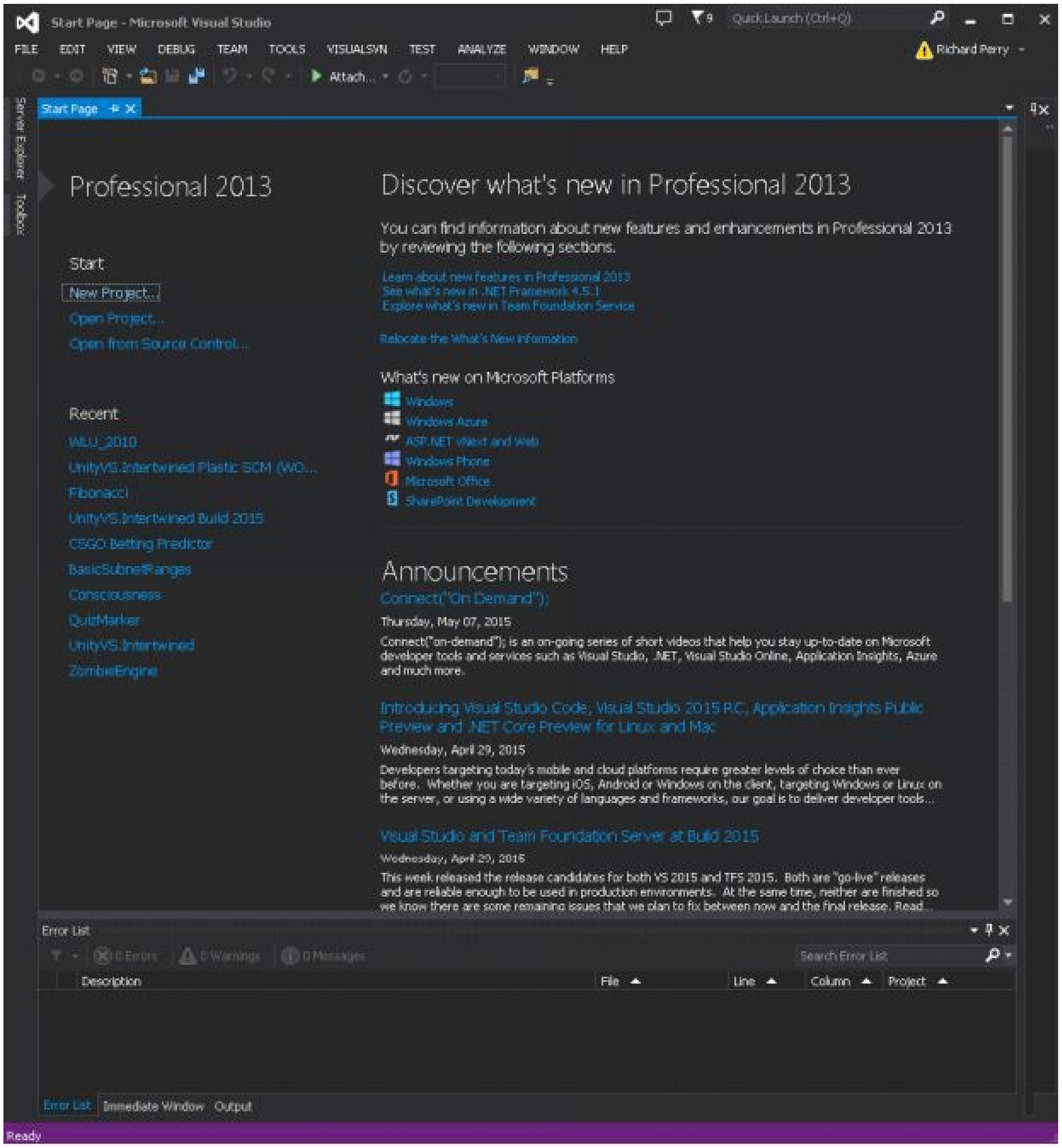
A full-featured IDE – Free for students, open source contributors, and small teams. Start coding the app of your dreams for Windows, Android, and iOS.

Note: *You can download Visual Studio Community 2013 or 2015. They both work same as per our requirements.*

Then, click download and a file will be downloaded accordingly. Once you have this file, execute it and follow the instructions on the wizard to install the Visual Studio IDE. It may try to check if you have .NET Framework installed or not, Visual Studio would take care of the frameworks and other assemblies. It would download and install the packages required to perfectly set up the environment for you to develop applications.

Once .NET Framework and the Visual Studio IDE has been installed, we can interact with our IDE in order to start coding. First of all, launch the Visual Studio IDE.

You should now be on this screen (or something similar – I am using the black theme of Visual Studio Professional 2013 IDE, later you will see the images in Light theme for Visual Studio Community 2015):



Discover what's new in Professional 2013

You can find information about new features and enhancements in Professional 2013 by reviewing the following sections.

Learn about new features in Professional 2013
See what's new in .NET Framework 4.5.1
Explore what's new in Team Foundation Service

Relocate the What's New information

What's new on Microsoft Platforms

Windows
Windows Azure
ASP.NET vNext and Web
Windows Phone
Microsoft Office
SharePoint Development

Announcements

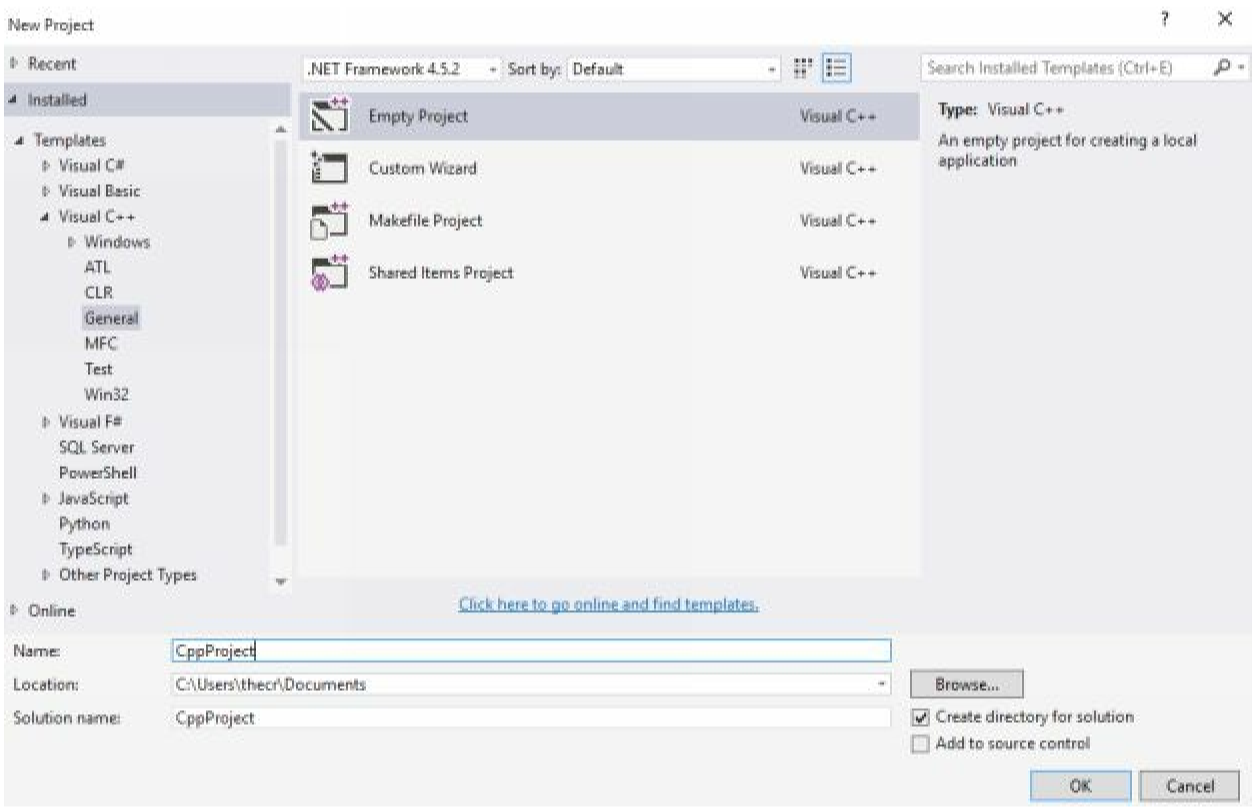
Connect("On Demand");
Thursday, May 07, 2015
Connect("on-demand"); is an on-going series of short videos that help you stay up-to-date on Microsoft developer tools and services such as Visual Studio, .NET, Visual Studio Online, Application Insights, Azure and much more.

Introducing Visual Studio Code, Visual Studio 2015 RC, Application Insights Public Preview and .NET Core Preview for Linux and Mac
Wednesday, April 29, 2015
Developers targeting today's mobile and cloud platforms require greater levels of choice than ever before. Whether you are targeting iOS, Android or Windows on the client, targeting Windows or Linux on the server, or using a wide variety of languages and frameworks, our goal is to deliver developer tools...

Visual Studio and Team Foundation Server at Build 2015
Wednesday, April 29, 2015
This week released the release candidates for both VS 2015 and TFS 2015. Both are "go-live" releases and are reliable enough to be used in production environments. At the same time, neither are finished so we know there are some remaining issues that we plan to fix between now and the final release. Read...

Instructions:

- Click File
- Click New → Project



Type: Visual C++
An empty project for creating a local application

Name: CppProject
Location: C:\Users\thecr\Documents
Solution name: CppProject
[✓] Create directory for solution
[ ] Add to source control

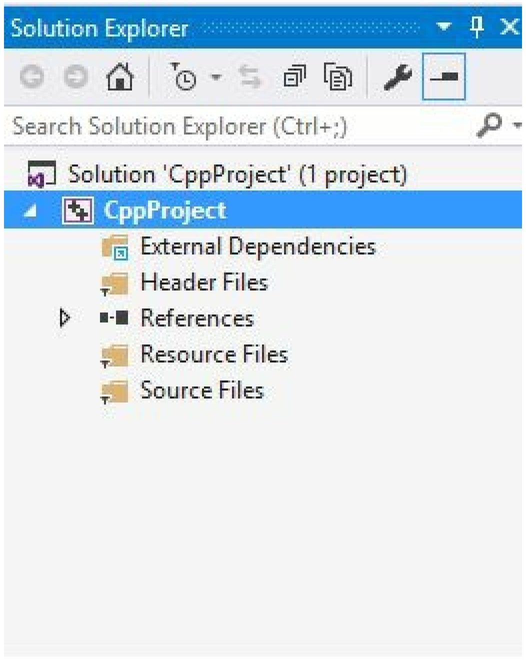
1. Select the "Empty Project" option under Templates Visual C++ (Because we need to understand how the code works, so empty project would be easy to work around with).

2. Make sure to have "Create directory for solution" for organization purposes

3. Type in a project name in the "Name:" field and set a location for your project. (Example: *HelloWorld*) I have used **CppProject**.

4. Click "Ok" to proceed

5. Now you will be presented with this page, and if you are, then you are done! Otherwise follow the steps from 1 to 4 to make sure you are doing it correctly or contact Visual Studio team for support with your product and machine.

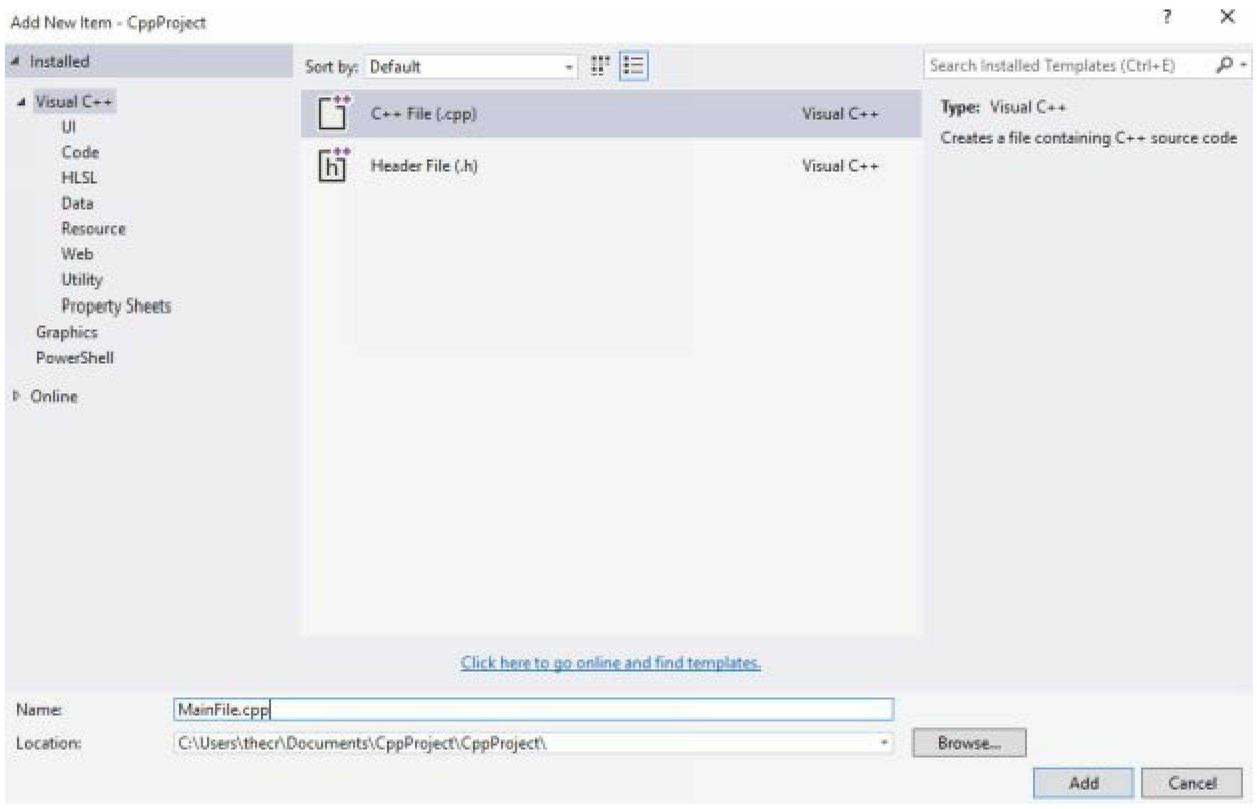


Since our project is an empty project, we won't see any header or C++ files in our project. We can add new files to the project right away.

By now, you have successfully completed the installation of C++ and setting up the environment to write your first C++ program! Next we will learn the program structure of C++ language.

Chapter 3
C++ Language Structure

Now we will learn how C++ as a programming language is structured through the sample source code file. To create a new source code file, you should Right click on project name and select Add. Inside Add menu, you should select New Item and choose C++ file (.cpp), name it what you want. I used MainFile.cpp as the name for the file.


Add New Item - CppProject
Installed
Visual C++
UI
Code
HLSL
Data
Resource
Web
Utility
Property Sheets
Graphics
PowerShell
Online
Sort by: Default
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
Search Installed Templates (Ctrl+E)
Click here to go online and find templates.
Browse...
Add
Cancel

Type: Visual C++
Creates a file containing C++ source code

Name: MainFile.cpp
Location: C:\Users\thecr\Documents\CppProject\CppProject\

This file would be empty as per now, because it is not template file (means, that it doesn't have any functionality initially).

We can add more code to it, to make it run. But first we need to understand the format (structure) of a C++ program.

```
#include <header.h>

int main() {
        // statements
        return 0;
}
```

There, the above code is pretty much simple but a little more technical description must be shared in order to make you understand how things work.

1. Include directive is used to add header files (I would talk about header files in a separate module) to your file.

2. Next comes a main function, which is the entry point for the code. When the code executes, it first of all enters this function and execute what is in it and terminates after execution.

3. Most of the Microsoft products require your C++ program to be of integer return type, and must have return 0; in it.

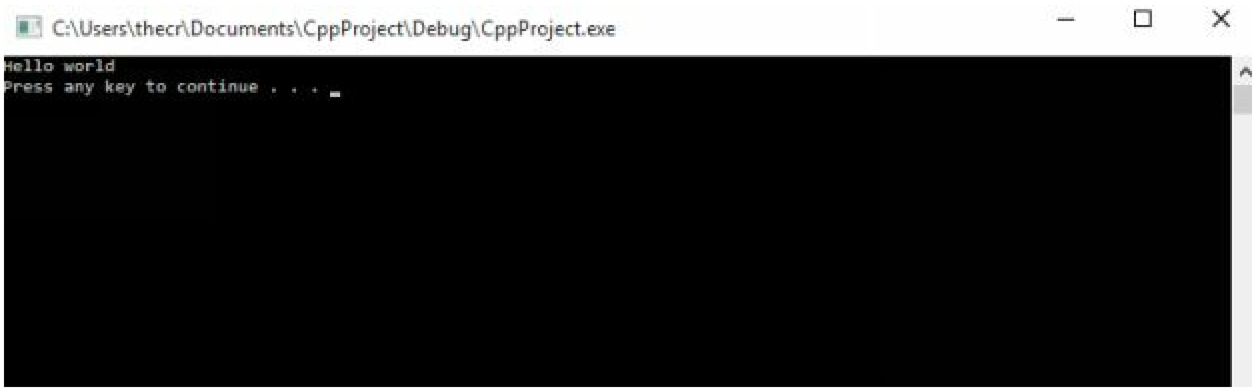
4. Double slash (//) in C++ means comments. Comments are special type of lines in a language that compiler ignores. So they are used to share description of the function, statement or something similar to other developers. Compiler doesn't read it, simply ignores the lines.

Now something that actually does something, let us write a code that says, "Hello world". (Indeed the sample project of every programmer!)

```
#include <iostream>

int main() {
        std::cout << "Hello world" << std::endl;;
        system("pause");
        return 0;
}
```

The above code sample would run and render Hello world on screen, cout is a command to send the object (string based Hello world) to the output stream. std is the namespace that is used to execute this function. Note: system("pause") is just used to stop the execution and see the output. Otherwise it is not required.



```
Hello world
Press any key to continue . . .
```

The first line #include <iostream> is using a special keyword include used along with # (*preprocessor directive*) which allows the programmer to import classes and other functions in the C++ source file which aren't present by default when creating a new program in C++. After the keyword #include - the programmer specifies a specific header file that is within the C++ library. In this case, we specified the iostream header file (that stands for *Input/output stream*), we can easily include other header files in our source code file to use their functions or other objects in our applications also.

By default, when starting a C++ program, tools and functions from the C++ library are provided at a bare minimum for any basic program that needs to be created. If the programmer for any reason wants to use more than just the basic functionalities, they must use the "#include" keyword to give themselves more tools to work with in the C++ environment.

You can also start to notice that there is a trend with a semicolon ";" after each statement. The semicolon is like a period to a statement in English ("Tim owned a dog."). When the compiler is going through the program, preparing it for its execution, it will check for semi-colons, so it knows when a specific statement ends and when a new one starts. Note that a semi-colon is not required after a preprocessor directive command. This token (";") is known as statement terminator.

The next element in the code is a function call. In C++ main function is used to execute initially, you write all of your code in this function and the code executes and then you terminate the program or restart it. You can use different structures to write your applications,

1. Sequential

2. Selection

3. Iteration

4. Function call

A function can be called from within another function; but ignore calling a function from within itself as it might throw Stack Overflow exception.

The namespace is defined by using the keyword "namespace" and then the name. Notice how there are curly braces around the code sample within it – these are used to indicate the "scope" that it has authority over. We will go over this in more detail soon. Now let us convert out code in a namespace-contained sample code.

```
#include <iostream>

namespace sample {
        int main() {
                std::cout << "Hello world" << std::endl;;
                system("pause");
                return 0;
        }
}
```

Our namespace's name is "sample" and can be used when referring to our function "main" from other source codes. You can think of a namespace as a package containing our objects and functions.

Chapter 4
Classes in C++

The next thing you'll need in the sample code is a class. C++ comes shipped with most of the data types a developer would require to create his application. But there are sometimes, when you need to wrap the data types for each user, function, event or other program. In such states, those data types do not suffice. For such objects you use classes. A class is defined similarly using a class keyword as namespace.

Let us first write the sample code for a class in C++ and see what it looks like, for example we need a class that stores age and weight of that person.

```
#include <iostream>

namespace sample {
        class person {
        public:
                int age;
                double weight;
        };
}
```

Above class can hold the details of a person object, and can store the age (of integer type) and weight (of floating-point value). Let us understand what there are in the code block above.

public - Defines the scope of the class members and whether or not other classes have access to the class member. This may not make sense now, but when learning "Inheritance and Polymorphism" - you will gain a better understanding of what this means.

class - A class can be thought of as a "type" of object.

person – This element of this important declaration is "person" which simply is a custom name that the user can redefine. You can call this anything, as all it is doing is giving the "Section" or "Class" a name. You can think of it this way:

Another thing you must be scratching your head looking at is the curly braces: "{" and "}" - all that the curly brace does is simply tells the compiler when a specific scope ends and when it starts. This could be thought of as when someone is writing an essay and they have to start their sentences with the appropriate words or indent their paragraphs.

One thing to note is that the spacing in code does not matter at all in terms of it working or not. However, it is conventional and efficient to have properly spaced code so that you and other programmers can properly read and understand it more easily. You will learn how to space as you read more sample code. Eventually, you will start to notice the trend in how conventional spacing works in programming.

```
#include <iostream>

namespace sample {
        class person {
        public:
                int age;
                double weight;
        };
}
```

As shown in the above sample code, two curly braces are communicating with each other depending on their occurrence. What-ever is in between them is the code block of name "person". The sample applies to the curly braces which are in bold and underlined to indicate that they are separate section dividers for the section within the parent section. The parent section is again, considered a class, where-as the section within the parent section, is considered a sub-section, or a sub-class or the formal term, a member.

A method is essentially a code block which also has its own special elements, similar to a class but more specific. This contains 4 elements: A scope (public/private/protected), a return type, a name, and parameters. The scope, return type and the parameters are something you will understand when learning about Methods, along with Inheritance and Polymorphism. An example of a function can be read above.

The contents of this method will also be introduced when learning about "Variables" later on in this guide.

The console is where the program's output is placed in order for the programmer to test his/her program.

A string of text must be surrounded by quotation marks in order for the program to know that it has to print out text, instead of an actual variable, like "x". The reason the user couldn't just say: "std::cout << "*The number is: x"*;" is because the program won't know if x is a variable or not because it is surrounded by quotation marks, the program would read anything within quotation marks to be a piece of text, and not actually a container holding a value. Therefore, the x must be outside of the quotation marks like, "std::cout << "The number is: " << x;"

The operator << are >> are used along with the output streams and input streams (respectively). They direct the program to either send the response to output stream, or to get a value from user using input stream.

The final element to understand in this code is system("pause") – this line simply tells the console not to close immediately and to wait for the user to press a key or exit the console implicitly. If this line wasn't there, the console would print out the output properly but only for a split second before the console would close itself down before terminating.

Commenting

The final fundamental concept to understand in a programming language, although not necessary, is commenting. It can greatly help you and other programmers around you if you ever forget how the program works

This concept is used by programmers to take the chance to explain themselves in their code, using English. When a compiler is processing the code, it will always ignore comments as it isn't actually real code. It is simply explanation to code that was written to help people understand what was programmed. In order to write a comment, you must use "//" or "/*" and "*/".

The "//" symbol is used to comment one line. For example, if you were to make a variable and explain what the variable was going to be used for, you would do the following:

```
int x = 5; // this variable is going to be used to find the total money
```

The compiler will ignore the commented line, but will process int x = 5; You can't use the "//" symbol with the following:

```
int x = 5; // this variable is going to be
used to find the total money
```

This is because the "//" symbol is only used for one line, and not two. You are able to do:

```
int x = 5;
// this variable is going to be used to find the total money
```

As long as the comment is one line, it is fine. Anything after that symbol for the line is commented.

The other technique which does the same thing as "//" except it supports multi commenting use. You must start the comment with "/*" and end it with "*/".

Example:

```
int x = 5; /* this variable is going to be
used to find the total money */
```

Up to this stage the basics have been covered and you are expected to have basic understanding of the programs and their structures.

Chapter 5
C++ Variables

What is a Variable?

A variable is essentially a storage in memory (RAM; Random access memory) that holds a certain type of data. It is named by the programmer using an identifier, and is used to identify the data it stores. A variable can usually be accessed or changed at any time. You can write information to it, take information from it, and even copy the information to store in another variable.

Variable Types

In C++, a variable has a specific type, which would determine what size it has in memory, and the layout of its memory. The C++ language defines following types of variables:

Global Variables

Instance variables are declared within a class but outside of any method, constructor or block. An instance variable is visible to all methods, constructors and blocks, within the class that it is declared in. By giving a variable the keyword “public”, it can then be accessed by sub-classes within the class. However, it is recommended to set a variable’s access modifier to “private” when possible.

Example:

```
#include <iostream>

int num = 10;

int main() {
        std::cout << num << std::endl;
        system("pause");
        return 0;
}
```

In this example, we declare the integer “num” outside of any method, and can easily access it within any method.

Static Variables:

Static Variables are declared with the keyword “static”. If a variable is static, only one instance of that variable will exist, regardless of how many instances of the object are called. Static variables are rarely used except for constant values, which are variables that are never changed or another object initialization is not intended.

Example:

```
#include <iostream>

int getnum() {
        static int num = 0;
        num++;
        return num;
}

int main() {
```

```
                std::cout << getnum() << std::endl;
                system("pause");
                return 0;
        }
```

In this example, we declare an **int** called "*num*" and set it to be static. This means that it can be accessed from within sub-classes, only one instance of it can ever exist, and the value can change. Notice that no matter how many times you call the function, num would never have old value instead value would be incremented all the times; static variable.

Local Variables

Local variables are only declared within methods, blocks or constructors. They are only created when the method, block or constructor is created, and then destroyed as soon as the method ends. You can only access local variables within the method, block, or constructor it is called, and are not visible outside of where it is called.
Example:

```
        #include <iostream>

        int main() {
                int num = 0;
                std::cout << num << std::endl;
                system("pause");
                return 0;
        }
```

In this example, a variable named "num" is called and the object is removed as soon as the main function ends. That variable only exists within the context of main, and cannot be called or accessed directly outside of that method.

Data Types

As said above, variables are used to store data. C++ has multiple built in variable data types that are used to store predefined types of data. These data type are called **primitive data types.** These data types are the most basic data types that are in the programming language. You can also create your own data types, which we will go over later. C++ has following primitive data types.

byte: The byte data type is a 8-bit signed two's complement integer. It has a default value of zero when it is declared, a maximum value of 127 and a minimum value of -128. A byte is useful to use when you want to save memory space, especially in large arrays.

Example:

byte b = 1; // has a value of one

short: The short data type is a 16-bit signed two's complement integer. Its maximum range is 32,767 and its minimum value is -32,768. A short is used to save memory, or to clarify your code.

Example:

short s = 1; // has a value of one

int: The int data type is a 32-bit signed two's complement integer. Its maximum value is 2,147,483,647 (2^{31} -1), and its minimum value is - 2,147,483,648 (-2^{31}). Int is the most common used data type for integral numbers, unless memory is a concern.

Example:

int i = 1; // has a value of one

long: The long data type is a 64-bit signed two's complement integer. Its maximum value is 9,223,372,036,854,775,807 (2^63 -1), and its minimum value is -9,223,372,036,854,775,808 (-2^63). This data type is used when a larger number is required to work with than would be possible with an int.

Example:

long l = 1L; // has a value of one with the letter L

float: The floating point data type is a double-precision 64-bit IEEE 754 floating point. The min and max range is too large to discuss here. A float is never to be used when precision is necessary, such as currency.

Example:

float f = 1200.5f; //value of one thousand two hundred, and a half

double: The double point data type is a double-precision 64-bit IEEE 754 floating point. It is often the data type of choice for decimal numbers. The min and max range is too large to discuss here. A float is never to be used when precision is necessary, such as currency.

Example:

double d = 1200.5d; // value of one thousand two hundred, and a half

bool: A bool data type represents one bit of data. It can contain two values: true or false. It is used for simple flags to track true or false conditions.

Example:

bool b = **true**; // has a value of true

char: The char data type is a single 16-bit Unicode character. Its minimum value is "*/u0000*" (or 0) and its maximum value is "*/uffff*" (or 65,535 inclusive)

Example:

char c = 'w'; // returns the letter "w"

Another commonly used data type is called string. string is not a primitive data type, but is a very commonly used data type which stores strings of text. Technically a string is an array of character type data.

Example:

string cat = "meow"; // sets value of cat to "meow"

This example gets a **string** named "cat", and sets it to the string of characters that spell out "meow".

Declaring a Variable

The declaration of a variable has three parts to it. The data type, variable name, and the stored (or initial) value. **Note** that there is a specific convention and set of rules to be used when naming variables. A variable name can be any length (less than 30) of Unicode letters and numbers, but it must start with a letter, the dollar sign “$” or an underscore “_”, or else it would return a syntax error.

```
#include <iostream>

int main() {
        int $num = 0;
        std::cout << $num << std::endl;
        system("pause");
        return 0;
}
```

It is also common naming convention to start the name with a lowercase letter, followed with each following word starting with a capital letter. For example, if the variable name was “theNum”, the first word “the” starts with a lower case, and each following word, in this case “Name”, starts with a capital letter.

Example:

```
int theNum = 123; // value of 123
```

In the example above, the data type is **int**, the name of the variable is theNum, and the value stored inside that is 123. You can also declare a variable without storing a value in it.

Example:

```
int num; // no value
```

You can do this if you choose to later declare it.

Using a Variable

After a variable is declared, you can then read its value, or change it. After the variable was initially declared, you can reference it only by its name; you only need to declare its data type when you are declaring the variable.

Example:

```
name = 2; // sets the int "name" to a value of 2
```

The example above sets the value of name to 2. Notice how I never restated the data type.

Example:

```
std::cout << name; // prints the value of name to the console
```

This example reads the value of “name” and writes it to the console.

Variables can also be added together:

Example:

```
int a; // no value
int b = 1; // value of one
int c = 2; // value of two
a = b + c; // sets a to the value of b + c
```

In the example above, we set the value of a, to equal the value of b and c added together. The addition sign is known as an operator, which we will go over in the following section.

It is also possible to combine values of variables that are different data types to a certain extent.

Example:

```
int a; // no value
float b = 1; // value of one
int c = 2; // value of two

a = (int) (b + c); // sets the int "name" to a value of b + c
```

This example is just like the one before it, except for that we changed the data type of b from **int**, to **float**. The only different when adding them together, is that we had to add something called a “cast” to the equation. What a cast does is it simply lets the compiler know that the value of (b + c) should be of the datatype int. **Note** that for this example, if the value of b + c were to equal a decimal number (for example 3.2), the value of a would not be 3.2 but rather 3, because **int** does not support decimals.

Assignment

Using what we learned about variables, we can now create a simple calculator to add numbers together for us. The first thing we will want to do is declare 3 variables: one to store the value, one to represent the first number we want to add, and one to represent the second number we want to add. We will declare these variables as **double** so that we can add decimal numbers:

```
                  double a = 0; // stores value of addition
    double b = 3.55; // first number to add
    double c = 52.6; // second number to add
```

Next, we will simply set the value of a, to equal the value of b and c combined, and then print out the value of a.

```
                  a = b + c;

    std::cout << a;
```

If you run the program now, it should print out 56.15. You now have just created a very simple calculator. I highly encourage you to play around with this, and test things for yourself. Change the data type of the variables, add more numbers together, and experiment to understand how things work.

Pointer-types

In C++ a programmer can use another type of data; objects. That is known as pointers (or pointer-types). Pointers are special types of variables as they consume the same type and operations. But, they are used to store the addresses of objects in the memory. You can think of a pointer-type to be just a directory. Actual object exists in the memory, only its address is stored in the variable. That is why it is 32-bit in size.

You can define your own pointers to any variable type, int, char, float etc. The pointer-type variable doesn't exist itself, instead is just a reference to the other variable in the memory. Have a look at the following code,

```
    #include <iostream>

    int main() {
            int a;
            int *prt;

            system("pause");
            return 0;
    }
```

In the above code sample, I have created two variables. One is of integer type, other one is a pointer to integer-type variable. There is a difference. What? The difference is that "a" is an actual object that exists in the memory and is similar to what you have been taught. Whereas ptr is not an actual object, but a reference to another integer type object in the memory. You can set the reference of a variable as its value using ampersand (&) operator that returns the address of a variable.

```
    #include <iostream>

    int main() {
            int a;
            int *prt;

            a = 5;
            prt = &a;

            std::cout << *prt << std::endl;

            system("pause");
            return 0;
    }
```

In the above code we have created two objects, one as an object, and other as an object to hold the address of other variable. We have then initialized a variable with a value of 5 and then passed the reference of that variable to the pointer-type variable. Thus, it now holds the handle to the variable in the memory.

Further we would use the pointer resolver to get the value at that location; remember pointer holds the address, not the value of the variable. That is why we would append an asterisk before it to resolve the value at that location. Just as a side note, pointers can also point to other pointers and this is called Pointer-to-pointer type. They are doubled in process, you have to use double asterisk to get the actual value.

Chapter 6
C++ Operators

Just like in Math class, you learned about addition, subtraction, multiplication, division, etc. These are all arithmetic operators that are used within programming to intake numbers, process them, and calculate them accordingly. Let’s go over these operators in programming, as they are one of the most important things to understand and also one of the easiest to grasp.

The Arithmetic Operators

Addition

```
5+5 = 10
int x = 5;
int y = 5;
int sum = 0;
sum = x + y;
```

In the example above, a variable of sum is taking the addition of two variables (x and y) and adding them together to produce a value of 10.

Subtraction

```
10-5 = 5
int x = 10;
int y = 5;
int total = 0;
total = x - y;
```

In the example above, a variable of total is taking the subtraction of two variables (x and y) and subtracting them together to produce a value of 5.

Multiplication

```
5*4 = 20
int x = 5;
int y = 4;
int total = 0;
total = x * y;
```

In the example above, a variable of total is taking the multiplication of two variables (x and y) and multiplying them together to produce a value of 20.

Division

```
20/5 = 4
int x = 20;
int y = 5;
int total = 0;
total= x / y;
```

In the example above, a variable of total is taking the division of two variables (x and y) and dividing them together to produce a value of 20.

Modulus

```
7 % 2 = 1
int x = 7;
int y = 2;
int total = 0;
total = x % y;
```

In the example above, a variable of total is taking the remainder of two variables (x and y) and by finding how many times 2 multiplies in to 7 evenly before it can’t. After that processes, the remainder is the output. For example:

> How many times does 2 go into 7 evenly?
> 3 times.
> 2 * 3 = 6
> 7 - 6 = 1
>
> Therefore, 7 modules 2 is equal to 1 because 1 is the remainder of the division.

Post-Incrementation

```
5 + 1 = 6
int x = 5;
x++;
int y = 0;
y = x++;
```

In the example above, a variable of x is being incremented by the value of 1. Therefore, the value of x must now be 6 because x + 1 = 6, since x = 5. If you were to print out the value of x, the output would be 6. You can do x ++ again and the value would then become 7. The value of y on the other hand is still 6 because it is post incrementing meaning that the value of x has changed by an increment of 1 (to 7). However, the value of y has yet to be changed until it is called again (y = x ++) which in the case would be y = 7.

Pre-Incrementation

```
5 + 1 = 6
int x = 5;
++x;
int y = 0;
y = ++x;
```

In the example above, a variable of x is being incremented just like it was when it was post-incrementing as a variable by itself. The value of y on the other hand changes directly to 6 because it is pre-incrementing. What this means is that the value of x has changed by an increment of 1, and the value of y changes, meaning that y = 7.

Post-Decrementing

```
5 - 1 = 4
int x = 5;
x--;
int y = 0;
y = x--;
```

In the example above, a variable of x is being decremented by the value of 1. Therefore, the value of x must now be 4 because x - 1 = 4, since x used to be equal to 5. If you were to print out the value of x, the output would be 4. You can do x -- again and the value would then become 3. The value of y on the other hand is still 4 because it is post decrementing meaning that the value of x has changed by a decrement of 1. The value of y still has yet to be changed until it is called again (y = x --) which in the case would be y = 3.

Pre-Decrementing

```
5 - 1 = 4
int x = 5;
--x;
int y = 0;
y = --x;
```

In the example above, a variable of x is being decremented by the value of 1. Therefore, the value of x must now be 4 because x - 1 = 4 since x was equal to 5. If you were to print out the value of x, the output would be 4. You can do -- x again and the value would then become 3. The value of y on the other hand is 3 because it is pre decrementing. This means that the value of x has changed by a decrement of 1, and the value of y has changed.

The Assignment Operators

The assignment operators are operators that are used when assigning variables values, the most commonly used operator being (=). Here are a list of examples:

Equal Sign: =

```
  int x = 5;
```

In this example, if you were to print out x, the value would be 5 because you assigned the variable x equal to 5.

Add-Equal Sign: +=

```
int x = 5;
  x += 5;
```

In this example, if you were to print out x, the value would be 10 because you are adding the value of 5 onto the value of x. This statement is the same thing as saying x = x + 5 → x += 5.

Subtract-Equal Sign: -=

```
int x = 5;
  x -= 5;
```

In this example, if you were to print out x, the value would be 0 because you are subtracting the value of 5 from the value of x. This statement is the same thing as saying x = x - 5 → x -= 5.

Multiplication-Equal Sign: *=

```
int x = 5;
  x *= 5;
```

In this example, if you were to print out x, the value would be 25 because you are multiplying the value of 5 onto the value of x. This statement is the same thing as saying x = x * 5 → x *= 5.

Division-Equal Sign: /=

```
int x = 5;
  x /= 5;
```

In this example, if you were to print out x, the value would be 1 because you are dividing the value of 5 onto the value of x. This statement is the same thing as saying x = x / 5 → x /= 5.

Modules-Equal Sign: -=

```
int x = 5;
  x %= 5;
```

In this example, if you were to print out x, the value would be 0 because you are finding the remainder in (5/5). This statement is the same thing as saying x = x % 5 → x %= 5.

Operator Overloading

Operator overloading is a technique used in C++ that allows developers to create their own types and functions to be executed when an operator is called on a type. Their use is very much popular in custom classes.

For example, you would not want your food to be added in a built-in way, instead you would try to *pack it up*. Thus using operator overloading, you can easily run the code to pack up the food objects otherwise there might be some spillage.

```
#include <iostream>


class Pack {
public:
        int numberOfFoodItems;
};

class Food {
private:
        Pack p;

public:
        void operator +(Food f) {
                // Food was added, increment the pack
                p.numberOfFoodItems++;
        }

        void openPack() {
                        std::cout << "Pack has " << p.numberOfFoodItems << " items..." <<
std::endl;
        }

        Food() {
                p.numberOfFoodItems = 1;
        }
};

int main() {
        Food f = Food::Food();
        Food g = Food::Food();

        f + g;

        f.openPack();
```

```
                    system("pause");
                    return 0;
          }
```
Now this program would provide with an output of 2; why? Because initially there were only 1 food item, after adding, we now have 2 food items ready in our pack. Similarly, you can override other operators and use them as you want to use them in your application just to reflect the intuitive response for the user.

Chapter 7
User Input

The std::cin is the command that will allow us to input information from the console, so the program is able to process it and do something with it.

How it works:
Input → Process → Output

The first thing you want to do is use the iostream header file and the std namespace by doing the following:

```
#include <iostream>
```

You can append the std namespace before cin command (cin stands for input stream). Then we must set the skeleton of our program. The next line you have to code, assuming your class and main method are already set, is to set a variable equal to the user input. This is going to wait for the user to type something and once you have, you must press enter for it to be stored in the variable accordingly.

```
#include <iostream>

int main() {
        int $num = 0;
        std::cin >> $num;
        std::cout << $num << std::endl;
        system("pause");
        return 0;
}
```

Now you must decide, what would you like the user to input? A string? An integer? Well in our situation, we want the user to input a number, so we set the variable to a int and use it in std::cin command.

Chapter 8
Strings in C++

Strings are quite commonly used in C++ programming. A string is a sequence of characters, or a line of text, or putting it technically, it is an array of char type objects. A string in C++ is like an object, meaning that it has properties that can be used, and different functions or methods that can be executed on it.

The way to declare a string variable is by doing the following:

char name[] = “Your name”;

Complete code example is following:

```
#include <iostream>

int main() {
        char name[] = "Name";
        std::cout << name << std::endl;
        system("pause");
        return 0;
}
```

There are two ways to declare a string, one way is to use char array, other way is to use a type string; which is indeed a class object. It requires you to use a header file named string.h.

```
#include <iostream>
#include <string>

int main() {
        std::string name = "Name";
        std::cout << name << std::endl;
        system("pause");
        return 0;
}
```

Now let’s go over the fundamental properties of a string object that can be used to gain extra information over the piece of text or string.

String Length

A way to receive information on the amount of characters in a String is by doing the following:

```
#include <iostream>
#include <string>

int main() {
        std::string name = "Name";
        std::cout << name.length() << std::endl;
        system("pause");
        return 0;
}
```

The length function in a String object returns a value for the amount of characters within the String object and assigns the value to the integer amount. The output in this sample code would be 4 since “test” is 4 characters long.

Concatenating Strings

Concatenating is the idea of joining two strings together. Here is an example:

```
#include <iostream>
#include <string>

int main() {
        std::string name = "Name";
        name = "Hello" + name;
        std::cout << name.length() << std::endl;
        system("pause");
        return 0;
}
```

As shown in the example above, you are able to use the “+” operator to join two or more strings together.

Here are various examples of utilizing the String object:

Examples

Example – string[int index]

```
#include <iostream>
#include <string>

int main() {
        std::string name = "Name";
        name = "Hello" + name;
        std::cout << name[0] << std::endl;
        system("pause");
        return 0;
}
```

The value of “name” contains the letter “HelloName”. Console would show only H because the inputted index of the string is 0 which is the first letter of the string. The way the computer reads the string, is from 0 to the length of the string, not 1. To the computer, a string is a list of characters, so indicating a square bracket with an index is like referring to a specific index of a character in a list of characters when really it’s a string variable.

Example – Equality check

```
#include <iostream>
#include <string>

int main() {
        std::string name = "Name";

        bool equals = (name == "Name");
        std::cout << equals << std::endl;
        system("pause");
        return 0;
}
```

The value of equals is going to equal true in this case because the == operator returns a bool value (true or false) and the text does indeed equal to Name. Later on, you will learn about “if” statements, which can allow you to check whether or not something equals to something else. When comparing most other data types you will use “==”. You will learn more about this later on.

Example - Substring(int beginIndex)

```
#include <iostream>
#include <string>
```

```
int main() {
        std::string name = "Name";

        std::cout << name.substr(0, 2) << std::endl;
        system("pause");
        return 0;
}
```

Since the position of the letter “N” is 0, if you say substr (0, 2) - it will essentially divide that text starting from that position. Therefore, the “name” variable ends up containing the value of “Na”.

Chapter 9
Boolean Logic

C++ provides us a vast set of operators to manipulate variables with. In the following section we are going to go over Relational Operators and Logical Operators. Before we can go to understand how to use these operators, we need to know where we can use them. There are many different **statements** that can be used in order to perform and test different types of logic. We will go over some more complex statements in later sections.

If Statements

For the following examples, I will use an **if** statement. An **if** statement, simply put, checks to see **IF** something is true, or false. To create an **if** statement, you simply write the word **if** followed by two rounded brackets. This will contain the logic to check **IF** something is true or not. **if** statements can also include **else if** statements, and **else** statements. An **else if** statement will run if the **if** statement returns false. However, it also contains its own parameter that must return true. An **else** statement also is called if the **if** statement returns false, the difference between an else statement, and an **else if** statement, is that no parameters need to be true for an else statement to run.

Example:

```
#include <iostream>

int main() {
        int a = 10;
        int b = 20;

        if (a == b) {
                // Equal
        }
        else if (a > b) {
                // Greater than
        }
        else if (a < b) {
                // Smaller than
        }
        else {
                // Otherwise..
        }

        system("pause");
        return 0;
}
```

Note: The output code is not implemented in the source code, but is explained in the following section.

The operands used in this example ("==" and ">") are explained below. In the following code, we first check to see if "a" is equal to "b". If that is true, we simple print "a is equal to b" and ignore the rest of the code. If "a" does not equal "b" (this would be ignore because other two blocks would fulfill the requirement of this not equal to block), then it goes down and calls the else if. The **else** is due to the first **if** returning false. The **if** part of **else if** is checking its own parameters, which happen to be whether "a" is greater than "b". If "a" is greater than b, then we would have printed "a is greater than b". However, since "a" is less than "b", we go further down and simply call **else**. The **else** runs if all the above parameters return false. It does not require its own parameter to be true. **Note** That each **if** statement can have an infinite amount of **else if** statements which follow it, but only one **else** statement. Also **Note** that you cannot have code between an **if** statement and an **else** statement, because the code would not be able to find the **else**. As well, an **else** statement requires there to be an **if** statement before it, since for an **else** statement to be called, an **if** statement HAS to equal false immediately before it. It is also important to know that it is possible to have an **if** statement INSIDE of another **if** statement. This is called a **NESTED if** statement

Example:

```
#include <iostream>

int main() {
        int a = 10;
        int b = 20;

        if (a == b) {
                // Equal
        }
        else {
                // Not equal
                if (a > b) {
                        // Greater than
                }
        }

        system("pause");
        return 0;
}
```

In this example, we first check to see if the value of "a" is equal to the value of "b". If it is, we then check to see if the value of "a" is less than the value of "b". If this statement also returns true, we can print to the console "a is less than b and it is equal to 20". This can work with as many **if** statements as you would like. This can be useful if completely necessary to check the parameters separately, or if you want to add additional logic between each check.
You will understand fully the way an **if** statement works by the end of this section.

The Relational Operators

C++ supports different types of relational operators, which can be used to compare the value of variables. They are the following:

== This operator checks to see if the value of two integers are equal to each other. If they are the equal, the operator returns true. If they are not, it returns false.

Example:

```
int a = 10;
int b = 20;

if (a == b) {
        // Equal
}
```

This code checks to see **IF** the value of "a" is equal to the value of "b". Since the value of "a" is equal to the value of "b", the value within the brackets will be true. This ends up causing the above code to print out the statement "they are the same"; if the output code is added which is not added in our current code.

!= This operator checks to see if the value of two things DO NOT equal the same thing. If two things DO NOT equal the same thing, then it will return true. If they DO equal the same thing, it will return false.

Example:

```
int a = 10;
int b = 20;

if (a != b) {
        // Not Equal
}
```

This code will check if "a" does not equal "b". Since "a" does not equal to "b", the program will print "they are not the same" in the console; if that has been implemented in the code.

> This operator Checks to see if something is **Greater than** something else. If the value of the variable in front of it is greater than the value of the variable after it, it will return true. If the value of the variable in front of it is less than the value of the variable after it, it will return false.

Example:

```
int a = 10;
int b = 20;
```

```
if (a > b) {
        // Not Equal
}
```

This example checks to see if "a" is larger than "b". Since "a" is not larger than "b", this if statement will return false. Instead, it will print "a is not larger than b", because the code passes the failed **if** statement and calls the **else** statement.

< This operator checks to see if something is less than something else. If the value of the variable before the operator is less than the value of the operator after the variable, then the code will return true; else, it will return false.

Example:

```
int a = 10;
int b = 20;

if (a < b) {
        // Not Equal
}
```

This example is just like the one before it, however, the operator changed from greater than, to less than. Therefore the **if** statement will return true this time. Since the value of "a" is less than the value of "b", and the program will print to the console "a is not larger than b".

<= This operator checks to see if something is greater than **Or Equal** to something else. If the value of the variable before it is greater than **Or Equal** to the variable after it, then it will return true. If the variable after it is greater but not equal to the variable before it, it will return false.

Example:

```
int a = 10;
int b = 20;

if (a <= b) {
        // Equal or less than than
}
```

In this example, "a" and "b" both have a value of 10 and 20 respectively. Although "a" is not greater than "b", it is also not EQUAL to "b", therefore the statement returns true, and the code ends up printing "a is less than or equal to b".

>= This operator checks to see if something is less than **Or Equal** to something else. If the value of the variable before it is less than **Or Equal** to the variable after it, then it will return true. If the variable after it is less but not equal to the variable before it, it will return false.

Example:

```
int a = 10;
int b = 20;

if (a >= b) {
        // Equal or greater than
}
```

This example is identical to the one before it except the operator was changed from >= to <=. Although the operator has changed, the result is the same. Since “a” is not equal to “b” and also not greater than b, this code will return false printing “a is not larger than b” to the console.

The Logical Operators

C++ supports 3 Logical Operators that can be used in your logic, which will often be used in conjunction with Relational Operators.

&& This is known as the logical AND operator. If the operands before and after this operator both return true, then the condition returns true. If both of the operands are false, or one operand is false, the condition will return false. **BOTH** operands MUST be true for the condition to return true.

Example:

```
int a = 10;
int b = 20;

if (a == b && a == 10) {
        // Condition met
}
```

In this Example, we check to see if “a” is equal to 10, AND if “a” is equal to “b”. These two conditions will be referred to as operands. Since “a” is equal to 10, but not equal to “b”, the statement returns false, and we print to the console “A is not equal to 10 or b”.

This means that when we run this code, the **if** statement will return false because although “a” is equal to 10, it is not equal to “b”. Therefore we print out “a is not equal to 10 or a is not equal to b”. Take note that the order does not matter. If the first operand was false instead of the second, we would have the same result.

|| This Operator is known as the logical OR operator. If the first operand **OR** the second operand is true, the statement will return true. This means that if the first operand returns false and the second is true, the statement will still return true, and vice versa. The statement also will return true if both the operands are true. Essentially, when using an OR operator, at least one operand must return true.

Example:

```
int a = 10;
int b = 20;

if (a == b || a == 10) {
        // Condition met
}
```

In this example. We check to see if “a” is equal to 10, OR if “a” is equal to “b”. In this case, “a” is equal to 10 but value of “a” is not equal to “b”. Since only one of these operands need to be true, the statement as a whole will return as true. Therefore the program will print “a is equal to 10 or b” in the console.

! This operator is known as the logical NOT operator. It is used to reverse the logical state of its operand. This operand can be used with any other operand to reverse its output. If an operand was returning true, after applying the logical NOT operator, the operand will return false.

Example:

```
int a = 10;
int b = 20;

if (!(a == b)) {
        // Condition met
}
```

In this example, we check to see if the value of “a” is equal to “b”. Since “a” is not equal “b”, we can print out to the console “a is not equal to b”; because the NOT operator would alter the value of the bool expression.

Combining Operators

In C++, it is possible to use as many operators as you require per statement.

Example:

```
int a = 10;
int b = 20;

if (((a == 20 && a > b) || (a != 20 && a < b))) {
        // Condition met
}
```

With this example, we can see how we can achieve much more complicated statements. Here, we apply an OR operator with two operands, however, each operand also incorporates an AND operator. Therefore, for this statement to return true, “a” must be equal to 20 and be greater than “b” OR “a” must not be equal to 20 AND “a” must be less than “b”. Note that we use brackets to make the code a lot easier to understand from a reader’s perspective, and to prevent the code from misreading our logic.

Example 2:

```
int a = 10;
int b = 20;

if (a == 20 && (a > b || a != 20) && a < b) {
        // Condition met
}
```

In the above example, we have significantly changed the logic of the program, by only switching the position of a bracket. Notice how now the brackets surround the two inner operands, instead of two pairs of brackets surrounding each pair of outer operands. For this statement to return true now, 3 different conditions must return true. Variable “a” must be equal to 20, AND, either “a” must be greater than “b” OR it must not be equal to 20; AND “a” must be less than “b”. With this new logic, the first operand must be true, on top of EITHER the second or third having to return true. Also on top of that, “a” must be less than “b”. This statement will return false. It will fail the first condition, because “a” does not equal 20. It will pass the second condition, because it is greater than “b”, and it will fail the last condition because “a” is not less than “b”. Since this would have required three correct conditions, and only had one, the statement returns false and prints “wrong”.

Assignment

We are now going to go through another assignment to make sure we understand everything about operators. We have 3 friends who want to know how much money they have compared to the others. We will make a program that will output who is richer than who. We will have three integers named bob, john, and tom. We will give each one of them a different value, and the program must output who is richer. It must also tell us if someone has the same amount of money as someone else.

For example:

If bob = 20, tom = 10, and john = 5, the output must be:
"bob is richer than tom, who is richer than john"

But if bob = 20, tom = 20, and john = 5, the output must be:
"bob is just as rich as tom, both are richer than john"

The program needs to work for each possible outcome. I highly encourage you to try this program by yourself and struggle through it as much as you can before you look at the answer. A big part of programming is problem solving and understanding code, so I recommend you try to figure it out by yourself first as it would help you in getting to know which parts of programming do you need to work around on and need to still learn more about.

Answer and Explanation

Note that the program will be explained through comments in the code.

```
#include <iostream>

int main() {
        int tom = 10;
        int bob = 20;
        int john = 30;

        /* We first check if everyone has the same amount of money
        *  note how I never check if tom is equal to john, because if
        *  tom is equal to bob and bob is equal to john, then obviously
        *  tom is equal to john
        */
        if (tom == bob && bob == john) {

                std::cout << "Everyone has the same amount of money";
        }

        /*
        *  If the first statement returns false
        *  I next check the unique case where tom and bob have the same amount
        *  but john does not
        */
        else if (tom == bob && bob != john) {

                /*
                * I now check if bob is greater or less than john
                * and then output the correct answer
                */

                if (bob > john) {
                        std::cout << "tom is just as rich as bob, both are richer than john";
                }
                else if (bob < john) {
                        std::cout << "tom is just as rich as bob, both are more poor than john";
                }
        }
        /*
        * I repeat the same process as the previous statement but with different people
        */
        else if (tom == john && john != bob) {
```

```
        if (john > bob) {
                std::cout << "tom is just as rich as john, both are richer than bob";
        }
        else if (john < bob) {
                std::cout << "tom is just as rich as john, both are more poor than bob";
        }

}
/*
* The last possible combination of names,
* same check as previous statement
*/
else if (john == bob && bob != tom) {

        if (bob > tom) {
                std::cout << "bob is just as rich as john, both are richer than tom";
        }
        else if (bob < tom) {
                std::cout << "bob is just as rich as john, both are more poor than tom";
        }
}

/*
* Now I check the last possible combinations
* where each person has different amounts of money
* the next 6 statements cover each possible outcome
*/

else if (tom > bob && bob > john) {

        std::cout << "tom is richest, followed by bob, followed by john";
}
else if (tom > john && john > bob) {

        std::cout << "tom is richest, followed by john, followed by bob";
}
else if (bob > tom && tom > john) {

        std::cout << "bob is richest, followed by tom, followed by john";
}
else if (bob > john && john > tom) {

        std::cout << "bob is richest, followed by john, followed by tom";
}
else if (john > bob && bob > tom) {
```

```
                std::cout << "john is richest, followed by bob, followed by tom";
        }
        else if (john > tom && tom > bob) {

                std::cout << "john is richest, followed by tom, followed by bob";
        }
        system("pause");
        return 0;
}
```

When writing code it is important to comment all of your logic so that someone who has not written it can easily understand and edit it. I recommend that when you write code you comment it as much as you feel is required. I also encourage you to try to find a better way to solve this problem. Maybe there is a way you can write this in a quarter of the length; with programming, there is never only one answer.

Apart from code, other naming conventions such as identifiers of functions, variables also help you out a lot in allowing developers to understand your code. Developers read the code as if they were reading a theory, so a good theory is always good formatted and well designed. Your software source code matters a lot. If you write a good software code, then chances are that your readers would be happy to read the API and understand what is being done.

Chapter 10
Loops and Arrays

Loops

When programming, you might run into a situation where you will need to loop through a large amount of numbers. If we used what we learned so far to loop through 100 numbers, we would need 100 if statements. I think you would agree that would get really messy and frustrating. This is why C++ supports multiple type of loops. C++ has three types of loops that you can use, that all loop through things in slightly different ways. Below is a quick explanation of each one of them.

While Loop

A **while** loop is a control structure that will allow you to repeat a task as many times are you program it to. The syntax for a while loop is as follows:

```
            while (expression) {
    // insert code
  }
```

This works in a similar way that if statements work, except WHILE the expression is true, the code within the **while** loop will run until the expression becomes false (if it ever does).

Example:

```
#include <iostream>

int main() {
        int i = 0;

        while (i < 10) {
                std::cout << i << std::endl;
                i++;
        }

        system("pause");
        return 0;
}
```

The Following code will print out the value of "x", and then subtract it by one, continuously, until the value of "x" is no longer greater than zero. If you were to run this code, you would get an output of every number from 0 to 9. Once the value of "x" reaches 10, it no longer allows the expression to return true, which is when the **while** loop finishes. **Note** that you should watch out for infinite loops. This is where an error in your code causes the loop to never end, essentially freezing the program. This would happen if there is an error in your logic, and the statement never becomes false. It is pretty much bad for your CPU, because CPU would execute and waste most of the cycles in executing that infinite loop.

Do While Loop

A **do while** loop works in a very similar way to the while loop except for one major difference. The do, as in, what happens when the while statement is true, is called before the condition is asked. What that means is that a **do while** loop will always run at least once, because it does something first and then asks questions later, so to speak.

Example:

```
#include <iostream>

int main() {
        int i = 0;

        do {
                std::cout << i << std::endl;
                i++;
        } while (i < 10);

        system("pause");
        return 0;
}
```

In this example, I directly transferred the logic from the **while** loop to the **do while** loop. As you can see, the only difference is that the logic that gets run while the statement is true, is above the while loop, instead of below it. For this specific example, the output of the code is exactly the same. The output would be different then a **while** loop. If this was a **while** loop, the code within the loop would never run. However, in a do while loop, the **do** happens before the **while**, so the code will always run at least once. During the first run, it never technically gets checked for a true statement until the block of code is processed first. This can be helpful for specific situations, but it is not used as often as a normal while loop.

A very common type of example for this would be when the program needs the user to press Y to close the program. First time the program would print the "Press Y to end the program" and next time it would keep prompting the user to enter Y.

For Loops

For loops allow you an easy way to loop or increment through a specific range of values. The syntax for a **for** loop is as follows:

for (initialization; termination; increment) {

```
                //statements

        }
```

Initialization: Initializes a counter variable, you can initialize multiple variables in the block.
Termination: States the expression to evaluate for true, you can enter any much complex expression and it would be resolved to be true.

Increment: Increments the initialized counter variable, you can also decrement the values in this block.

Example:

```
#include <iostream>

int main() {

        for (int i = 0; i < 10; i++) {
                std::cout << i << std::endl;
        }

        system("pause");
        return 0;
}
```

In this example, we start by initializing the **for** loop. We create a new **int** named “x” and set it to zero. We then state when it will terminate. We say that the loop will continue for as long as “x” is less than 10. We then set the increment to increase the value of “x” by one. The initialization happens only once when the **for** loop is first called. After the loop is initialized, it should run at least once (unless you initialized it outside its bounds). Once it reaches the end of the code per iteration, it will call the increment value. After it increments, it will check whether it should be terminated and if it does not get terminated, the content is called again and the process repeats until termination. For the example code above, the output will be every number from zero to 9.

You need to understand that you can use any variable and the loop would continue until a condition is false. You can evaluate the condition to be false using anything, any operator or expression. You are not only required to increment the value by 1, you can write any code in that section to reflect the changes for the next iteration control.

Arrays

Arrays are data structures which store a fixed amount of variables of the same element. What an array lets you do it, instead of having to create 100 different integers to store 100 different number values, you can create one integer **array** which can store 100 different integer values. This makes it a lot easier to manage the integers, as well as making it a lot more organized and simple.

Declaring Arrays

Declaring an array is similar to declaring a regular variable. The syntax is as follows:

dataType arrayName[];

You can also associate a length of the array in the parenthesis.

dataType arrayName[10];

In the above code, the length can have only 10 elements; 10 is the length. The Arrays above currently hold no data. We declared the array, but not the values in each index of the array:

Example:

int intArray[] = **{ 1, 2, 3, 4, 5}**;

The example above declares an integer array with 5 available slots. What this means is that, intArray can hold 5 different integer numbers within it. Each of these numbers can be editing and accessed independently in the following way:

```
#include <iostream>

int main() {
        int arr[5] = {1, 2, 3, 4, 5};

        system("pause");
        return 0;
}
```

The code above declares the arr with 5 data slots. We can then access those slots by adding an **int** value inside the square brackets beside the array name, which is known as index of the element in the array. The moment that we declared the array with 5 values, we then created 5 slots numbers 0-4. When it was first declared they contained the value zero. We then can access them each independently just like any normal variable. We can even read the value of each index just like a normal variable.

Multidimensional Arrays

Arrays can also be made much more complex by adding multiple dimensions to them. In the above examples, the arrays consisted of only one dimension. By adding more dimensions, it is essentially like adding arrays within our arrays. It may sound complex, but it's relatively easy to understand.

Example:

Int[2][2] intArray;

The following array consists of two dimensions. The first 2 means that we declare the array with 2 index values in it. The second array means that for each of those 2 index values, you can access another 2 index values inside them. As an example, for the index of (intArray[0]) we added another 2 indexes that we can access. If we thing of index zero as just a regular integer, then we can essentially think of intArray[0] being equal to intSecondArray[2]. This is because its array at zero contains 5 indexes. Therefore, since each index of intArray can be thought of as an independent array, we have 2 arrays, all containing 2 indexes, giving us a total of 4 indexes.

We can then access each of these indexes in the following way:

```
#include <iostream>

int main() {
        int arr[2][2] = { {1, 2}, {3, 4} };

        system("pause");
        return 0;
}
```

Combining Loops and Arrays

After reading about of loops and arrays you might have noticed the possible potential to combine them to get greater use and production out of your program. Loops allow us to assign or get values very quickly for large ranges, while arrays allow us to create really large ranges of numbers. How convenient would it be to combine the two of them? Below I'll show a few examples for each type of loop and how to loop through an array quickly, and how to edit a multidimensional array easily.

Example 1:

```
#include <iostream>

int main() {
        int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int i = 0;

        while (i < 10) {
                std::cout << arr[i] << std::endl;
                i++;
        }

        system("pause");
        return 0;
}
```

In this example, we simply declare an array with 10 index values. We then declare an **int** which we need to increment. We tell the **while** loop to continue looping while "x" is less than the length of intArray (which is the amount of indexes it has; 10), and then we increment "x" by one after each iteration. This code quickly and easily edits 10 different values, and prints them to console. All of that in just 6 easy lines.

Example 2:

```
#include <iostream>

int main() {
        int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int i = 0;

        do {
                std::cout << arr[i] << std::endl;
                i++;
        } while (i < 10);

        system("pause");
```

```
		return 0;
	}
```

This example is just like the first example, except we applied the same concept to a **do while**, instead of a **while**. The output is the exact same for this, however you should always be ware of **do while** loops since they always run at least once.

Example 3:

```
	#include <iostream>

	int main() {
		int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
		int i = 0;

		for (int i = 0; i < 10; i++) {
			std::cout << arr[i] << std::endl;
		}

		system("pause");
		return 0;
	}
```

This method is probably my favourite way to do it. The advantage of this is that the counter variable is built inside the loop, so you don’t need to declare another variable or manually increment it. In my opinion, it is just a lot more convenient and organized to work with.

Example 4:

```
	#include <iostream>

	int main() {
		int arr[5][5] =
		{
			{1, 2, 3, 4, 5},
			{6, 7, 8, 9, 10},
			{11, 12, 13, 14, 15},
			{16, 17, 18, 19, 20},
			{21, 22, 23, 24, 25}
		};
		int i = 0;

		for (int i = 0; i < 5; i++) {
			for (int j = 0; j < 5; j++) {
				std::cout << arr[i][j] << std::endl;
```

```
                        }
                }

                system("pause");
                return 0;
        }
```

This example demonstrates the most practical way to loop through a multi-dimensional array. It may look intimidating, but it is actually quite simple. We have two for loops inside of each other, each of them will loop through a specific dimension of the array. For this example, the first time it loops, it will start with i = 0. It will then loop through every single array index where the first index is zero. Once all indexes of [0,j] get called, it increments "i" by one. It will then continue repeat the process all over again until every single index has been called.

Chapter 11
C++ Methods

A method is a collection of code or statements that is grouped together to create a specific function or task. This section will teach you how to create and utilize methods to their fullest extent.

Creating a Method

The syntax to create a method is as follows:

```
modifier returnType methodName(datatype parameters){

    //code
}
```

modifier: Defines what has access to the method, and other properties. You can recall that public or private keywords from the above sections.

returnType: The dataType that the method will return. Does not return anything if set to void.

methodName: The name of the method. Same syntax as variables and uses the identifiers.

Parameters

Parameters are variables that you can give to the method to utilize when you call it. A method can take zero parameters as well or can take a number of parameters that you want to use inside the method body.

Example 1:

```
#include <iostream>

int samplemethod(int x) {
        return x + 1;
}

int main() {
        int x = 0;
        x = samplemethod(x);

        system("pause");
        return 0;
}
```

We create here a method of type **int** called samplemethod which takes one parameter of the **int** data type. We also state that the method is **public** (by default) meaning it can be accessed from sub-classes. In the main method, we declare an **int** “x”. We then set the value of “x” to samplemethod(x). Since sample method has a return type of **int**, whenever the method is called, it will ALWAYS return an **int** value. We gave it the value of “x”, and the method adds one to that value and returns it. Therefore, “x” will now have a value of 1.

Example 2:

```
#include <iostream>

void average(double x, double y) {
        std::cout << "Average is: " << (x + y) / 2;
}

int main() {
        double a = 3.0;
        double b = 6.0;

        average(a, b);

        system("pause");
        return 0;
}
```

The example method above has a data type of **void**. This means that the method will not return any value. This is good if you need the method to just perform a specific function and don't need to assign a value with it. In the example above, we create a method which calculates the average of two numbers. The method simply performs the function of calculating an average and prints it to the console. It will never return any values.

Example 3:

```
#include <iostream>

int a = 1;
int b = 2;

void sum(int x, int y) {
        std::cout << "Sum is: " << a + b;
}

int main() {
        sum();

        system("pause");
        return 0;
}
```

In this example, we create a method called sum() which takes no parameters. When a method takes no parameters it is called with empty brackets, but the brackets must still be there no matter what. This also shows that you can use public variables from the class within any method in that class. The variable does not have to be an argument in the parameters for it to be useable.

Method Overloading

Method overloading is when you have two methods with the same name, except different parameters.

Example:

```
#include <iostream>

void sum(int x, int y) {
        std::cout << "Sum is: " << (x + y);
}

void sum(double x, double y) {
        std::cout << "Sum of double variables is: " << (x + y);
}

int main() {
        sum();

        system("pause");
        return 0;
}
```

In the example above, two methods are exactly the same, except they take in different arguments. Unlike variables, you can name two methods the same, because C++ can differentiate between the two based on their arguments. If you run the above code, you will see that both methods work. One takes the sum of the double values, while the other processes the **int** values.

Assignment

Create a more advanced calculator. Have three values. The first tells you which kind of operation to perform (for example if a == 1, add the values, of a == 2, subtract them). The next two values will be used for manipulation. Make the calculator support addition, subtraction, division, and multiplication, and give each its own method.

Solution

```
#include <iostream>

// Addition method
double Addition(double a, double b) {
	return (a + b); // Returns the sum of a and b
}
// Subtraction method
double Subtraction(double a, double b) {

	return (a - b); // Returns a subtract b
}
// Multiplication method
double Multiply(double a, double b) {
	return (a * b); // Returns a multiplied by b
}
// Division method
double Divide(double a, double b) {
	return (a / b); // Returns a divided by b
}

int main() {
	// Integer that chooses the action
	int action = 4;
	// The variables to manipulate
	double x = 4.5;
	double y = 3;
	/*
	* Based on the value of action we decide what to do with
	* the two variables. Each action is performed in its own method
	* if action is not valid, we print invalid operation, as we have nothing
	* to do for those values.
	*/
	if (action == 1) {
		std::cout << "The sum is " << Addition(x, y);
	}
	else if (action == 2) {
		std::cout << "The difference is " << Subtraction(x, y);
	}
	else if (action == 3) {
		std::cout << "The product is " << Multiply(x, y);
	}
	else if (action == 4) {
		std::cout << "The answer is " << Divide(x, y);
```

```
        }
        else {
                std::cout << "Invalid operation";
        }
}
```

At this point the code above should be pretty self-explanatory. You should attempt to spice things up a little. Maybe add an array in there to play with a few hundred values instead of two. Add some more functions and see how complex you can make this calculator. You have all the knowledge you need to make the best calculator possible.

Recursion

In programming, there is a technique called recursion which simply is the idea of calling a method within itself until a certain condition is met. So basically a loop through a method calling itself until it reaches a certain condition. Here are a few examples:

Fibonacci

1, 2, 3, 5, 8, 13…

1 + 2 = 3
3 + 2 = 5
5 + 3 = 8
8 + 5 = 13

I will show you both the recursive way of doing this and also the iterative way.

Recursive Way

```
#include <iostream>

int FibonacciT(int n) {
        if (n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
                return (FibonacciT(n - 1) + FibonacciT(n - 2));
}


int main() {
        FibonacciT(10);
}
```

Iterative Way:

```
#include <iostream>

int Fibonacci(int element) {
        int sum = 0;
        int f = 0;
        int l = 1;
        for (int e = 0; e < element; e++) {
                sum = f + l;
```

```
                        l = f;
                        f = sum;
                }
                return sum;
        }

        int main() {
                Fibonacci(10);
        }
```

Whole Program

```
#include <iostream>

int Fibonacci(int element) {
        int sum = 0;
        int f = 0;
        int l = 1;
        for (int e = 0; e < element; e++) {
                sum = f + l;
                l = f;
                f = sum;
        }
        return sum;
}

int FibonacciT(int n) {
        if (n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
                return (FibonacciT(n - 1) + FibonacciT(n - 2));
}

int main() {
        // Use the functions here.
}
```

Chapter 12
Inheritance and Polymorphism

Let's redefine classes and introduce objects. Imagine your average suburban neighborhood. If you look at a single street, you'd see a variety of different houses. In fact, if you look at any two houses in a given neighborhood, they are most likely different by colour, by model, or something else. But if you look at all the houses, you may start noticing repetitions; specifically the structure of the houses may be rather similar. This is because, rather than the architect designing and testing hundreds of houses, they just make a "few" (this could actually be a significant number) houses, and then just use different colors and orientations to distract people from the fact that there are duplicates. Classes and objects work a similar way. Classes are models or structures of how an object must be laid out. Objects are like the differences in the houses: their color, their orientation, etc. In a coding way, objects can each have unique values assigned to variables, but must always contain the variables defined in the class.

Classes are defined by the class keyword, followed by an identifier called a class name, and then followed by a block ({}) and an ending semi-colon character. This is at bare minimum. Here is an example of a class called Fruit.

```
class Fruit {

};
```

In this class, we can define variables for use within the class (or outside of the class if the variable is public or protected). For now, simply put public in front of your data types, until access modifiers are explained.

```
#include <iostream>

class Fruit {
public:
        char name[];
        int timeSincePicker;
};
```

Now let's create an object. First we need our main method and a class. We will put our main method inside the class. It does not matter where the main method is, as long as it is written out properly. To instantiate an object (which is essentially building the house in our neighborhood analogy), we simply write the new keyword, followed by the class name of the object you wish to instantiate, followed by parenthesis (like a method, which will be explained why that is in a bit) and a semicolon. So like this:

```
#include <iostream>

class Fruit {
public:
		char name[5];
		int timeSincePicked;
	};

	int main() {
		new Fruit();
	}
```

This on its own is not very useful. We create our object, but then we cannot use it, so we must assign it to a variable. Classes can act as variable data types, and objects are their values.

```
	#include <iostream>

	class Fruit {
	public:
		char name[5];
		int timeSincePicked;
	};

	int main() {
		Fruit fruit;
	}
```

Now we can access the variables inside of our fruit object. To access these variables, we simply write the object name, a dot (.), and the member name. Methods can also be put inside classes and can access variables in its own object. Remember that variables in objects can have different values. Be careful where you write code though, as you can only have code inside methods. Classes can only contain variables, methods and other classes directly.

```
#include <iostream>
#include <string>

class Fruit {
public:
	std::string name;
	int timeSincePicked;

	void checkIfRotten()
	{
		if (timeSincePicked > 100000)
		{
			std::cout << "This " << name << " is rotten!" << std::endl;
		}
		else
		{
```

```
                                        std::cout << "This " << name << " is not rotten." << std::endl;
                        }
            }

};

int main() {
            Fruit pear;
            Fruit apple;

            pear.name = "pear";
            pear.timeSincePicked = 3333333;
            pear.checkIfRotten();

            apple.name = "apple";
            apple.timeSincePicked = 2222;
            apple.checkIfRotten();
}
```

Classes can take on the properties of another class and build on top of them. This is called inheritance. Classes that "extend" another class will have all the variables and methods of the extended class. The class that is extending is called the child, and the class that is being extended to is the parent. To extend, after the class name type a colon (:) and the name of the parent class.

This leads to polymorphism. This means that a child class can be the value for an object of the parent class. This is called upcasting. Here is some code to clear that explanation up.

```
#include <iostream>
#include <string>

class Food {
public:
            std::string name;
};

class Fruit : public Food {
public:
            bool isRotten;
};

int main() {
            Food food;
            Fruit fruit;

            fruit.name = "pear";
```

```
                food = (Food)fruit; // Casting

                // food.isRotten doesn't exist because it is of type food.
        }
```

Casting, or downcasting can convert an object to another type in order to be able to access variables of that type. This has specific rules in that an object cannot be cast into a type that it is not. For example, if Fruit and Meat both extend from Food, and Pear extends from Fruit, and you have an object of type Fruit that has been upcasted to type Food, it cannot be cast into type Meat. This is due to the fact that the original object was not of that type. Similarly, the object cannot be cast into type Pear, because it was not of that type before. Casting is done by simply putting the class you wish to cast to in brackets before the object you wish to cast. You can see the above code sample as an example of type-casting.

Access modifiers pretty much explain themselves — they modify what can access it. Access modifiers can be put on variables, methods, and classes. They always go before the data type of a variable, the return type of a method or the class keyword. There are three access modifiers:

- **public** - Anything can access it.
- **protected** - Only the class itself and its children can access it.
- **private** - Only the class itself can access it.

If you omit an access modifier, the default is protected. Access modifiers are essentially to prevent other programmers from messing up processes within your class. There are very few requirements on putting access modifiers. You could make pretty much everything public and your code would run fine. It is simply to prevent errors by people who do not know what they are doing when interacting with your code (or you, if you forget how your code works).

One common thing programmers use the private access modifier for is to make read-only variables (outside the class at least). The way it is done is to make methods that simply return the value of the variables. This works because outside classes can access the methods, and the method—since it is inside the class—can access the variables. This allows outside classes to access the value, but since they cannot access the variable, they cannot modify the value. Be careful with this system though, because objects are “references”, meaning changing values within them will change the original object, no matter where it is in your code.

```
        #include <iostream>
        #include <string>

        class Something {
        private:
                int aNumber;
                AnotherThing anObject;
```

```
public:
	int getNumber() {
		return aNumber;
	}

	AnotherThing getObject() {
		return anObject;
	}

public:
	Something() {}
	Something(int aNumber, AnotherThing anObject) {
		this->aNumber = aNumber;   // Pointer-type
		this->anObject = anObject; // Pointer-type
	}
};

class AnotherThing {
public:
	int number;

public:
	AnotherThing() {} // Required default constructor
	AnotherThing(int number) {
		this->number = number; // This is a pointer type
	}
};

int main() {
	AnotherThing athing = AnotherThing::AnotherThing(4);
	Something something = Something::Something(4, athing);

	std::cout << athing.number << std::endl;
	std::cout << something.getObject().number << std::endl;

	system("pause");
	return 0;
}
```

Constructors are special methods. They can only be called when creating an object and can only be run once. They do not have a return type (void; as technically they return the object being created), and they can have parameters. Constructors must always have the same name as the class it is in. Constructors can be used to set the initial values of an object.

```
#include <iostream>
#include <string>

class Fruit {
public:
	std::string name;

	void saySomething() {
		std::cout << "The name is " << name << std::endl;
	}

	Fruit() {}
	Fruit(std::string name) {
		this->name = name;
	}
};

int main() {
	Fruit f;
	f.name = "Pear";

	f.saySomething();

	system("pause");
	return 0;
}
```

Constructors can also be overloaded, meaning you can have multiple of the same method as long as the parameter types have a different order or number. In C++ you need to have a default constructor, because each constructor must be shared with C++ to generate class instances.

Chapter 13
Abstract Classes in C++

There are also some special types of classes. Abstract classes are classes that cannot be instantiated—you cannot make an object of an abstract class. You can however make objects of children of the abstract class. This is useful when you want a group of classes to have a common set of variables or methods, but do not want to have an unspecific object. Abstract classes require the use of upcasting and downcasting. Abstract classes are defined by the virtual keyword before along with their functions; specifying the use of override before use. Such functions are not required to have a body, and can be a pure function. When a child inherits from the parent abstract class, it must override, or define abstract methods. Abstract classes can also contain normal methods, in order to add common functionality as in normal classes.

```
#include <iostream>
#include <string>


class Food {
public:
        std::string name;
public:
        virtual void saySomething() = 0;
};


class Fruit : public Food {
public:
        void saySomething() override {
                std::cout << "The name is " << name << std::endl;
        }

        Fruit() {}
        Fruit(std::string name) {
                this->name = name;
        }
};

int main() {
        Fruit f;
        f.name = "Pear";

        f.saySomething();

        system("pause");
        return 0;
```

```
}

#include <iostream>
#include <string>


class Food {
public:
        std::string name;
public:
        virtual void saySomething() = 0;
};


class Fruit : public Food {
public:
        void saySomething() override {
                std::cout << "The name of fruit is " << name << std::endl;
        }

        Fruit() {}
        Fruit(std::string name) {
                this->name = name;
        }
};

class Meat : public Food {
public:
        void saySomething() override {
                std::cout << "The name of meat is " << name << std::endl;
        }

        Meat() {}
        Meat(std::string name) {
                this->name = name;
        }
};

int main() {
        Fruit f;
        f.name = "Pear";

        f.saySomething();

        system("pause");
        return 0;
}
```

Chapter 14
Debugging

Sometimes when programming, your code may give wrong outputs. This is called a logic error. Your IDE will not tell you anything is wrong, because technically your code follows the rules of C++ language specification. It's like writing in any language. You may follow the rules of the language, like grammar, structure, etc., but your paragraphs don't necessarily have to make sense. The most common method of debugging is simply using std::cout and printing out variables to find where values stop making sense. Then you can find the lines that are causing the error and debug that. Another tool when debugging is breakpoints. These are points that act as "checkpoints", where the code will stop executing until the programmer lets the code continue. In Eclipse, on the line you wish to put a breakpoint on, simply move your cursor to the left hand side, right click and press Toggle Breakpoint; this way you can slow down your code to find out where the error is occurring (or when, if you are doing lots of iterations in a loop).

One thing that may or may not be considered debugging is optimization. Once you've written your code, it may be tempting to just leave it and continue on with other tasks, but this may not be what you want to do. If your program gets the desired results eventually, but it takes a long amount of time, then it is not very good. With that being said, there is no single strategy to make code more optimized. Sometimes your logic could simply be a long method of doing something, and sometimes it could be how you implement that logic. Be wary of creating unused variables, as they can take up processing time and memory. It goes through all the variables in your code and removes any ones that are no longer valid. For example, if you make a for loop, when it completes, that variable is no longer valid, so it should be deleted to save on memory. This takes up (small) amounts of processing time.

Here is an optimization example. Say you want to find the distance from one point to another. Math class tells us to use Pythagorean Theorem. But this is rather slow. First the computer must square both sides (which is quite expensive performance-wise), then add them, and finally square root them. This calculates the accurate distance between two points. Another solution though, could be to use "Manhattan distance". This will not get you accurate distances, but it could be good in situations where the exact value is not important, and speed is more important. To do Manhattan distance, simply absolute (which means to make a value positive, so -7 becomes 7 and 8 becomes 8) both the x and y components and then add them. This gives you an inaccurate distance, but it is much faster than its more accurate counterpart. This is particularly good when you are guessing the fastest route. Rather than constantly calculating the distance accurately, Manhattan distance is a cheap alternative and will get you a near enough distance.

Chapter 15
Enums and Generic Types

Enums

Enums are essentially presets. They can only have a limited number of possibilities. They are defined similar to classes and interfaces except using the enum keyword instead. They can have constructors and variables. Enums cannot be instantiated or be children (but they can implement interfaces). Be careful when accessing variables within an enum, because if you modify an enum's variables, all instances of that enum will be modified. To fill an enum, you need a list of all presets. This list must be at the top of your enum block. If your enum does not have a constructor, you may simply write out the possible values as if it were a variable name. The rest would separate by commas, and end with a semicolon. If your enum does have a constructor, you must write the variable name, then the parameters of the constructor (meaning the values), separated by commas and ending with a semicolon. After the semicolon, you may go on to write your variable declarations, constructor(s) and any methods you wish. The constructor(s) cannot be public, only protected or private.

```
#include <iostream>

enum Color {
        Red, Green, Blue
};

int main() {
        Color c = Red;
        if (c == Red) {
                std::cout << "Color is red." << std::endl;
        }
        else {
                std::cout << "Color is not red" << std::endl;
        }

        system("pause");
        return 0;
}
```

Generic Types

Generic types allow us to make classes that are generic, meaning they can act differently based on what type it is given. Generic types are given a list of classes by the programmer that will be used to define certain object types. Generic types are created by placing angled brackets (< >) after the class name and then a list of type identifiers. These work just as variable names, but the convention is to use capital letters, specifically T (for type) for one of them.

These type identifiers can be used in place of data types. This allows you to keep certain variables a consistent type in order to prevent the need for casting. When you're creating a variable with the data type of the class using generic types, the data type most also has angled brackets containing a list of types corresponding to the class's list of types. The object you create will also need the same set of angled brackets before the parameters.

In C++, this generic programming is done using templates. You can create a template for a class or object and set a type to it. Then the developers can use different objects in those templates and consume them. For example, a vector object is a single container, but can contain objects of different type specified in their angled brackets.

```
#include <iostream>
#include <string>


class Food {
public:
        std::string name;
public:
        virtual void saySomething() = 0;
};


class Fruit : public Food {
public:
        void saySomething() override {
                std::cout << "The name of fruit is " << name << std::endl;
        }

        Fruit() {}
        Fruit(std::string name) {
                this->name = name;
        }
};

template <class T>
class Stack {
```

```
public:
	T object;
};

int main() {
	Stack<Fruit> stack;
	std::cout << stack.object.name << std::endl;

	system("pause");
	return 0;
}
```

Chapter 16
Exception Handling and File Handling

Exception Handling

When you're coding, you may land in situations where your code returns an exception when attempting to do something. For example, if an object requires an integer in its constructor, but you feed it null instead, the compiler will return an error indicating that the object didn't receive the appropriate parameter.

Another example would be when you're working with file handling which will later be taught, and you're trying to read a file in a directory that doesn't exist. The compiler would then again return an exception such as "**FileNotFoundException**" - there are different exceptions but this type of exception is regularly found in File Handling.

In order to regulate exceptions in your code, you can use the try-catch statements. What these statements do is check if a block of code returns an exception, and if it does, instead of halting the whole operation of the program, it will catch the exception and try to either fix it or do something alternatively depending on what the programmer codes. Here is a use of a try-catch statement:

```
#include <iostream>

int main() {

        try {
                // Code to try out
        }
        catch(std::exception e) {

        }

        system("pause");
        return 0;
}
```

The try statement does not require parameters but the catch exception requires the parameter of an exception object, and when an exception does occur, the variable "e" of type Exception is given data, and the "e" variable can be manipulated to either print out the error or do various other things. A commonly used method for the e variable is: e.what();

File Handling

File Handling is the idea of reading and writing external files in the machine that the program is being coded on. In C++, file handling isn't that complicated when a few basic ideas have been understood. In this guide, we will be going over some basic File Handling fundamentals for reading text files.

Writing to Files

To write to a text file in C++, you must output/write to a text file. Streams can be used to write the content of your data to your file destination. A lot of the steps taken to write to a file is similar to reading a file. The first thing you must do is import everything that has to do with Input and Output - the import directory for this is:

```
#include <fstream>
```

We will use the text file "test.txt" as an example.

The first thing you must do is declare your file name, you can do this either within the declaration of the object or within a string variable which can then be later on used within the constructor. Remember, when writing to a file, it will either overwrite a file with the same file name OR create a file with the file name specified if it does not exist.

Example:

```
std::string name = "test.txt";
```

You can use the constructor for fstream object and pass the file name to it. Or you can simply just hard-code the values in to it.

```
std::fstream fs("E:\\Test.txt");
```

That done, you can now call the .write function on it and pass the string content and the number of characters to write (mostly it is the number of characters in the string, or less than the string if you want to store a substring. Otherwise you cannot have length more than buffer size).

```
#include <iostream>
#include <fstream>

int main() {

        try {
                // Code to try out
                std::fstream fs("E:\\Test.txt");

                fs.write("Hello World", 10);
        }
        catch(std::exception e) {
                // In case of any error
        }

        system("pause");
        return 0;
}
```

Reading Files

In order to read the content of a file, you can use the same header file and then call some functions on it to read the buffer from it.

You can use the fstream object, and then use the getline function of the object to get the lines of characters from the stream and store it in some sort of variable that can hold multiple characters; yes, an array of characters.

Then you can print the output of the program easily to the stream using the std::cout command.

```
#include <iostream>
#include <fstream>

int main() {

        try {
                // Code to try out
                std::fstream fs("E:\\Test.txt");
                char lines[5];

                fs.getline(lines, 10);

                std::cout << lines << std::endl;
        }
        catch(std::exception e) {
                // In case of any error
        }

        system("pause");
        return 0;
}
```

Chapter 17
Example Programs

How about we test our knowledge by doing some example programs. For these, we will be using the 2014 Canadian Computing Competition. The problems are linked here:

https://cemc.math.uwaterloo.ca/contests/computing/2014/stage%201/juniorEn.pdf

Here's a solution to J1:

```
#include <iostream>
#include <fstream>

int main() {
	int a, b, c;
	std::cin >> a;
	std::cin >> b;
	std::cin >> c;

	if (a > 0 && b > 0 && c > 0 && a < 180 && b < 180 && c < 180)
	{
		if (a == 60 && b == 60 && c == 60)
		{
			std::cout << "Equilateral" << std::endl;
		}
		else if ((a + b + c == 180) && ((a == b) || (b == c) || (c == a)))
		{
			std::cout << "Isosceles" << std::endl;
		}
		else if ((a + b + c == 180) && ((a != b) && (b != c) && (c != a)))
		{
			std::cout << "Scalene" << std::endl;
		}
		else
		{
			std::cout << "Error" << std::endl;
		}
	}
	system("pause");
	return 0;
}
```

So what did we do there? First, we create a scanner. Then we get the three angles given to us. We then check to see if the angle is a valid angle. Next, we check if the triangle has all angles equal to 60 degrees, meaning it is equilateral. If so, we print that out. Otherwise, we check if the angles add up to

180 to check if the triangle is valid, and we check if any two angles are equal. This would prove that it is an isosceles triangle. If so, print it out. Otherwise, we check if the triangle is valid and if all the sides are dissimilar, meaning it is scalene. If so, print it out. If none of those situations are true, print out “Error”.

Here’s a solution to J2:

```
#include <iostream>
#include <fstream>

int main() {
        int chars;
        char votes[100]; // 100 as a limit of the buffer

        std::cin >> chars;
        std::cin >> votes;

        if (chars > 0 && chars < 16)
        {
                int voteA = 0;
                int voteB = 0;

                for (int i = 0; i < chars; i++)
                {
                        if (votes[i] == 'A')
                        {
                                voteA++;
                        }
                        else if (votes[i] == 'B')
                        {
                                voteB++;
                        }
                }

                if (voteA == voteB)
                {
                        std::cout << "Tie" << std::endl;
                }
                else if (voteA > voteB)
                {
                        std::cout << "A" << std::endl;
                }
                else
                {
                        std::cout << "B" << std::endl;
                }
        }

        system("pause");
        return 0;
}
```

First, we create a scanner. We get the first line of input which is the number of votes. We get the second line of input which is the votes. Then we loop through each letter of the second line and if the letter is a capital A we add one to the voteA variable. If it is a capital B, we add one to the voteB variable. Finally we check whether voteA and voteB are equal; if so, it is a tie. If voteA is greater than voteB, A wins, otherwise, B wins.

Here's a solution to J3:

```
#include <iostream>
#include <fstream>

int main() {
        int rounds;
        std::cin >> rounds;

        if (rounds > 0 && rounds < 16)
        {
                int scoreA = 100;
                int scoreD = 100;

                for (int i = 0; i < rounds; i++)
                {
                        int rollA;
                        int rollD;

                        std::cin >> rollA;
                        std::cin >> rollD;

                        if (rollA > 0 && rollA < 7 && rollD > 0 && rollD < 7)
                        {
                                if (rollA > rollD)
                                {
                                        scoreD -= rollA;
                                }
                                else if (rollA < rollD)
                                {
                                        scoreA -= rollD;
                                }
                }
            }

            std::cout << scoreA << std::endl;
            std::cout << scoreD << std::endl;
        }

        system("pause");
        return 0;
}
```

As usual, we create a variable. We make a variable to store the number of rounds and we store the input. We make 2 variables: one to store Antonia’s score, and one to store David’s score. We then do a loop that runs for every round. We get Antonia’s roll and David’s roll and store them in respective variables. We check if each roll is within range. Then we check if Antonia rolled higher, and if so, we remove that roll from David’s score. Otherwise, we check if David rolled higher, and if so, we remove that roll from Antonia’s score. We exclude a final else statement, because if neither is higher than the other, then the rolls are equal, meaning neither player should lose points.

Chapter 18
Bonus: Algorithms

Let’s take a look at a few low difficulty problems on ProjectEuler (http://projecteuler.net) that we can solve and let’s go over how we can solve them using C++. I will explain how we can approach each algorithm to help you start thinking in a programmer’s mind and then you can try to solve it. It is recommended that you sign up for this site and practice your new found C++ skills on there.

Multiples of 3 and 5

Problem 1

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.
Find the sum of all the multiples of 3 or 5 below 1000.

Now let’s think about this. What exactly is this problem asking from us? This problem is asking us to find all the multiples of 3 and 5 below 1000. Now mathematically, without programming, let’s think about how we would approach that. Anything that is divisible by 3 or 5 that is within the sequence from 1 to 999 would be added to a list of numbers.

After finding all the numbers that are divisible by 3 or 5, we would then take a look at our list of numbers and add all these numbers up together and then provide the sum to ProjectEuler to see if we got the answer correct. This seems pretty easy enough but it would take a long time to complete if we were to simply have a pen and paper handy. But now, with our programming knowledge, we are able to figure out how to get the sum of all the multiples of 3 and 5 below 1000.

In order to actually loop through from 1 to 1000, we should use a loop. It doesn’t matter what type of loop we would use but we need to use a loop that affects a variable to increment from 1 to 1000, not including 1000 because the question is asking us to find the sum of all the multiples of 3 or 5 below 1000, not below or equal to.

Then, through each iteration of the loop, we must check if the variable being affected by the loop is divisible by 3 or 5. There are multiple ways to do this, which can vary from checking the remainder (using the % symbol) or dividing and checking whether or not it is a whole number. This is pretty trivial to figure out on your own. The next thing we must do if the number is divisible by 3 or 5, is to add it to a list. We can either add it to a list and then loop through the list (or array) or add all the numbers together or we can create a variable before the loop and simply add on to any number it finds is divisible by 3 or 5. An efficient way to do it is by doing the latter.

The reason I pointed out the array way is to essentially let you know that there are multiple ways to solve problems in programming but it is important to find out which way is the most efficient way using your knowledge and a spice of logic. Now try programming this using what you know in C++ and the logic I have provided and try to create a block of code that can do the functionality accordingly.

Let’s move on to the next problem on Project Euler.

We first check if our solution works with their example:

```
#include <iostream>

int main() {
	int limit = 10;
	int sum = 0;
	for (int i = 3; i < limit; i++) {
		if ((i % 3) == 0) {
			sum += i;
		}
		else if ((i % 5) == 0) {
			sum += i;
		}
	}

	std::cout << "The sum below " + limit << " is: " << sum << std::endl;

	system("pause");
	return 0;
}
```

We then get the output: The sum below 10 is 23

The output is correct, so we are going to input the number they ask to solve the problem with the following code:

```
#include <iostream>

int main() {

	int limit = 1000;
	int sum = 0;
	for (int i = 3; i < limit; i++) {
		if ((i % 3) == 0) {
			sum += i;
		}
		else if ((i % 5) == 0) {
			sum += i;
		}
	}

	std::cout << "The sum below " << limit << " is: " << sum << std::endl;

	system("pause");
	return 0;
}
```

This time the limit is 1000. The output is now:

The sum below 1000 is 233168.

Even Fibonacci Numbers

Problem 2

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

In this problem, we are told the pattern the Fibonacci sequence is generated by and then we are told what we must solve. This problem asks us to find the sum of the even valued term values in the Fibonacci sequence. Don't get confused and assume they are talking about the term indexes, but rather the term values.

There are technically two problems in this problem, one of which we have already completed. You see, in the last problem, we were checking whether or not a number was divisible by x or y in a range from 1 to 1000 and then adding it to an accumulation variable. Well the same idea applies here, where while we generate the Fibonacci sequence, we check whether or not the term value that was generated in the certain iteration is even or odd. If the term is even, then we simply add it in to an accumulation variable, if not, we do nothing about that iteration.

So truly, we're only really solving the Fibonacci problem here. This is the great thing about practicing algorithms, you start to gain a muscle memory over them so you get better and better at doing them efficiently the more you practice.

So how exactly do you generate the Fibonacci sequence? Well we know that you start with 1 and 2. Then what happens is you add the two terms together to get the third. Then you get the last 2 terms and add those together to get the fourth, and so on.

1, 2, 3, 5, 8, 13…

1 + 2 = 3
3 + 2 = 5
5 + 3 = 8
8 + 5 = 13

Now that we have a better understanding of how the Fibonacci sequence works, let's see how we can recreate that pattern in C++. First, we must create a loop that has the amount of iterations identical to the term value that we want to get. So if we want to get the 3rd term value, then we have 3 iterations, and if we have 3 iterations, we're going to end up getting the value of 3 because in the sequence, the third term value is 3.

Although, before creating the loop, what we should do is declare two variables, one that is initialized at one, and another which is initialized at two. Why? Because we should simulate exactly how the Fibonacci sequence works by having these two variables dynamically change as the iterations of the loop go on. Let's go back to our loop and look at how we would do this. First, let's create a quick variable that is the sum of the two variables we initialized before the loop. Next, let's check if the second variable that we declared as 2 is even, and if it is, then we add it to an accumulation variable. Next, we assign our variable that was assigned one, to the second variable that was assigned two and then assign our second variable that was assigned two, to the sum of the two numbers.

Now if you read that carefully, you would understand that we are simply cycling through the Fibonacci sequence in this exact algorithm that I have just specified. Now you can try it out!

Solution for first 10 terms by specifying limit of 89:

```
#include <iostream>

int main() {

	int limit = 89; // declaring limit
	int sum = 0; // declaring sum
	int left = 1; // first term of fib seq
	int right = 2; // second term of fib seq
	while (true) { // loop continuously
		int sumLR = left + right; // adding two prev terms
		left = right; // swapping values
		if (right > limit) { // if goes over limit, break
			break;
		}
		else if ((right % 2) == 0) { // if even, add to sum
			std::cout << right << ", ";
			sum += right;
		}
		right = sumLR; // swap values for next iteration
	}

	std::cout << "The sum is " << sum << std::endl;

	system("pause");
	return 0;
}
```

Solution for problem with limit changed to four million:

```
#include <iostream>

int main() {
	int limit = 4000000; // declaring limit
	int sum = 0; // declaring sum
	int left = 1; // first term of fib seq
	int right = 2; // second term of fib seq
	while (true) { // loop continuously
		int sumLR = left + right; // adding two prev terms
		left = right; // swapping values
		if (right > limit) { // if goes over limit, break
			break;
		}
		else if ((right % 2) == 0) { // if even, add to sum
			std::cout << right << ", ";
			sum += right;
		}
		right = sumLR; // swap values for next iteration
	}

	std::cout << "The sum is " << sum << std::endl;

	system("pause");
	return 0;
}
```

Let’s check out another Project Euler problem.

Largest Prime Factor

Problem 3

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143 ?

Now in order to do this problem efficiently and under a minute, we must carefully consider how we want to approach solving this problem. Since it is asking us to find the largest prime factor for a really high digit number, it can cause long processing time for the program to figure it out if our algorithm isn't properly configured. That is why we must think about this carefully.

I know that if I take a number like 100 and wanted to figure out the largest prime factor I would do it by doing the following:

100 / 2 = 50
50 / 2 = 25
25 / 2 = Remainder
25 / 3 = Remainder
25 / 4 = Remainder
25 / 5 = 5
5 / 2 = Remainder
5 / 3 = Remainder
5 / 4 = Remainder
5 / 5 = 1

Therefore, the largest prime factor of 100 is 5.

Once my answer has reached one, I know that I have reached my greatest prime factor. This is an efficient way of doing the algorithm because it means that I am not constantly processing the large number that we are to find the prime factor of, but rather are deducting its size as a number as we divide it until we find the largest prime factor of that number. This makes sense because all we are doing is simply slowing reducing the number as we change its ratio through division until we reach a certain point where we can't divide by anything other than 1 or itself, which is what a prime number's definition is.

So, how would we approach this problem programmatically in C++? Well, the first thing we must do is again, create a loop, but right before it, create a boolean that declares the MaxPrimeFound variable as false. Once we have this complete, we are able to process a loop from 1 to the number's variable itself. While we are processing, we try dividing the variable being affected in the loop's process by a factor of 2 for example, and checking whether or not it is a whole number. If it is a whole number, we continue on. If it is not a whole number, we increment the dividing factor until we reach a point where it divides evenly. If the number that it divides evenly is in to itself, then we know we have indeed found the largest prime factor of the large digit number. We simply repeat this process until it divides into itself and becomes 1. Good luck!

Solution using their example:

```
#include <iostream>

int main() {
            int number = 13195;
            int factor = 2;
            int largestPrimeFactor = 0;

            while (true) {
                        if ((number % factor) == 0) {
                                    number /= factor;
                        }
                        else {
                                    factor++;
                                    if (factor == number) {
                                                largestPrimeFactor = number;
                                                break;
                                    }
                        }
            }

            std::cout << "The largest prime factor is: " + largestPrimeFactor;

            system("pause");
            return 0;
}
```

Output: The largest prime factor is 29.

This is correct in the circumstance of 13915. Let's figure out the problem's solution though.
Solution for the problem:

```
#include <iostream>

int main() {
            long number = 600851475143L;
            int factor = 2;
            long largestPrimeFactor = 0;

            while (true) {
                        if ((number % factor) == 0) {
                                    number /= factor;
                        }
```

```
                else {
                        factor++;
                        if (factor == number) {
                                largestPrimeFactor = number;
                                break;
                        }
                }
        }
        std::cout << "The largest prime factor is: " + largestPrimeFactor;
        system("pause");
        return 0;
}
```

Output: The largest prime factor is 6857.

Chapter 19
Final Words

This is the start of your journey as a C++ programmer. You have barely scratched the surface with this guide as learning the syntax and conventions of a language is just the beginning. The most important part of programming is the logical aspect of it. Sure, you may know how to loop through an array of variables like a list of shopping items but if someone asks you to process an image using your knowledge of programming, and with the help of an API and some thinking, you can figure out how you are able to invert colors of an image, flip it, rotate it, scale it, etc.

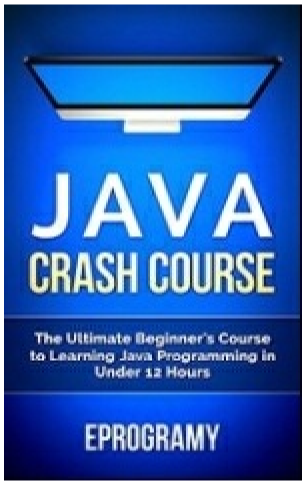
The real programming comes in the logical portion of the mind. It's similar to when you're learning any other language, like English for example. You may understand the grammar rules and the conventions like adding periods to the end of sentences, but to be able to write clean and logical thought-out and structured essays is where the true skill lies. The same concept applies to programming where the person writing the code, must know how to apply his knowledge of the rules in the considered language, like C++, and use it to his advantage to come up with neat programs.

The knowledge and understanding of programming is truly great because it's the closest thing to having a power. You can literally create something out of an empty notepad, from scratch and have it function to do things you want it to do. Whether it be a bot to analyze the stock market and come up with predictions or creating a game. That choice is yours.

In this guide, you have learned the fundamentals of C++. You haven't learned all the possible methods that can be used in the language, but that isn't the point. The point of this guide was to set you on a journey to discover objects and methods that you need in order to help you to create programs that you desire. You have been given the optimum knowledge to understand reading an API and be able to understand what it is saying and adding to your code.

Good luck as a new-born C++ programmer!

Eprogramy


JAVA
CRASH COURSE
The Ultimate Beginner's Course to Learning Java Programming in Under 12 Hours
EPROGRAMY

Preview of JAVA CRASH COURSE - The Ultimate Beginner’s Course to Learning Java Programming in Under 12 Hours

Introduction
Welcome to Your New Programming Language

So, you've decided to learn Java Programming? Well, congratulations and welcome to your new Programming Language! You're going to love it!

In Eprogramy we believe that the foundation of a good education is to provide the tools able to open the doors of the future. It is indisputable that the world has entered an era in which we all have everything at the fingertips. Social needs have changed. Thus, today communication has changed. Communication opens doors, so it is our challenge to learn how to communicate. In Eprogramy we want you to learn how to communicate with a new language: The programming language. This language allows us to use our PC or notebook to create.

Maybe the road is difficult and hard, but we will give the reader the tools to make learning enjoyable and fruitful. We'll show all aspects necessary to learn how to program. From the installation of software to the solution of the most common programming problems and much, much more. Always with the help of numerous examples that will contribute to a better understanding

We know what we do and believe that you are fully capable of incorporating our teachings.

The doors of the future are here. Let's go step by step together.

Let´s get started!

Eprogramy Team

Chapter 1
Java Programing Language

History of Java

Java is a programming language that was first publicly available in 1995, created by James Gosling from Sun Microsystems (Sun) in 1991. Sun Microsystems was then inherited by Oracle, and is the corporation in charge of the programming language's faith. Java then became open source available under the GNU General Public License (GPL).

The language gets much of its syntax from C and C++ but isn't as powerful because it requires less for the user to do (less customization but more simple). Something like garbage collection, the process of reducing memory being used from the program, is automated in Java.

It was originally designed for interactive television but surpassed the technology and design of the digital cable television industry at the time. There were five principles that were used in the creation of the Java programming language:

1. It must be "simple, object-oriented and familiar"

2. It must be "robust and secure"

3. It must be "architecture-neutral and portable"

4. It must execute with "high performance"

5. It must be "interpreted, threaded, and dynamic"

Another important design goal to note is portability, which is was a key factor in Java's sudden popularity. The portability in this context means that the code written in the Java platform can be executed in any combination of hardware and operating system.

Unfortunately, Java has a reputation for being a slower programming language (requiring more memory) than other languages like C++ but as time went, Java 1.1 was introduced and program execution times were significantly improved.

Java was built exclusively as an object-oriented programming language, which doesn't mean much right now, but it will later in the guide. Object-Oriented programming allows for efficient, organized and powerful code to be created and will be seen throughout this guide.

What is Java?

Java is a programming language that has multi-platform capability, meaning that you can program Java for any type of device, whether it is an Android phone, a Windows computer or an Apple product. Due to the flexibility of Java, it has made it one of the most popular programming languages used through the globe by many programmers. Java can be used to create web applications, games, windows applications, database systems, Android apps and much more.

Java is different from other programming languages because of its simplicity and powerful nature. That combination makes the Java programming language great to use. Java is a simple programming language because it doesn't expect too much from the user in terms of memory management or dealing with a vast and complex hive of intricate classes extending from each other. Although this doesn't make much sense right now, it will make sense when reaching the point of learning inheritance in Java.

A Java program is run through a Java Virtual Machine (JVM) and is essentially a software implementation of an Operating System which is used as a way to execute Java programs. The compiler (process of converting code into readable instructions for the computer) analyzes the Java code and converts it in to byte code, allowing the computer to understand the instructions issued by the programmer and execute them accordingly.

When downloading and installing Java, the distribution of the platform comes in two ways; the Java Runtime Environment (JRE) and the Java Development Kit (JDK). The JRE is essentially the Java Virtual Machine (JVM) where the Java programs will run on. JDK on the other hand is a fully featured Software Development kit for Java which includes the JRE, compilers, tools, etc.

A casual user wanting to run Java programs on their machine would only need to install JRE as it contains the JVM which allows Java programs to be executed as explained before. On the other hand, a Java programmer must download JDK in order to actually program Java programs.

Chapter 2
Installation of Java

In order to install Java on to a machine, you must download the following:

1. IDE for Java Developers

2. Java JDK

The download of these two tools will put you on your way to becoming a Java programmer. An IDE (integrated development environment) is a packaged application program used by programmers because it contains necessary tools in order to process and execute code. An IDE contains a code editor, a compiler, a debugger, and a graphical user interface (GUI). There are many different type of IDE's but the most commonly used ones are:

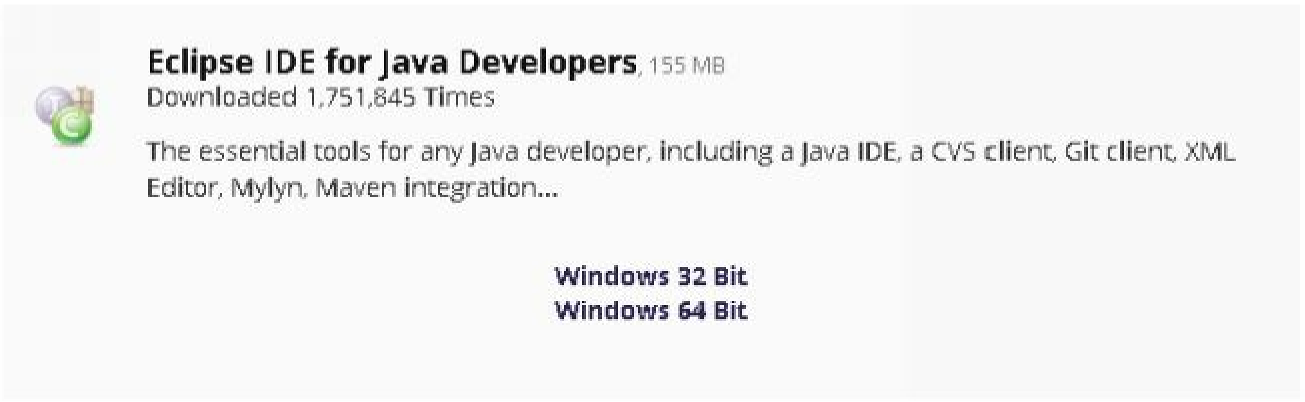
1. Netbeans

2. Eclipse

In this guide, it would be recommended to use Eclipse because of its simplistic nature. In order to download Eclipse, please use the following link:

https://eclipse.org/downloads/

Once you have reached this link, you will have to find this:



Eclipse IDE for Java Developers, 155 MB
Downloaded 1,751,845 Times

The essential tools for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Mylyn, Maven integration...

Windows 32 Bit
Windows 64 Bit

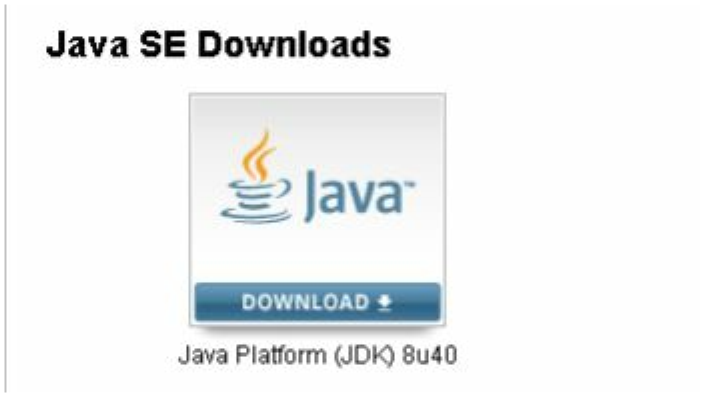
Then, select either the Windows 32 Bit OS or Windows 64 depending on the type of OS / processor you have.

Once the IDE has been installed, we'll move on to downloading and installing the JDK which will allow us to interact with a coding editor in order to execute and create Java code.

To download the JDK, go to the following link:

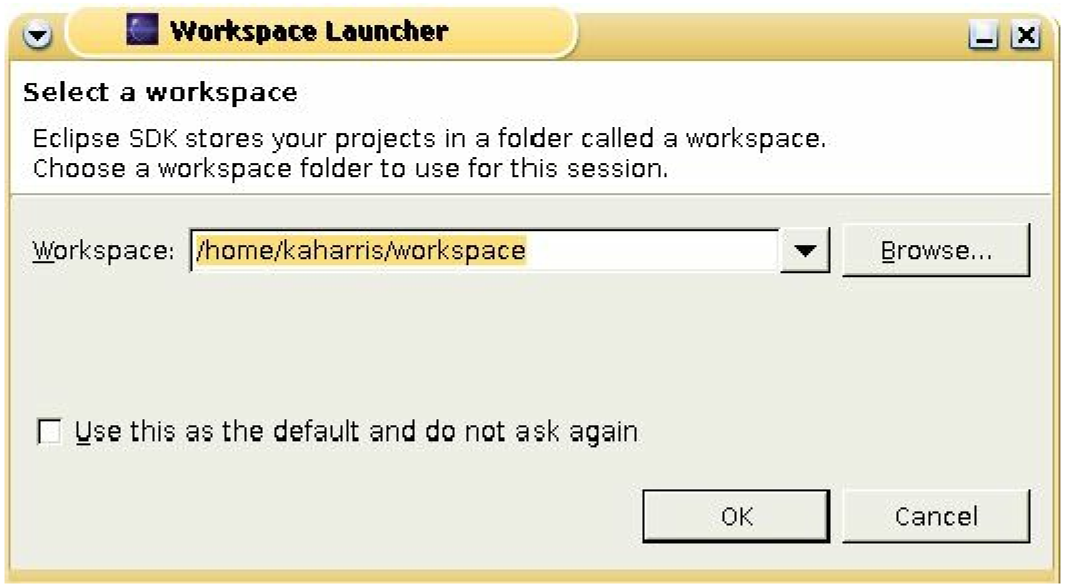
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Upon entering this link, find this figure:



Once you've installed the JDK, you are able to launch Eclipse accordingly.

The folder that was extracted from the Eclipse download will contain an eclipse.exe that can be launched. Once this has been launched, you will be met with the following figure:


Workspace Launcher

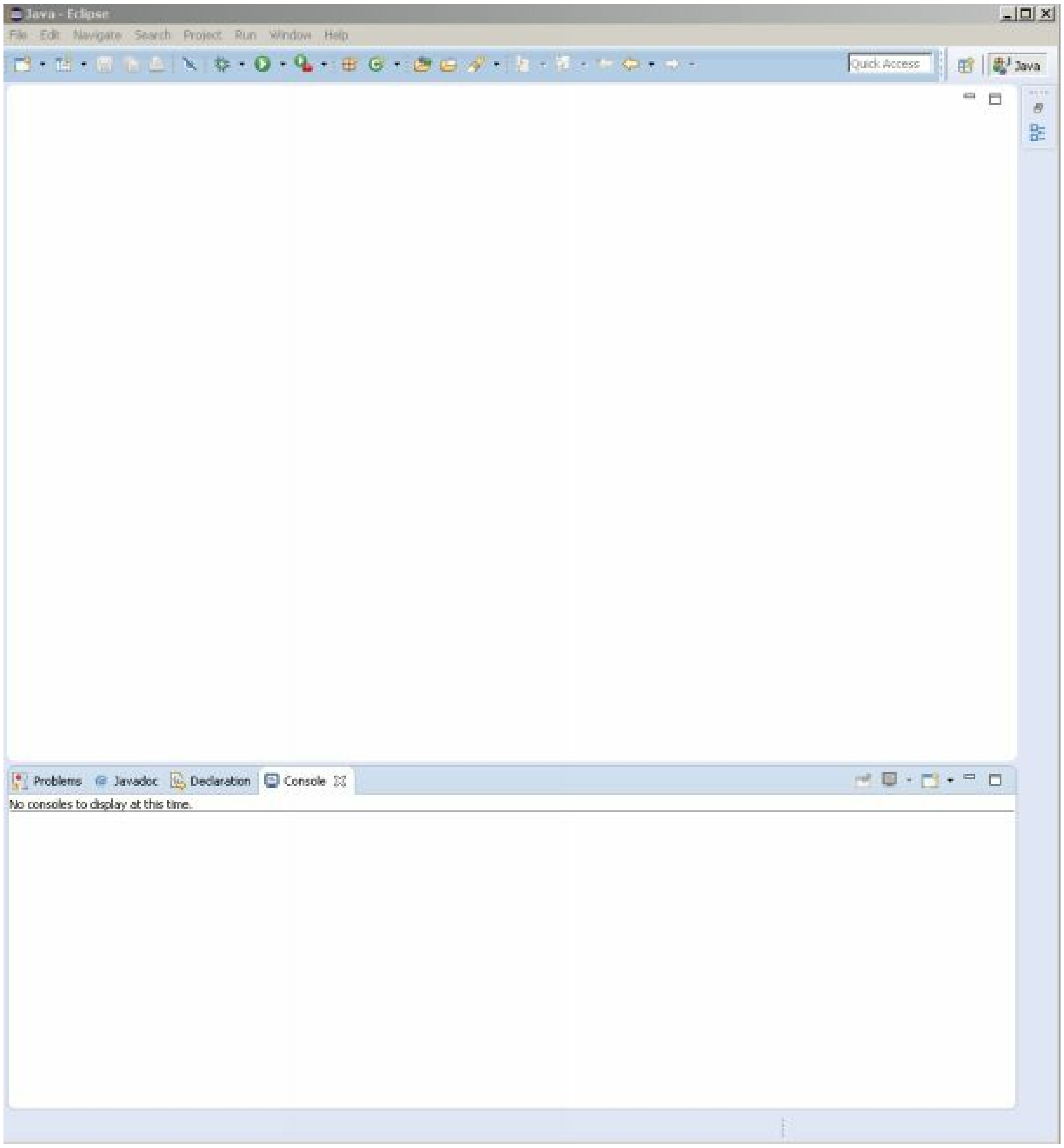
Select a workspace

Eclipse SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /home/kaharris/workspace

All this prompt is asking for is where to set up the output directory for where all the code written by you is going to be saved. Once this has been selected accordingly, click "OK" to continue.

You should now be on this screen (or something similar):

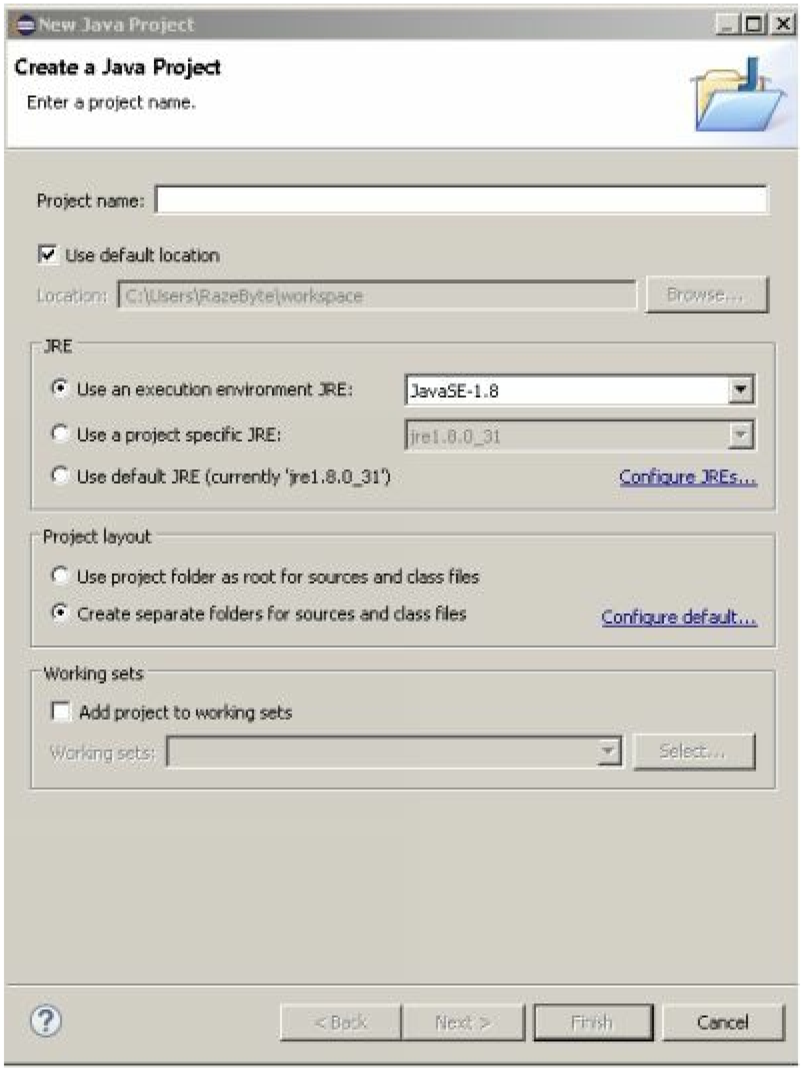


No consoles to display at this time.

Instructions:

1. Click File

2. Click New → Java Project



New Java Project

Create a Java Project

Enter a project name.

Project name:

[✓] Use default location

Location: C:\Users\RazeByte\workspace

JRE

(●) Use an execution environment JRE: JavaSE-1.8

( ) Use a project specific JRE: jre1.8.0_31

( ) Use default JRE (currently 'jre1.8.0_31')

Project layout

( ) Use project folder as root for sources and class files

(●) Create separate folders for sources and class files

Working sets

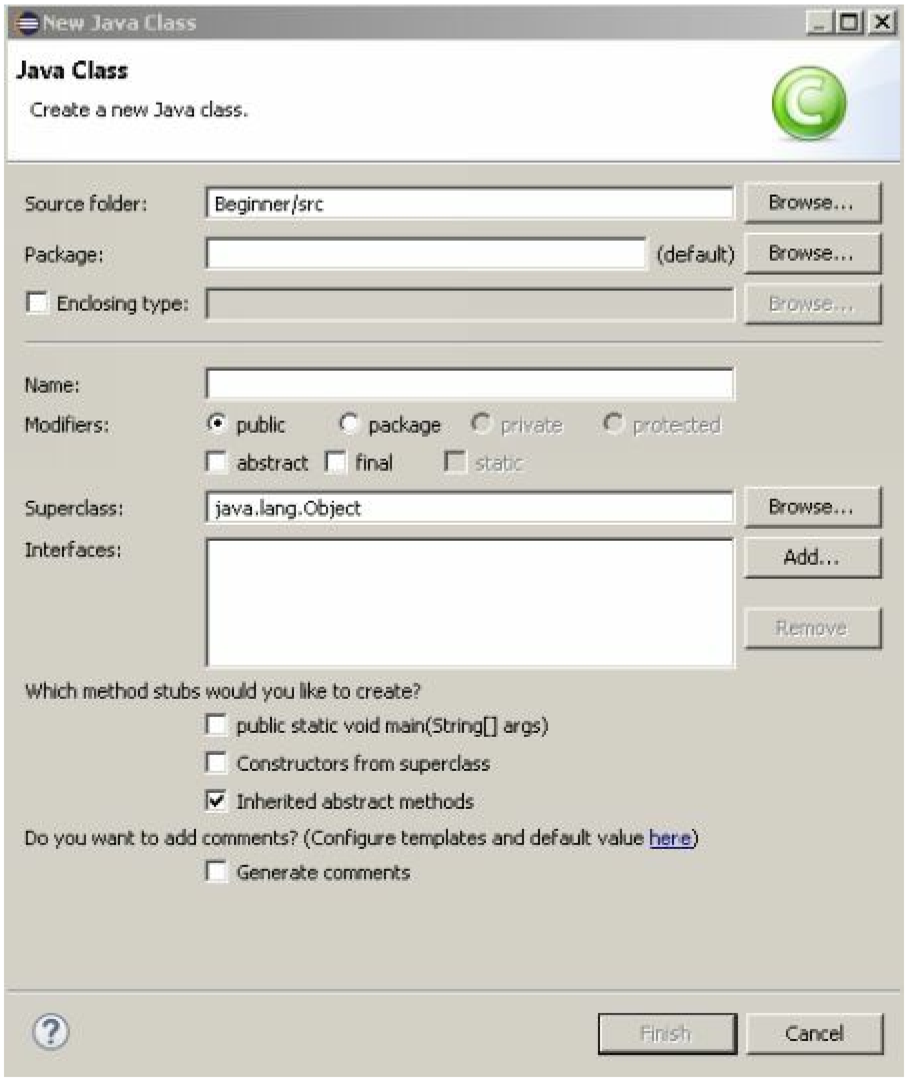
[ ] Add project to working sets

Working sets:

3. Type in a project name in the “Project name:” field

4. Click “Finish”

5. Now right click “src” → New → Class


New Java Class

Java Class

Create a new Java class.

Source folder: Beginner/src

Package: (default)

Enclosing type:

Name:

Modifiers: [✓ public] package private protected

abstract final static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

public static void main(String[] args)

Constructors from superclass

[✓ Inherited abstract methods]

Do you want to add comments? (Configure templates and default value here)

Generate comments

6. Fill in the name field to anything with *letters* and no special characters. Example: “ThisIsAClass”

7. Click Finish

8. You will now be presented with a screen that says:

```
public class ThisIsAClass {

}
```

You have successfully completed the installation of Java!

About the Author

Eprogramy Academy was created by a group of professionals from various areas of IT with a single purpose: To provide knowledge in the 3.0 era.

Education is changing as well as social needs. Today, in the era of information, education should provide the tools to create and to solve problems in a 3.0 world.

At Eprogramy we understand this and work to give people appropriate responses in this context.

Keeping this objective in mind, we offer a wide variety of courses to teach the basics of many programming languages. We believe that anyone can learn a programming language and apply the lessons in order to solve problems. In our academy we provide the essential tools to allow everyone to incorporate into the daily life a set of solutions obtained through programming.

Possibilities and solutions are endless.

In short, at Eprogramy we are committed to help everybody to decodify the messages of the future.