Intro To C++ & Unreal Engine.

Ron T.

1st Edition

About this book:

Author: Ron T.

I created this book for the sheer enjoyment of creating video games and playing them. I hope younger developers can use this book to get a grasp of C++, and hopefully make quality diverse games.

Chapter 1

Creating Our First Program

Introduction

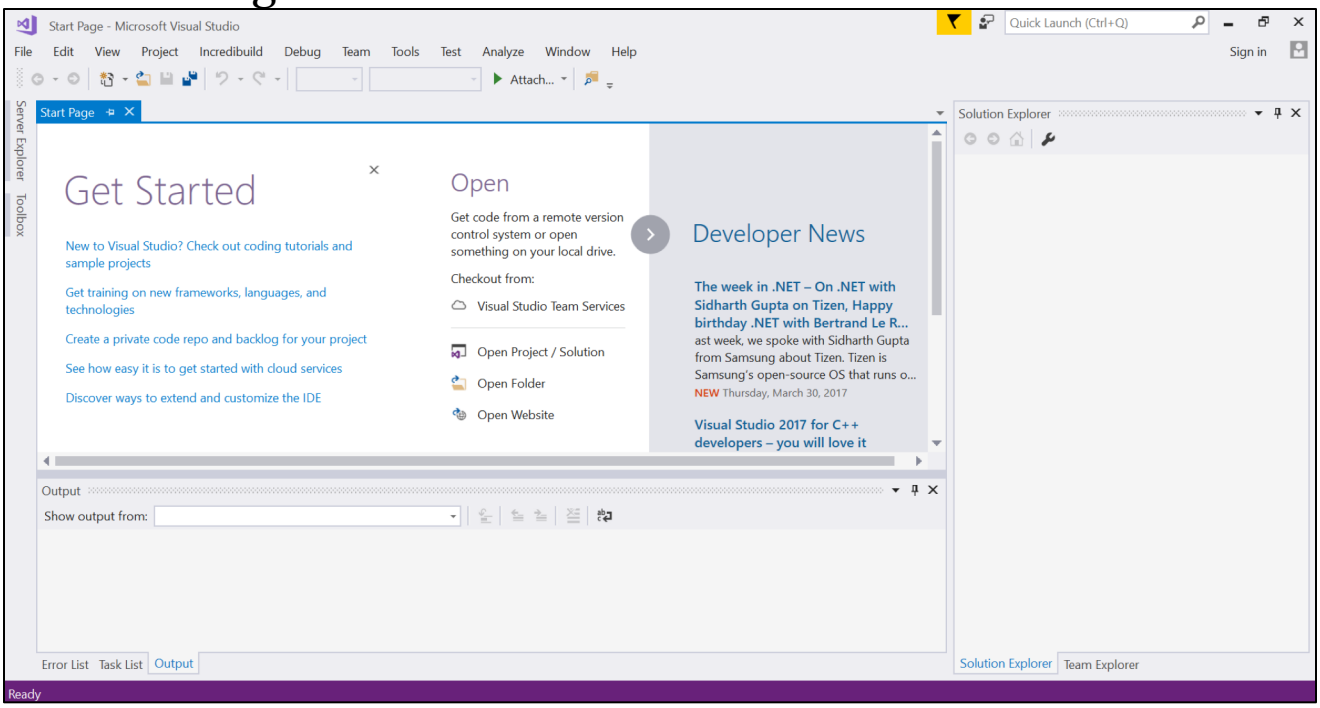
```
void main()
{
    return;
}
```

Before we create a program, we need to know what it is first. A program provides the computer or machine with a list of coded lines that a **programmer** created. A **programmer** is a person who codes computer software. A few of the languages coders specialize in are (C, C#, C++, Java, Visual Basic). The language we our going to focus on in this book is **C++**. **C++** is a general purpose programming language.

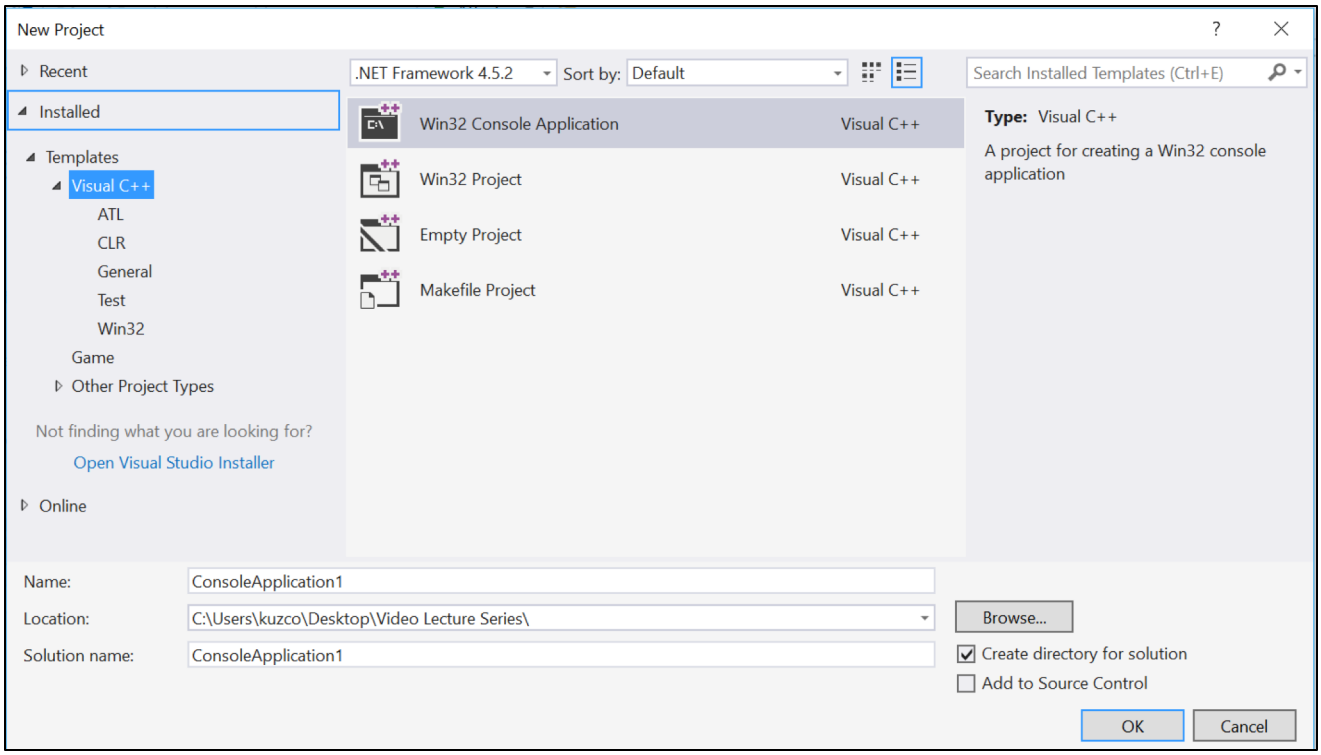
Introduction To Visual Studio 2017 | How to make a console application.

A **Console application** is a application that compiles to a console window. A console application allows a user to input and receive output from a console Window. A console application allows us to start by creating a simple application. We can also build way more complex applications as well.

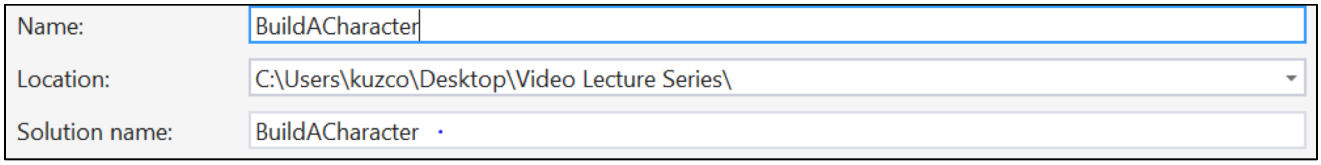
The first step we need to take is to find **Visual Studio 2017** and open it. Upon opening it you will get a screen that looks like the image below.



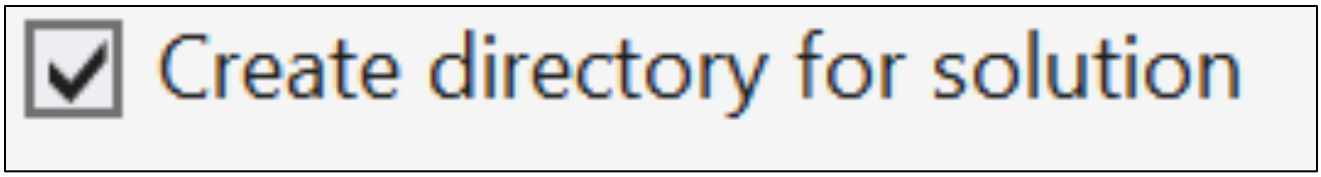
We then select **File > New > Project.** You will open a window that looks like the image below. This is our **New Project Window.** In the new project window we select a Win32 Console Application, Which will allow us to create a Console Application. Win32 Project allows us to create a windows form application. For the time being we are going to select Win32 Console Application.



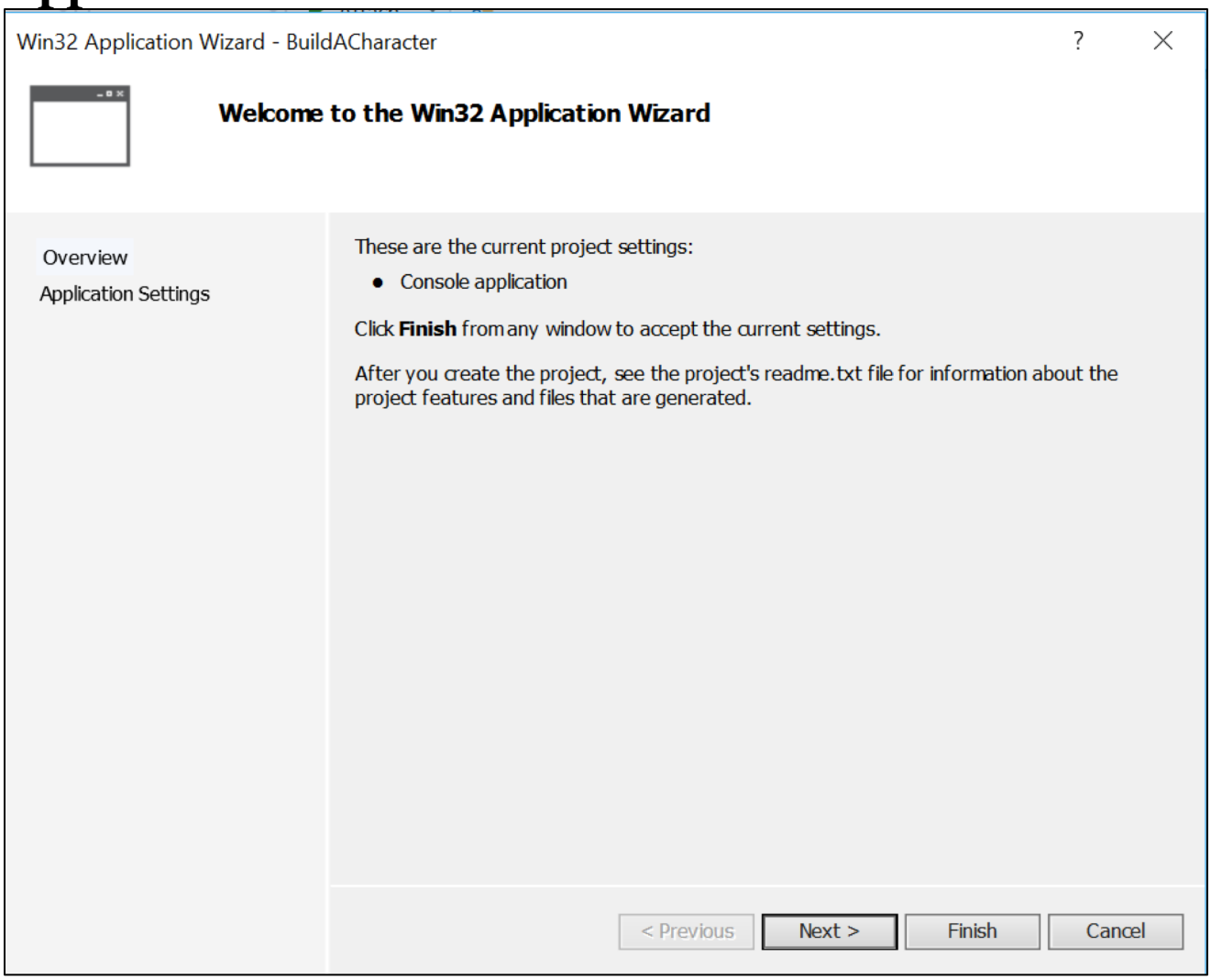
Next we are going to select **Win32 Console Application**. We are going to name the application **BuildACharacter.** The **Solution Name** will also default to **BuildACharacter**. If you would like your solution name to be different you can change it. For the time being I am just going to leave mine as is. Check the image below for more details.



There is also a tiny check box that states : **Create directory for solution.** This will create a new folder for the solution. I am going to leave it checked for the time being.

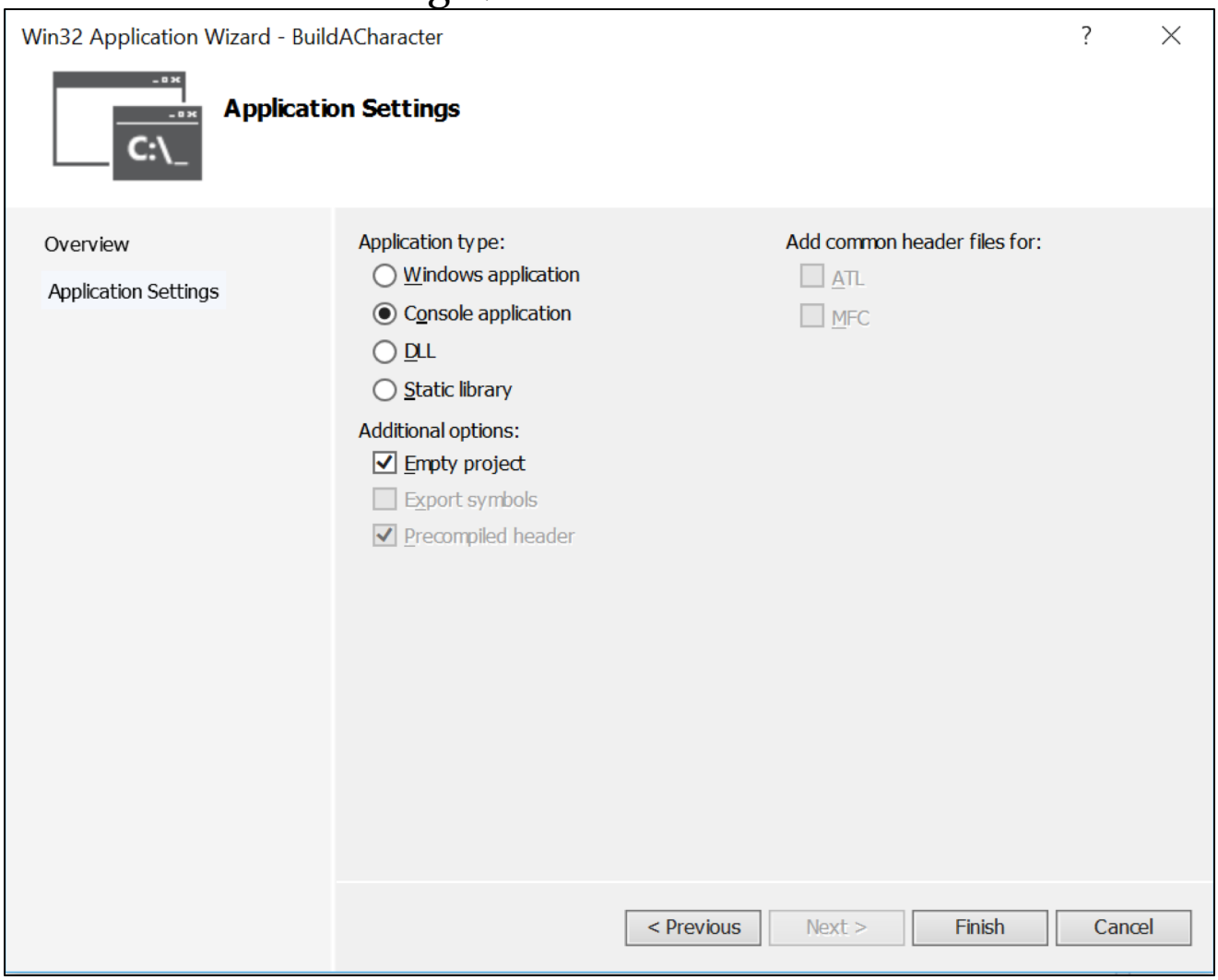


We will then select **OK**. Which will bring up the **Win32 Application Wizard**.

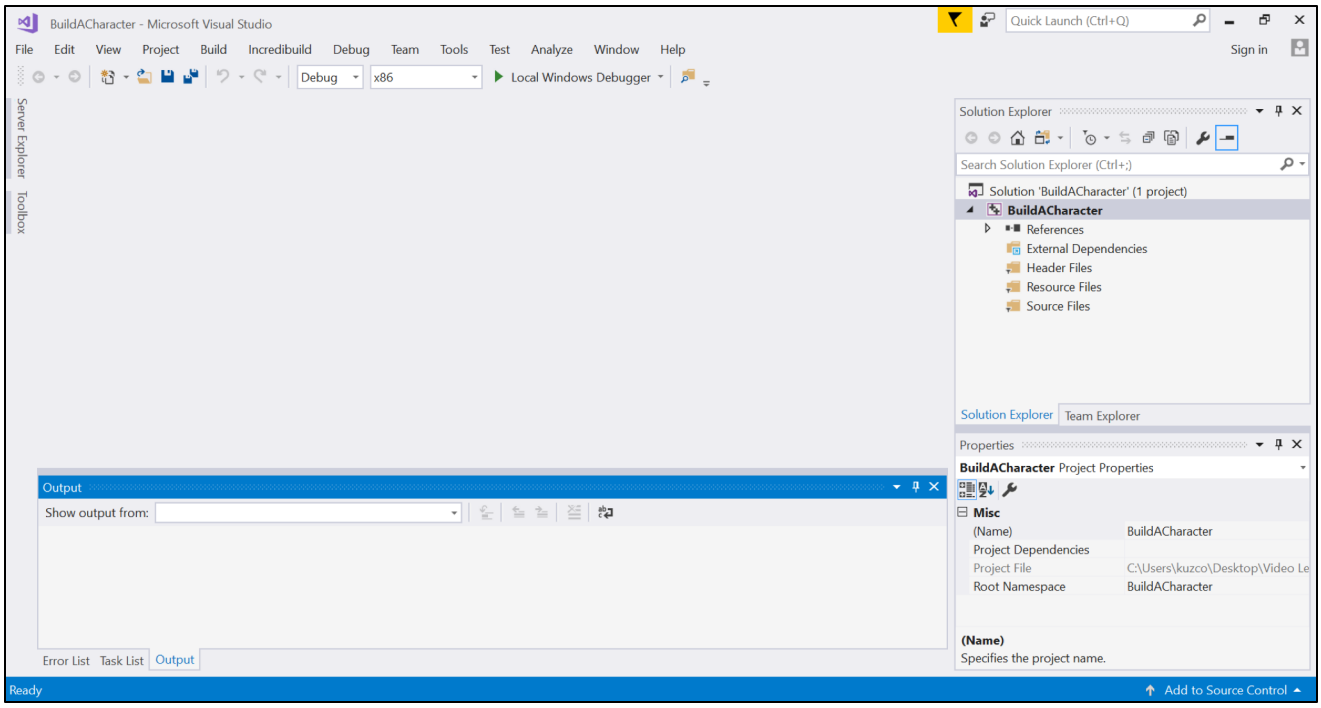


Go ahead and select **Next.** Which will bring us to our **Application Settings**, We want to Find the **Option Header** called **Application Types** and make sure **Console Application** is selected. Also under the **Option Header** called **Additional options** select **Empty Project.** If you

have been following along your **Application Settings** should match the image below.

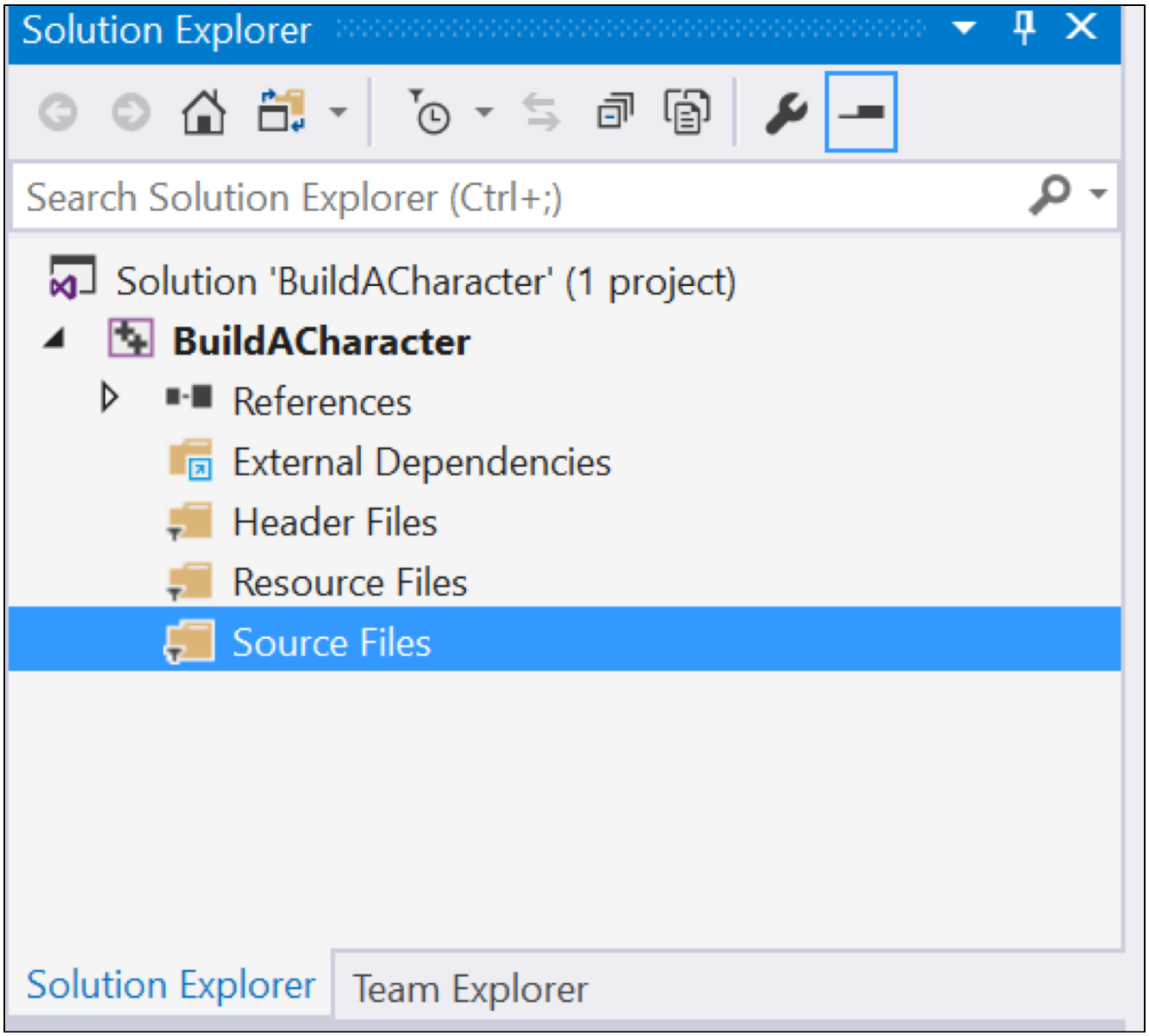


Finally we will select **Finish**. We have created an empty project and in the next step we will be creating a new **.cpp** file. The image below is what our new empty project should look like.

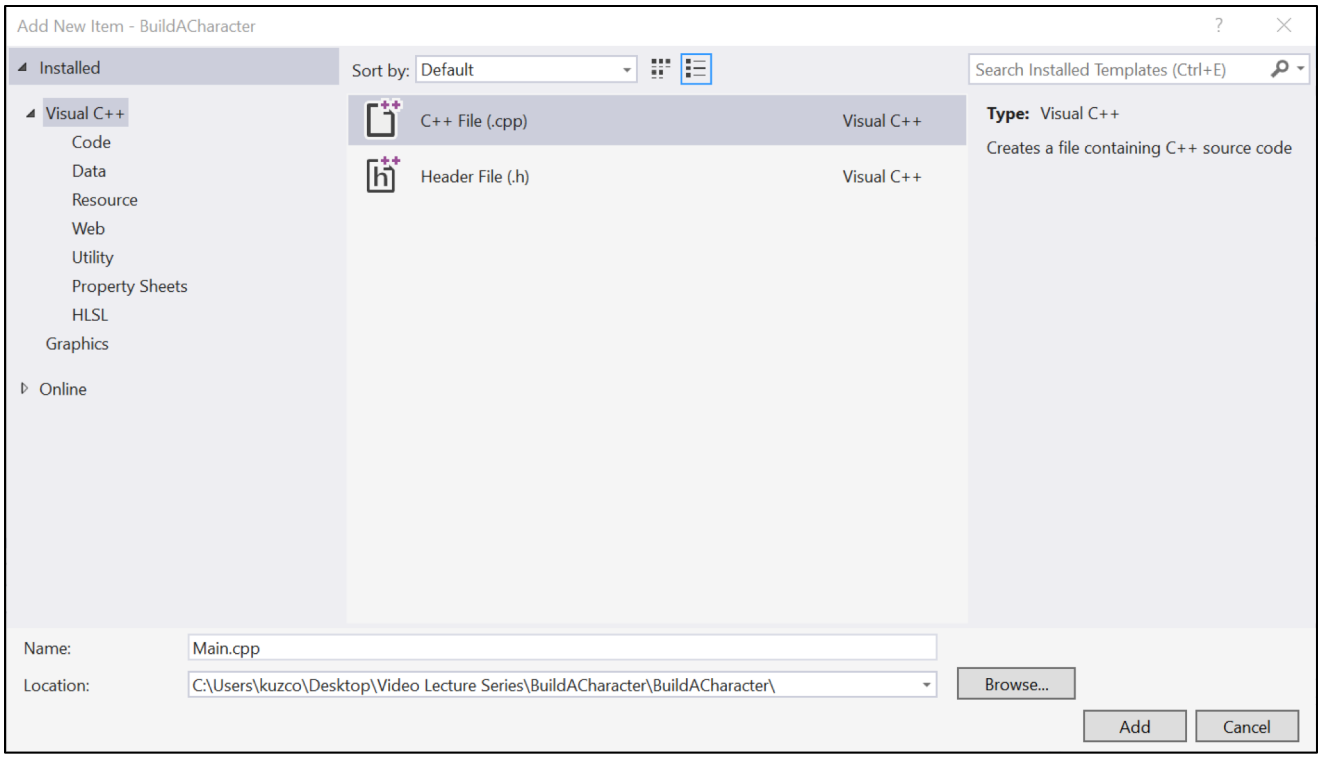


Introduction To Visual Studio 2017 | Create a new .cpp file called main.

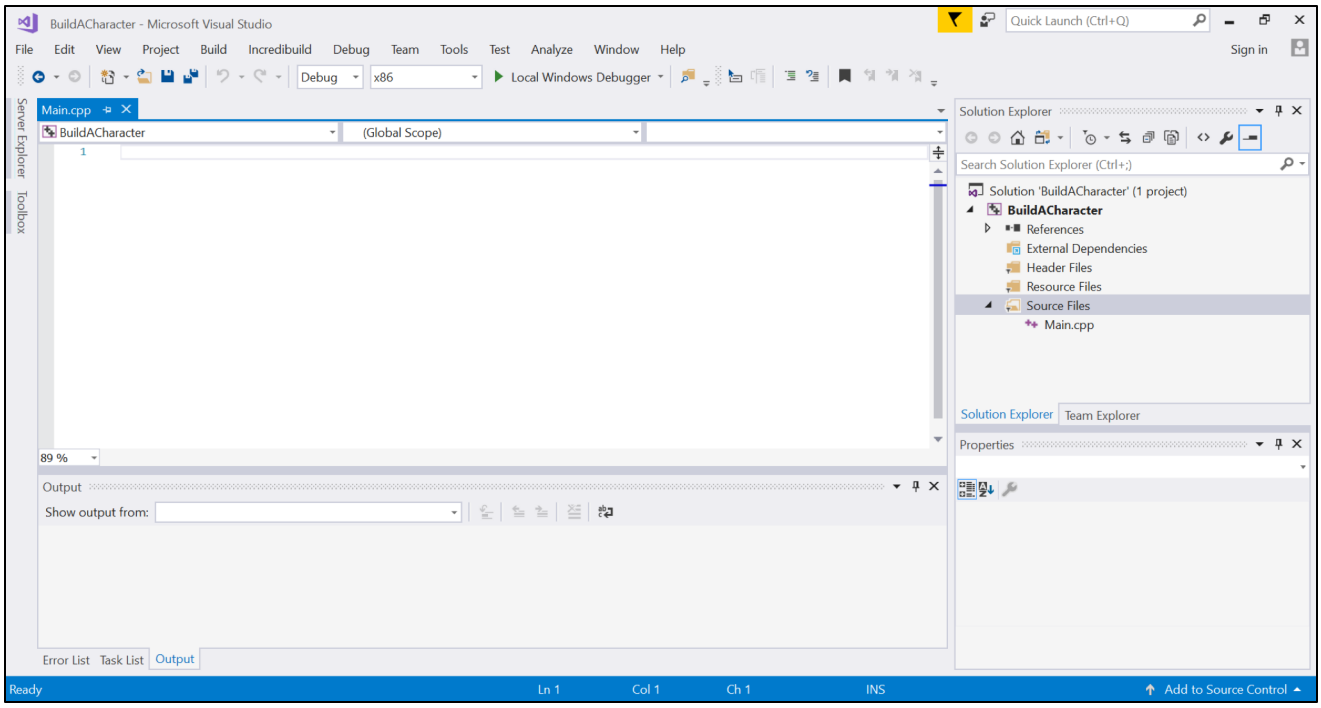
The next step is to create a new .cpp file called main. We need to look for a window called **Solution Explorer**. The Solution Explorer holds all are project files in a neat and organized manner. If you do not see the Solution Explorer please go to the top tool bar and select **View -> Solution Explorer.** An alternative way to find your solution explorer is to press **CTRL+ALT+L** for a keyboard shortcut.



There are a few different ways to create a .cpp file. The first option you have is to select the **Source Files** folder from the solution explorer and **right click Select Add -> New Item.** The next way is a keyboard shortcut select the **Source Files** folder and press **CTRL+SHIFT+A** to add a new item. The final way i am going to cover is in top toolbar we need to select **Project -> Add new items...** This will bring us to a window shown in the image below.



We need to select **C++ File,** and the **Name** needs to be **Main**.cpp. Then we will select **Add.** We have now added a Main.cpp file to our project. You will notice there is no code yet. All we have is a number detailing our line number we are on. We are now ready to start coding.



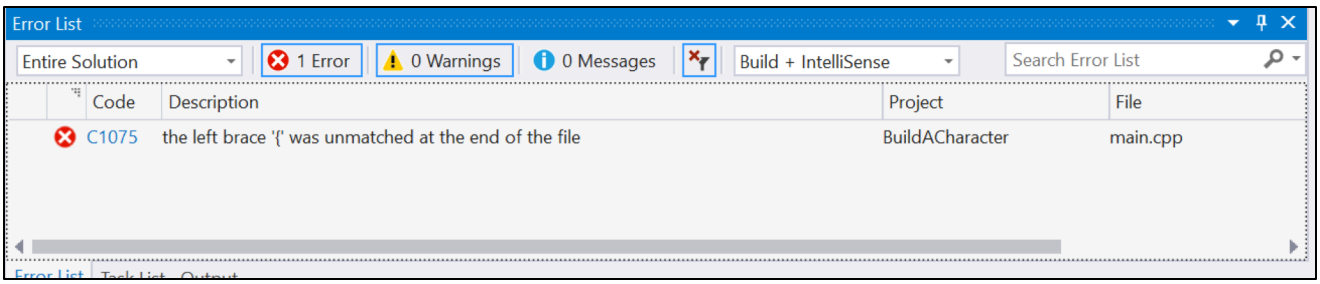
Introduction To C++ | Introduction to our first .cpp file.

At the moment we have no code. It is just a blank white screen. What we are going to do next is create our **Main** function. This is where our code will begin processing. So the first thing we need to do is write:

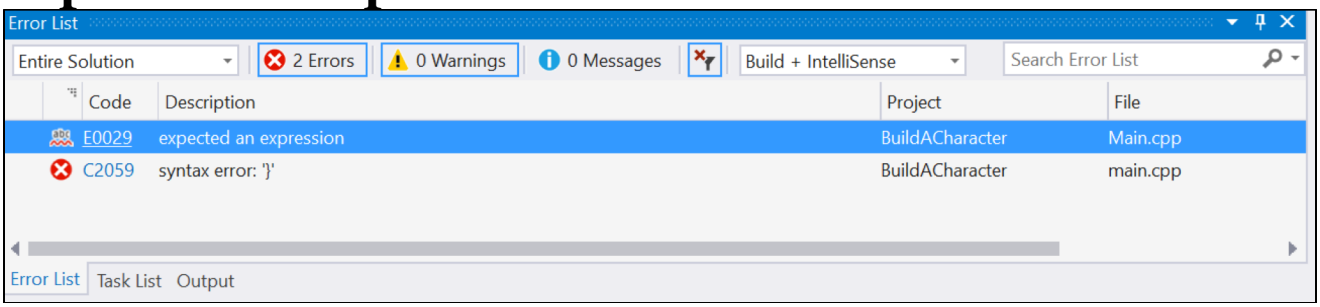
```
void main()
{
    return;
}
```

This will allow are program to start. If you compile your program now it should open a console window that does nothing. To compile your program you can press **F5** or you can go select the **green play button** that states **Local Windows Debugger.** If you have got this far with no errors. Congratulations you have created your first C++ program.

Welcome to the beautiful world of programming. Lets go over the code we have written. The first line states **void main()**. This means we are creating a **function** that has no return value. The main function is in every c++ program it gives our code an entry point. It may be the single most important function in c++. The next line is a **{**. The curly braces are very important it tells us what code we can store inside the **function.** Always remember when ever you have a open curly **{** you must close it off with a closing curly brace **}**. Otherwise you will get a nasty compilation error that states: **C1075: the left brace '{' was unmatched at the end of the file.**



The last line of this function is **return;**. There are two very important parts here. Lets dissect this a little further. Lets start of with **return** this tells us what we are returning and we should add these in every function we create even if we are not returning anything. The second is the **;** semi-colon almost every line of code will have a semi-colon if we forget to add these we get a error message that looks like. **E0029: expected an expression.**



Introduction To C++ | #include <>/"" Lets talk about Preprocessor Directives.

The next line of code we are going to add is at the top of the code we have already written enter a new line. **#include <iostream>**.

```
#include <iostream>
void main()
{
    return;
}
```

#include is a **preprocessor directive**. What is a preprocessor directive? Preprocessor directives are used to make our code easier to compile and change. For instance we can insert other code into our file by adding the **#include** line. <> is telling us that we are searching for a file along the compilers options. There is another form you can use **#include "local.h"**. The **""** tells us we are searching our local files for a file of type **.h**.

Introduction To C++ | What is IOSTREAM?

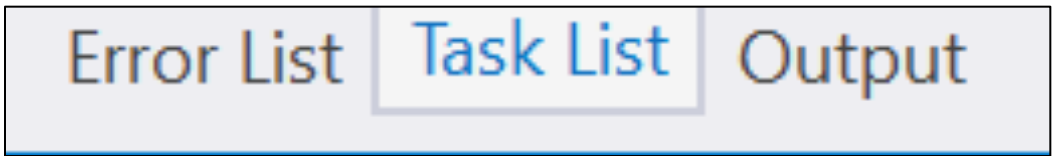
You might have in our #include<**iostream**> we added iostream. This will allow us to use **Functions** from the included file to **read,** and **write** to our console window. A few of the important functions we will be using from the file, **cout, cin, getline**. We will be covering these in more detail later on.

Introduction To C++ | Our first comment: // /**/

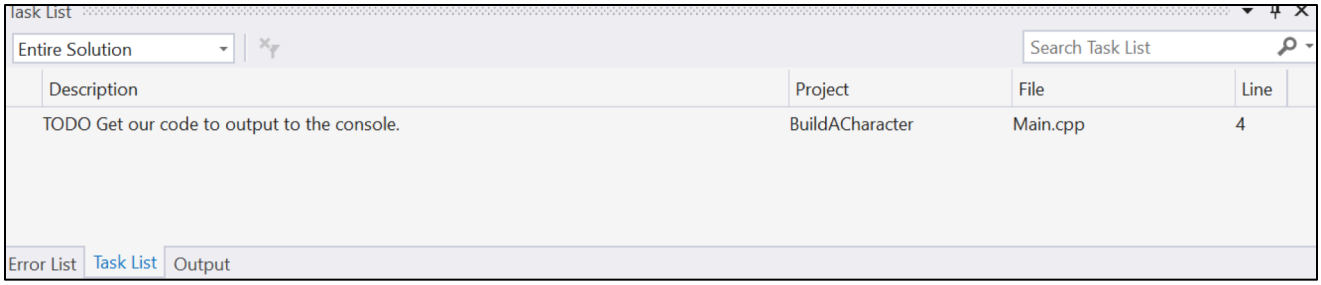
A comment allows us to keep our code nice and tidy. It will help us remember what a piece of code is doing. It will also tell other people who do not know our code what the code is doing. Comment's can either use // for a single line comment or /**/ for a multi line comment. Commenting is very important, and will be used a lot in this course. Let's write our first comment please add this code inside your **main** function. **// TODO Get our code to output to the console.**

```
#include <iostream>
void main()
{
    // TODO Get our code to output to the console.
    return;
}
```

TODO will allow us to use the task list function inside of visual studio and remind us what we are working on or still need to finish. To access the task list you need to go to the **Top Toolbar > view > task list.** You can also access it by using **CTRL+\,T** or near the bottom of the screen you may have tab that states task list.



The task list will look like:



Introduction To C++ | #using namespace: What is a namespace?

#using is another preprocessor directive that we may see used a lot. It can be used in a few different ways. **#using namespace name; #using <file>.** We are only going to cover the first way of using it. The first is using a **Namespace**. A namespace is region that is declared, it provides a scope for useful expressions, and allows organization of code. There are two ways to access a namespace. The first we will not really use, and the second i will use a lot. The first way is to use **#using namespace name;**, and the second looks like **name::thingwereusing**. The first will allow our code to use a namespace throughout our code without writing a single identifier(**name::thingwereusing**) every time we want to use the namespace. The problem with using **#using namespace name;** is we start having the possibility of have namespace clashes. Instead we are going to keep our code elegant, by just using a single identifier every time we wish to use a namespace.

Introduction To C++ | #define and using

The next few lines we are going to cover are #define and using. Since later on in the course we are going to be covering unreal engine 4. We are going to go ahead and use a preprocessor directive of **#define,** and another line of code using.

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;

void main()
{
    // TODO Get our code to output to the console.
    return;
}
```

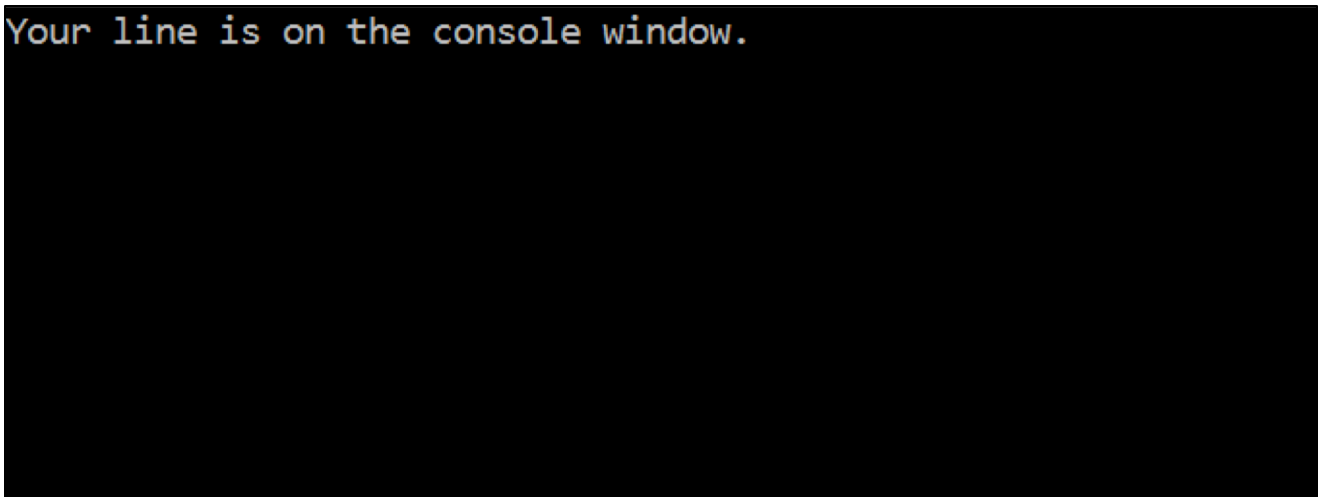
From the image above you can tell we added a few new lines of code. The first is **#include <string>.** This will allow us to use strings. A **string** is a line of letters. We will be covering this more later on when we talk about variables. For the time being we just need to know there is a file that has been included called string. The next line states **#define cout std::cout**. #define is another preprocessor directive this allows us to define a piece of code for the class that will allow us to use the name instead of typing out the whole sentence every time. In the case of cout we are shortening std::cout to cout. This will make our life a lot easier and allow us to code faster. The last two lines **using int32 = int;** and **using FText = std::string** will help us learn unreal engine better when we get there. using name = value; is setting a name that will allow us to use the value every time we call it.

Introduction To C++ | Writing our first console output.

We are finally going to write our first console window output. The lines we are going to need to add are:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;

void main()
{
    // TODO Get our code to output to the console.
    cout << "Your line is on the console window.\n";
    return;
}
```

The new line of code we have added is **cout << "Your line is on the console window.\n";** Let's dissect these lines a bit further. **cout** is a line we are using from the <iostream>. cout is allowing us to put the value on the screen. We follow it up by using a **insertion operator(<<)** that will take our string value and output it to the screen. The string value was set up by the **("string")** quotes. **\n** allows us to add a new line to the console. Compiling are program now will result in:



Now we can remove the comment on our **TODO** list by changing it to:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;

void main()
{
    // This is outputting to the console.
    cout << "Your line is on the console window.\n";
    // TODO introduction to variables
    return;
}
```

The next step we are going to take is introducing variables.

Introduction To C++ | What is a variable?

A **variable** is stored in a part of the system memory. There are quite a few different types that allow us to store different values that the compiler offers by default. We can

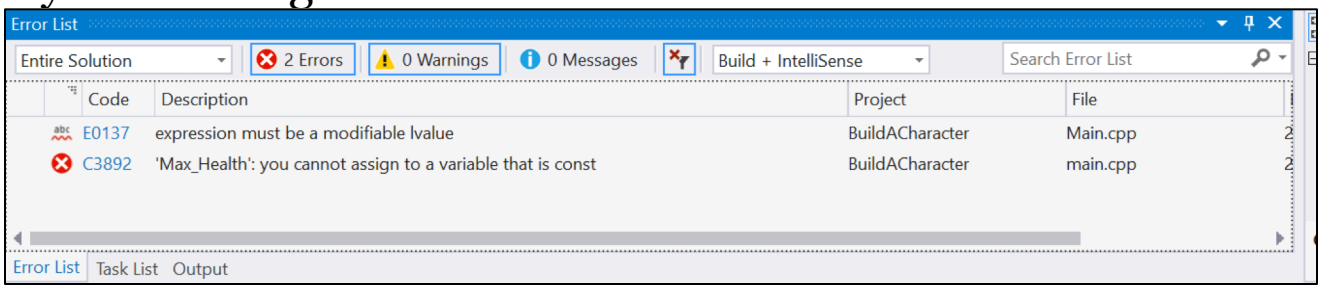
even create our own variables. Setting up our own variables is a ways off at the moment, so let us a take a look at the syntax of what a variable looks like <**type**> **name**;, this will allow us to create a variable. There are a few different types of variables in the image below i cover just a few of them:

```
    // TODO introduction to variables
    std::string newString = "This is a string
variable."; // literal assignment
    char character = 'c'; //This is a char variable, it is
used to store characters.
    int intNumber = 1; //This is a int, it store whole
numbers.
    float floatNumber = 0.0f; //This is a float, it stores
floating point numbers.
    bool bIsTrue = true; //A bool stores true or false.
    double largerFloat; //Non-literal assignment.
```

Lets break down one of these variables to understand it better. **int intNumber = 1;** The first thing we declare is the type of variable we have. It is a type of **int**, a int stands for integer and holds whole numbers between[-2,147,483,647 & 2,147,483,647]. We then set the name of our variable which we will call **intNumber**. Finally we give it a **literal assignment** to the value of 1. A **literal assignment** means we are automatically setting the value of **intNumber**. However we can create a non literal assignment like **double largerFloat;** the value will be something that we don't know, usually we will reassign the value later. For good practice i recommend setting all of your variables to literal assignments. Just so you never call something that doesn't have a value that you don't know. We can also set our variable to a type of **const**. A **const** variable means the value will never change. Take a look at the image below to see how you would go about writing a const variable:

```
//set max health to 100
const int Max_Health = 100;
```

In this line of code **const int Max_Health = 100;** we are setting Max_Health to a constant value of 100. The value will never change. We can not even reassign it. However if we try to we will get a error that looks like this:



Introduction To C++ | Type Casting

We can also cast different types. Casting is changing one type of variable to another type. Let me show you how it looks in code:

```
float floatNumber = 0.0f;
//cast from a float to a int type.
int castFromFloat = (int)floatNumber; //castFromFloat value will be 1
```

We create a float variable and set it to 1.5f. We then decide take the float variable and **cast** it to a new variable type of int. The int will only be able to store the whole number. So the value of the int is 1. We just casted a float to a int.

Introduction To C++ | Operators + - / *

A operator allows us to do addition, subtraction, multiplication, division. These are very powerful and are used in programming quite often. They follow the order of operations.
Lets take a look at some examples in code:

```
int var = 0;
int var2 = 1;
cout << var * var2; // outputs 0
cout << var + var2; // outputs 1
cout << var / var2; // outputs 0
cout << var2 - var; // outputs 1
```

Lets break down one of these line. We create a integer with the name of var and set its value to 0. We create another integer of with the name var2 and set its value to 1. Then we take the variable var and multiply it by var2. The result we should have is that the output on the screen will be 0. Because 0 * 1 is equal to zero.

Introduction To C++ | Writing our main menu code.

With the knowledge we have obtained by getting this far we should be able to set out and create a main menu for our a character creation system for a rpg game. Lets start by modifying our main functions code to look like the code below:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;
```

```
void main()
{
    // TODO setup our main menu
    // TODO get a value from the user for our menu.
    return;
}
```

Starting from here we are going to decide what we need **TODO** next. We start by writing out two comment lines telling us what we would like to do. The first thing we would like to do is to create a main menu. Then we would like to get user input to tell us what we our program should do. The first thing we should probably do is create a variable **FText getUserInput = "";**. Quick side note: quickUserInput is a type of camel casing. The next thing we would like to do is to start by getting our console window to display our menu. So we already learn there is line of code to do that **cout**. Our code will look like:

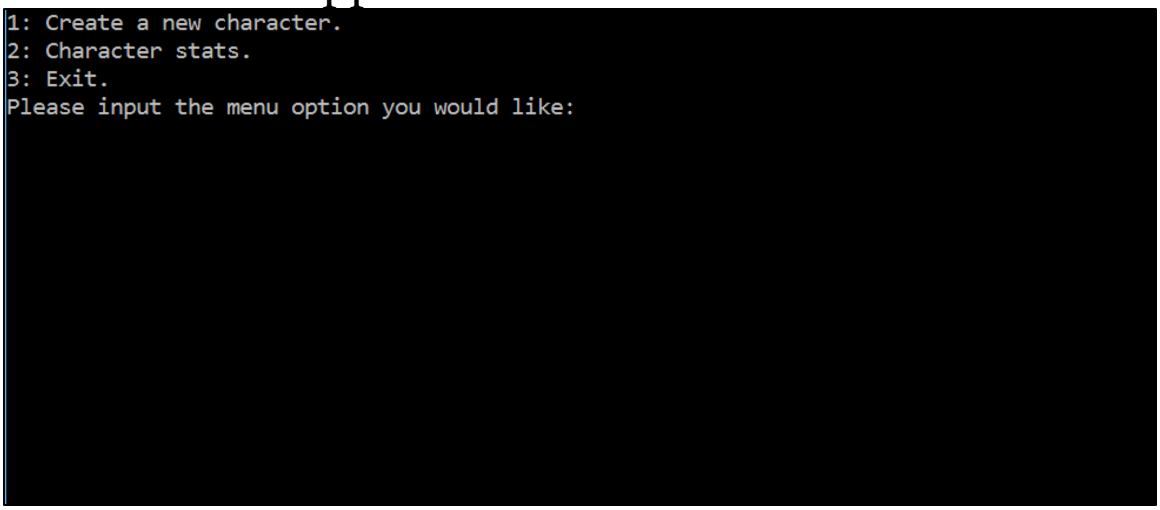
```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;

void main()
{
    // TODO setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    // TODO get a value from the user for our menu.
    return;
}
```

If we compile our code now we will get a few lines of text. But we can not input what the option is we would like to do.

We will call a function std::getline(). This will allow us to get input from the console. To create this function we need to write a line of code like this. **std::getline(std::cin, getUserInput);** Lets dissect this code. First off we call a function called getline that belongs in the std namespace. We pass in two **parameters**. The first a input type call **cin**. **cin** takes the input from the console, through an **(>>) extraction operator** and sets it to a the variable of **getUserInput**. **cin** can also be written out like this **std::cin >> variable;**. However we are going to be using **std::getline()**. The code when finished will look like this:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;

void main()
{
    FText getUserInput = "";
    // TODO setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    // TODO get a value from the user for our menu.
    cout << "Please input the main menu option you would
like: ";
    std::getline(std::cin, getUserInput);
    return;
}
```

If we run this application now it will look like this:



You will be able to put a option in but the program will close. For now this what we are aiming for.

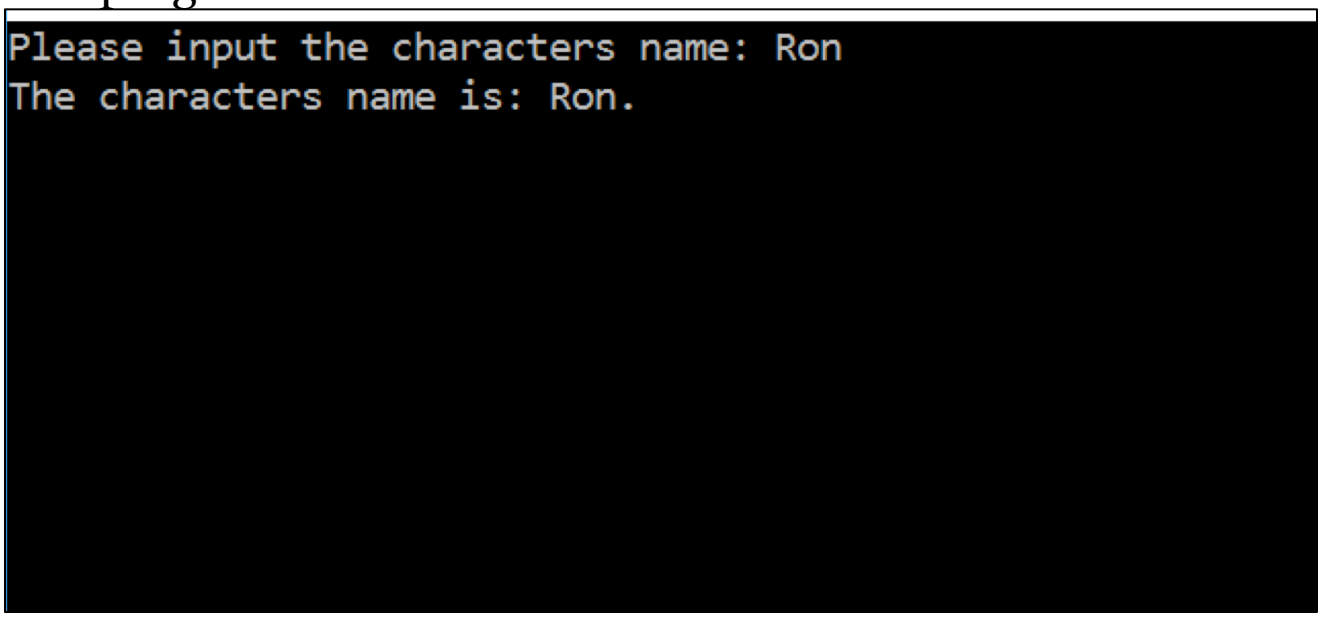
Introduction To C++ | Summary

In this section we learned how to create a project and write our first program. What preprocessor directives are and how we can put them to use. What namespaces, iostream and macros are. We covered what a variable was and how we declare them. We also covered operators and why they are useful. We also began writing our character creation system for a rpg menu. However we found out there is so much more to learn. If you followed along in this section you should be ready to move on to the next section where we will begin covering what a function is, what logic does, and how a switch works.

Introduction To C++ |Assignments

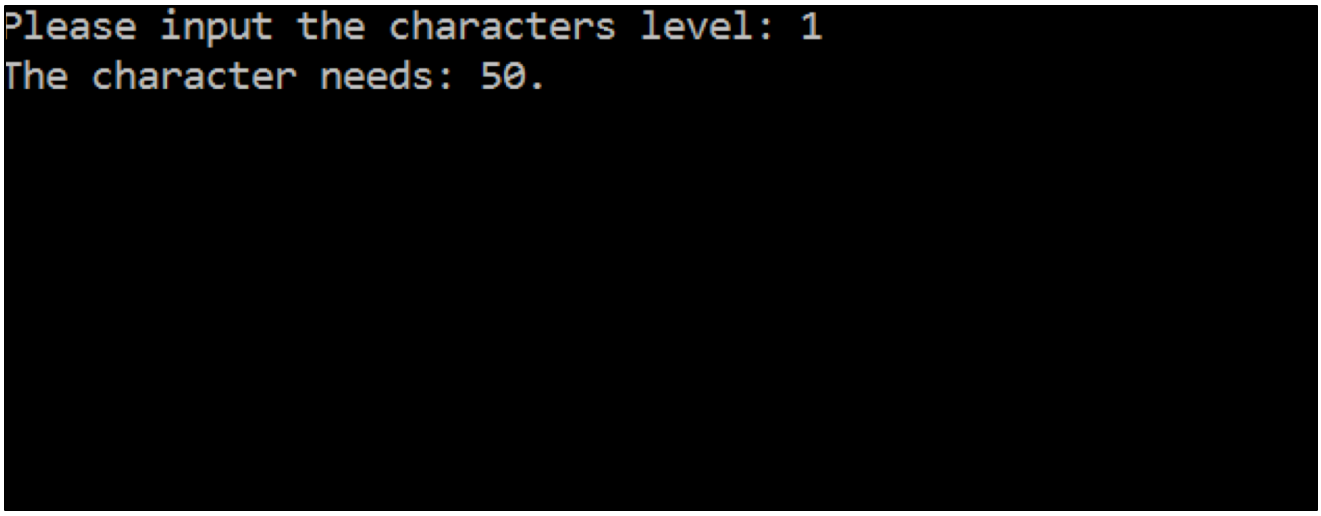
1: Get a characters name.

In this assignment, I want you to ask the user for a character's name. Then output to the user that "The characters name is: <character name>".
What this covers: Getting input from the console. Creating variables. Outputting to the console.
You program should look like:



2: Simple experience calculator:

In this assignment we want to calculate how much experience is needed at each level. The program will ask the user what level the character is. The user puts in a level and then the program outputs how much experience is required to level up. The program output should look like: HINT: use std::cin.
covers: input, output, creating variables, operators.



3: Define a macro.
Define a macro called cin that replaces std::cin.

Chapter 2

Functions, Switches, and While.

All About Functions |Introduction

This section covers functions, switches, and while loops. We will learn how to use a function and what they are useful for. We will then cover creating a switch and the logic that goes behind it. Finally we will finish up with learning about a while loop and how we better use them.

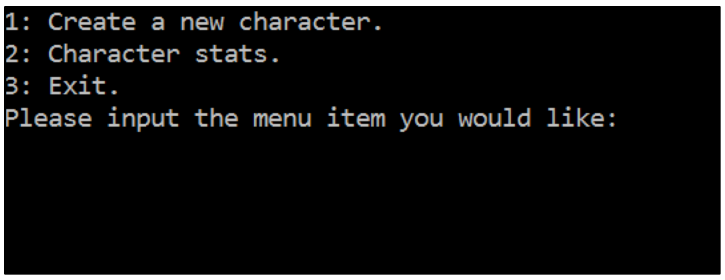
All About Functions |Functions

A **Function** is a section of code that allows you to perform a certain task. It helps shorten the need for rewriting code. We can pass in **Parameters** or even receive **values** back. Let's take a look at what a function would look like:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;
void DisplayMenu()
{
    FText getUserInput = "";
    // TODO setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    // TODO get a value from the user for our menu.
    cout << "Please input the main menu option you would
like: ";
    std::getline(std::cin, getUserInput);
    return;
}

void main()
{
    DisplayMenu();
    return;
}
```

We create a **Function** by first giving <**TYPE**> we are going to return and finally the **name** we are going to use followed by a **()** which will specify whether or not we are going to pass parameters through to the **Function**. In the above example we created a **Function** called **DisplayMenu** it is of type **void** meaning we are not returning any values. There are also no parameters being defined. If we look in the

main() function we **Call** the **DisplayMenu()**; function. You may have noticed we covered a new line **std::endl**. It is basically say we want to create a new line. Which will allow us to have a output in our console window that looks like this:

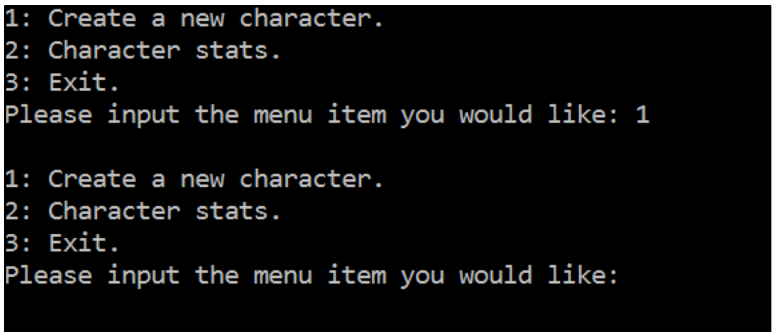


Now lets say we want this to repeat its self twice or even more times. All we have to do is **call** the function **DisplayMenu();** in the **main()** function as many times as you would like. For now i am going to have it ask us twice.

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;
void DisplayMenu()
{
    FText getUserInput = "";
    // TODO setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    // TODO get a value from the user for our menu.
    cout << "Please input the main menu option you would
like: ";
    std::getline(std::cin, getUserInput);
    return;
}

void main()
```

```
{
    DisplayMenu();
    DisplayMenu();
    return;
}
```

We will now receive a output that displays our menu twice. Imagine if we had to write all that code out twice how annoying would that be. The time to write out all that code would take quite a bit of time. We just saved our selves from writing out the display menu every time we want to see it. Now all we have to do is **call** the function. As you can tell functions are quite powerful and this is just the beginning. The display output of the program above:



All About Functions |Return Value

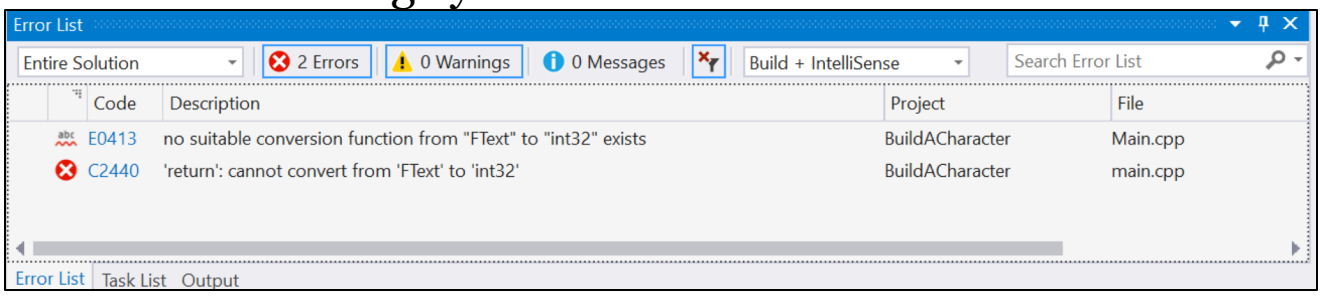
The next step in this program we are going to take is actually returning a value, so that we can take it and pass it into another function. So how do we actually get our function to return a value that we can use. Lets take a look at the example code:

```
#include <iostream>
#include <string>
#define cout std::cout
```

```
using int32 = int;
using FText = std::string;
int32 DisplayMenu() //we will return a integer value.
{
    int32 getUserInput = 0;
    //setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    //get a value from the user for our menu.
    cout << "Please input the main menu option you would
like: ";
    std::cin >> getUserInput;
    cout << std::endl;
    return getUserInput;
}

void main()
{
    int32 menuItem = DisplayMenu();
    return;
}
```

Our function is now changed from returning a type of **void** to a type of **int32**. Remember **int32** is just a macro for **int**. If you look at the bottom of **DisplayMenu()** you will notice we are returning a variable of type **int32** called **getUserInput**. If our user inputs the value of **1** then we will set the value of **getUserInput** to **1**. If we look inside the **Main()** function we will notice **int32 menuItem = DisplayMenu();** the value of **menuItem** will then be set to **1**. The last thing we need to know about returning values from a function is that the value being returned **has** to be the same type that the function is using. So if we have a type of function **int32**, it has to return a **int32**. A couple more examples would be **float** return has to be **float, FText** has to be

FText, basically just remember <**Type**> has to be <**Type**>. There error message you would receive would look like:



All About Functions |Passing Parameters

A **parameter** is a variable we can pass values into. Let's take a look at what a function with a parameter looks like.

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;
int32 DisplayMenu(); //we will return a integer value.
void ChoiceMenuOption(int32 choice);
void main()
{
    ChoiceMenuOption(DisplayMenu());
    return;
}

int32 DisplayMenu()
{
    int32 getUserInput = 0;
    //setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    //get a value from the user for our menu.
    cout << "Please input the main menu option you would
like: ";
```

```
    std::cin >> getUserInput;
    cout << std::endl;
    return getUserInput;
}

void ChoiceMenuOption(int32 choice)
{
    cout << choice;
    return;
}
```

The first thing we may notice is on line 8. We have changed the **int32 DisplayMenu()**. We have moved the main **DisplayMenu()** function underneath the **main()** function. The code on line 8 **int32 DisplayMenu();** Is telling our main function that there is a function called **DisplayMenu** but it is underneath the **Main()** function and will need to look for it to find out what the function does. This is called a **function prototype**. All **function prototypes** must have a **<return type>, name, parameters,** and finally a (**;**)**semicolon** at the end. Now lets look at our first parameter. We created a **void** function called **ChooseMenuOption(int32 choice)**. **int32 choice** is our parameter. Let's dissect how the above program works. First when we run the program we are going to go to the **main()** function. We will look inside the **main()** function and see the function **ChooseMenuOption().** The program will then check the parameters, and see a function **DisplayMenu()**. We will first Initiate **DisplayMenu()** then **DisplayMenu()** will return a **int32** variable. We will then enter the function **ChooseMenuOption()**. With a value of the returned **DisplayMenu()**. Following that we will print to the screen

the value of **DisplayMenu()** by using the parameter variable **choice**. The output will look like below:

```
1: Create a new character.
2: Character stats.
3: Exit.
Please input the menu item you would like: 1
1
```

We can have more than one parameter passed in. By creating a function like **<type> NAME(<type> param1, <type> param2, <type> param3...){}** You can use as many parameters as you would like. We can also give parameters default values. A function with default values would like **<type> NAME(<type> param1 = value, <type> param2 = value, <type> param3 = value);** this will allow us to use a function even if we choose not to pass values into our parameters. The next line of example code will be looking at how we would do this in code:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;
int32 DisplayMenu(); //we will return a
integer value.
void ChoiceMenuOption(int32 choice = 0, FText
text = "Your choice was: ");
void main()
{
    ChoiceMenuOption(DisplayMenu());
    return;
```

```
}

int32 DisplayMenu()
{
    int32 getUserInput = 0;
    //setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    //get a value from the user for our menu.
    cout << "Please input the main menu option
you would like: ";
    std::cin >> getUserInput;
    cout << std::endl;
    return getUserInput;
}

void ChoiceMenuOption(int32 choice, FText
text)
{
    cout << text << choice;
    return;
}
```

Our output of this code would result in:

```
1: Create a new character.
2: Character stats.
3: Exit.
Please input the menu item you would like: 1
Your choice was: 1
```

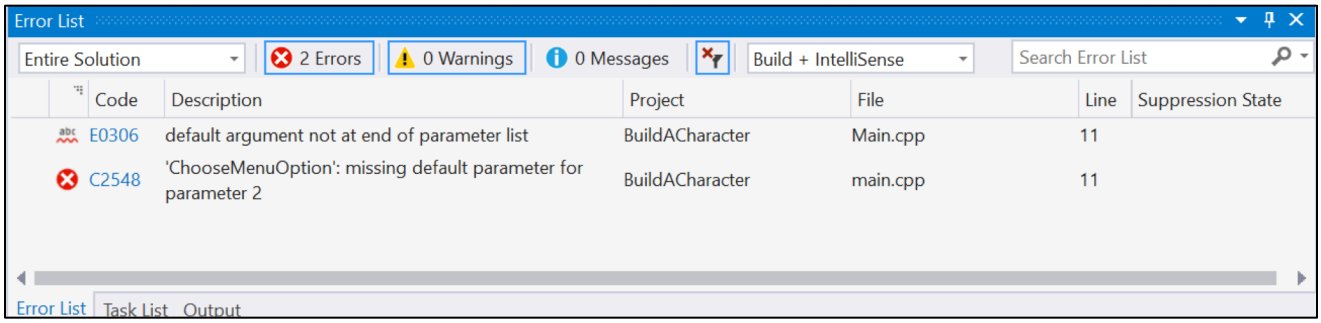
All About Functions |Function Overloading

Function overloading is when you have more than one of the same functions each with different parameters. Lets take a look at some example code:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;
int32 DisplayMenu(); //we will return a integer value.
void ChoiceMenuOption(int32 choice = 0); //The first
overload we will only pass choice.
void ChoiceMenuOption(FText text, int32 choice = 0); //The
second overload we add a parameter for text.
void main()
{
    ChoiceMenuOption(DisplayMenu());
    ChoiceMenuOption("This is a overload: ",
DisplayMenu());
    return;
}

int32 DisplayMenu()
{
    int32 getUserInput = 0;
    //setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    //get a value from the user for our menu.
    cout << "Please input the main menu option you would
like: ";
    std::cin >> getUserInput;
    cout << std::endl;
    return getUserInput;
}
```

```
void ChoiceMenuOption(FText text, int32 choice)
{
     cout << text << choice;
     return;
}

void ChoiceMenuOption(int32 choice)
{
     cout << choice;
     return;
}
```

We now have two of the **ChooseMenuOption()** functions, the only difference is that they have different parameters. They have different functions, **ChooseMenuOption(FText text, int32 choice)** allows you to add text. While the other parameter just allows for the choice. When setting up function overloading we start by creating the default function **<type> NAME(<type>param1),** Then we create a **function overload** by **<type> NAME(<type> param2, <type> param1)**. It is very important that the first parameter ends on the function overload. Otherwise you can get a error that looks like this:



Switch |Introduction

We are ending are discussion on functions. Now it is time to switch, and talk about **switch** statements and how they can help our code. So lets take a look at example code:

```
#include <iostream>
#include <string>
#define cout std::cout
using int32 = int;
using FText = std::string;
int32 DisplayMenu(); //we will return a integer value.
void ChoiceMenuOption(int32 choice = 0);
void main()
{
    ChoiceMenuOption(DisplayMenu());
    return;
}

int32 DisplayMenu()
{
    int32 getUserInput = 0;
    //setup our main menu
    cout << "1: Create a new character.\n";
    cout << "2: Character stats.\n";
    cout << "3: Exit.\n";
    //get a value from the user for our menu.
    cout << "Please input the main menu option you would
like: ";
    std::cin >> getUserInput;
    cout << std::endl;
    return getUserInput;
}

void ChoiceMenuOption(int32 choice)
{
    switch (choice)
    {
    case 1:
        break;
    case 2:
```

```
            break;
        case 3:
            break;
        default:
            break;
        }
        return;
}
```

A **Switch** takes a **condition** and checks to see if it matches up with any of our **cases**. If it does match up with a **case** we execute the code inside and then **break** out of the switch. However if it does not it will go to a **default** case and then **execute** the code and **break.** The switch statement in our example code does not do much currently. Right now we get our value from our parameter called **choice**. We then pass it into our **switch** statement and check to see if our value of **choice** is equal to any of the cases. If **choice** is equal to a case then we break out of the code. However if it is not we go to our **default** case and then **break** out. **break** statements are usually recommended after every line however there are some cases where you may choose to not include a **break** statement and let you code **fall-through**. The code below expands our **switch** statement a little further and allows our **menu** to finally respond to the user:

```
void ChoiceMenuOption(int32 choice)
{
    switch (choice)
    {
    case 1:
        cout << "Let's get started creating our
character.\n";
        break;
    case 2:
        cout << "These are our characters stats:\n";
```

```
            break;
        case 3:
            cout << "Thank you for playing our game.\n";
            break;
        default:
            cout << "You did not put in a valid answer.\n";
            cout << std::endl;
            break;
        }
        return;
}
```

Our output should now look like this if we choose the first option we will go to a **case** that allows us to output that we are starting a character creation system:

```
1: Create a new character.
2: Character stats.
3: Exit.
Please input the menu item you would like: 1

Let's get started creating our character.
```

If we input the answer 2:

```
1: Create a new character.
2: Character stats.
3: Exit.
Please input the menu item you would like: 2

These are our characters stats:
```

The third option will exit our game. Finally if we get the incorrect answer we will see that we get a message saying that we input a incorrect answer.

While | Introduction

While is a looping statement that allows us to continuously loop in our code while the **condition** is valid. When we refer to a **loop**, we are saying that we want to go over the code multiple times. Let us examine our example code:

```
#include <iostream>
#include <string>
#define cout std::cout
#define cin std::cin
using int32 = int;
using FText = std::string;
int32 DisplayMenu(); //we will return a integer value
bool ChooseMenuOption(int32 choice = 0); // the first
overload we will only pass a choice.
void BeginPlay(); // We begin our game.
void main()
{
    BeginPlay();
    return;
}
```

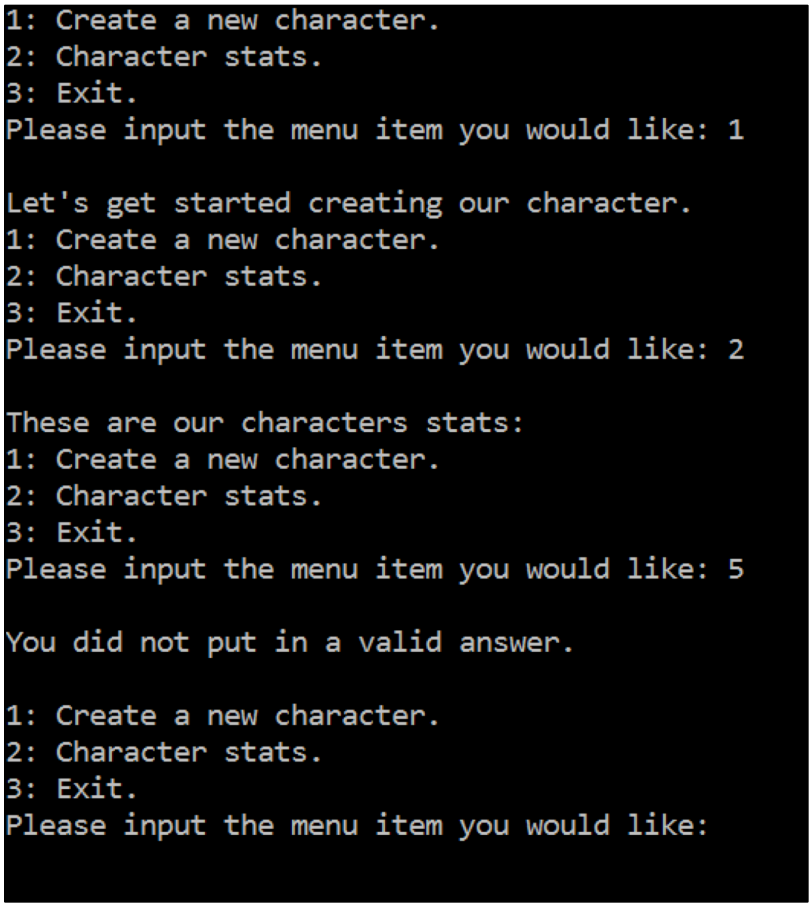
We begin by creating a new function of **BeginPlay()**, We then call **BeginPlay()** in our main function.

```
int32 DisplayMenu()
{
    {
        int32 getUserInput = 0;
        // Setting up our menu.
        cout << "1: Create a new character.\n";
        cout << "2: Character stats.\n";
        cout << "3: Exit.\n";
        cout << "Please input the menu item you would
like: ";
        cin >> getUserInput;
        cout << std::endl;
```

```
        return getUserInput;
    }
}

bool ChooseMenuOption(int32 choice)
{
    switch (choice)
    {
    case 1:
        cout << "Let's get started creating our
character.\n";
        //will loop forever.
        return true;
        break;
    case 2:
        cout << "These are our characters stats:\n";
        //will loop forever.
        return true;
        break;
    case 3:
        cout << "Thank you for playing our game.\n";
        return false;
        break;
    default:
        cout << "You did not put in a valid answer.\n";
        cout << std::endl;
        //will loop till we get a correct input answer.
        return true;
        break;
    }
}

void BeginPlay()
{
    bool bIsPlaying = true;
    while (bIsPlaying)
    {
        bIsPlaying = ChooseMenuOption(DisplayMenu());
```

```
    }
    return;
}
```

In the **BeginPlay()** function we create a bool **bIsPlaying**. Then we start a while statement. The while statement checks to see if **bIsPlaying()**. We set **bIsPlaying** equal to our return value from the **ChooseMenuOption()** function. The while statements common syntax is written like **while(condition is valid.){Do something infinitly amount of times till the condition comes back Invalid.}**. While statements are commonly used when you don't know how many times you want to repeat a certain set of statements. Our output at the current moment will look like this:



While |Do While

The do while is another way a while statement can be written. We start by saying we want to do this while(the condition is valid.). The syntax for this would be: **do{Do this first.}while(our condition is valid)** We start by getting our program to do something then we check to see if our while loop is valid. If it is we go back and do the statements again. We would use a do while loop if we were wanting to do something first and then check to see if the while loop is still valid instead of checking to see if our while loop is valid first. Let us see some example code:

```
void BeginPlay()
{
    bool bIsPlaying = true;
    do
    {
        bIsPlaying = ChooseMenuOption(DisplayMenu());
    } while (bIsPlaying);
    return;
}
```

So first we set **bIsPlaying** equal to our **ChooseMenuOption(DisplayMenu())** return value. So if it returns **false**. We go into the **while** loop and plug in bIsPlaying = false. The while loop will not repeat its self. We will then end the program at the moment. However if it comes back **true** we continue on and reloop back to setting the value of **bIsPlaying**.

Summary |

We learned that a function is a set of code. That is used to make our life easier instead of rewriting code over and over. We learned how to give a function a return value. We also can

set a function to any type we wish. We figured out how to work with function overloads, and also what parameters are. We then progressed our code by learning how to create switch, and what it does. Finally we learned what a while loop and do while loop are, and how we can use them in our code.

Assignments |

1: Level up function.

Create a function that allows the user to level up. You should be able to input how much exp the character has and then the level up command will level up your character till there is no more experience. Hint: There are operands (>=) **Greater than or equal to** or (<=) **Less than or equal to.** These are relational operators. We will be covering them in more depth in the next section.

Covers: Functions, while loops

Output:

```
How much experience would you like to give the character: 500
Your current level is: 4
```

2: Battle Menu.

Create a menu that ask for the players attack. If they put in a incorrect answer. Give them a default output of "you have entered a incorrect input: Please try again." And have them do it again.

covers: switch statements, and while loops.

output:

```
Battle Menu:
1: Attack
2: Run Away
Choose a option: 1

You have attacked the enemy.
Battle Menu:
1: Attack
2: Run Away
Choose a option: 3

That option is not avaible.
Battle Menu:
1: Attack
2: Run Away
Choose a option:
```

3: Use potion function.

Create a function that has the player input the type of potion he is going to use. Then take the potion and add it to the current health or mana.

covers: functions, switches, and while loops.

output:

```
Potion Menu:
1: Health Potion
2: Mana Potion
3: Exit.
Choose a potion: 1

You have selected a health potion.
Your health: 10 Your mana: 0
Potion Menu:
1: Health Potion
2: Mana Potion
3: Exit.
Choose a potion: 2

You have chosen a mana potion.
Your health: 10 Your mana: 10
Potion Menu:
1: Health Potion
2: Mana Potion
3: Exit.
Choose a potion: 4

That option is not avaible.
Potion Menu:
1: Health Potion
2: Mana Potion
3: Exit.
Choose a potion:
```

Chapter 3

If,For, and Arrays.

Introduction | What is logic?

Logic means we ask, can a **valid input** be completed. If the **valid input** can be completed then we execute the circumstance that we would like to complete. However if the **valid input** can not be completed then we can execute a whole different set of circumstances. We use logic everyday in real life. Do i drive my car to work, or do i take the bus? Which route is longer the one on the left or the one on the right? In video games logic is used quite a bit. Is the boss dead? If the boss is did then reward the player. However if the boss is not dead then check to see if the player is alive. If the player is alive check how much more health the boss has to lose to be considered dead. As we can see logic is quite useful. We have actually already used logic in the previous section when we created a **switch** statement. The first part of this section is going to discuss **relational operators**, then we will create a **if**, following that we will dive into **for loops**, and saving the best for last we will discuss what an array consist of.

Introduction | Relational Operators

A **Relational operator** is a type of operator that allows us to **test** or **define** a relationship between two values. An example would be is (5 (>=) 3) this would return a **condition** of true. Because 5 is greater than or equal to 3. Let

us take a look at some of the **relational operators** we have access to.

(>) Greater than: Example is (5 (>) 3) result true
(>=) Greater than or equal to: Example is (5(>=)3) result true
(<) Less than: Example is (5(<)3) result false
(<=) Less than or equal to: Example is (5(<=)3) result false
(==) Equal to: Example is (5 (==) 3) result false
(**!**=) Not equal to: Example is (5(!=)3) result true

There are a bunch of different ways we can use **relational operators**. We can check to see if we have a certain type of data type. We can use it to see if we have a data type that is equal to another type of data. Now we are ready to begin discussing **if** statements and seeing how we use **relation operators**.

If statements | Introduction

A **if** statement is another type of a **conditional** statement. It checks to see if something is equatable. If there is something equatable then a if statement would return back a **true** or **1**. In computers **1** is equal to **true**. Otherwise if it is not equatable it will return back a **false** or **0**. The syntax for a **if** statement looks like **if(some value equals true) then { execute true code } else { execute false code. }** Lets see some example code:

```
#include <iostream>
#define cout std::cout
void main()
{
    if (5>3) // if 5 is greater than 3
    {
```

```
        cout << "Your value is true";
    }
    else
    {
        cout << "Your value is false";
    }
}
```

Inside the **main**() function we can see a **if** statement. Let's dissect what this statement is doing. First we have **if** asking is **5 > 3**. Well 5 is greater than 3 so we get the value **true**. Upon the statement of true. We would write output to the console screen "Your value is true". If the value however would of been **false**. We would have output on the screen "Your value is false". So we might be wondering well what if we want to check more things than a simple if else statement. We have a **nested if statement**. A **nested if statement**, allows us to check and see more than one **if(statement)** value by using **else if(statement)**. Lets take a look at the the syntax for this statement: **if(statement1 > statement2) then { execute first statement code} else if(statement2 > statement1) then { execute second statement code} else { execute else statement code }**. Let us take a look at some example code:

```
#include <iostream>
#define cout std::cout
#define cin std::cin
void main()
{
    bool bContinue = true;
    char c = 'a';
    while (bContinue)
    {
        cout << "Would you like to continue[y/n]: ";
        cin >> c;
```

```
            cout << std::endl;
            if (c == 'y' || c == 'Y')
            {
                cout << "You have chosen to continue!\n";
                cout << std::endl;
                bContinue = true;
            }
            else if(c == 'n' || c == 'N')
            {
                cout << "You have chosen to exit.";
                bContinue = false;
            }
            else
            {
                cout << "You have input a wrong key.\n\n";
                bContinue = true;
            }
        }
}
```

Our output from the above program will look like this:

```
Would you like to continue[y/n]: y

You have chosen to continue!

Would you like to continue[y/n]: l

You have input a wrong key.

Would you like to continue[y/n]:
```

Lets dissect what exactly is going on in the above program. If we go to the **main()** function, we can first see what are creating a **bool** and **char** variable. We go into the

while statement and then we ask the user "would you like to continue[y/n]:". We get the input from the user. Then we enter our **nested if statement**. The first statement is **if** we answer **y (||)or Y**. **(||)** stands for **or** so in this **if** statement we are basically asking **if input == y (||) or input == Y** then do what is executed. We would then execute the code to output "You have chosen to continue!". We would then run through the **while** statement again. The next part of the **nested if statement** is if the user said **n**. We would then write to the screen "You have chosen to exit.". We would then return false to the while loop and exit the program. The last part of our **nest if statement** tells the user "You have input a wrong key." we would reloop through the while loop until they got input the correct answer. **(&&) and** allows us to compare with multiple statements. I did not add this in the code above however i did add **(||) or**. To use **(&&) and** we would write a **if statement** that looks like this: **if(statement one == true && statement two == true) {execute code.}**. The **(&&) operator** only returns **true** if both statements are **true** or both statements are **false**. In all other cases it will return **false**.

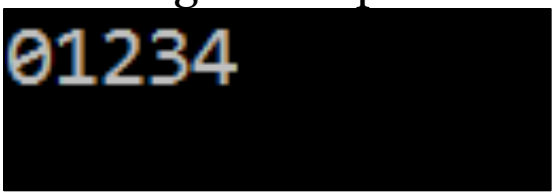
For statements | Introduction

A **for** loop is a control flow statement that will allow us to give a **maximum number** of **iterations** we want to loop around the statement for. The syntax for a **for loop** would be initialized as: **for(some value; value < Maximum Iterations; value++){execute code here}** Let us dissect some example code:

```
#include <iostream>
#define cout std::cout
```

```
void main()
{
	for (int i = 0; i < 5; i++)
	{
		cout << i;
	}
}
```

We will get a output that looks like this:



We look inside the **main()** function and we see a **for loop**. The first part of the **for loop statement** is setting a **counting variable**. We then look at the second statement and see that it is condition. The condition we are check is to see if our **i** variable is < **5**. While thats true we move on to the **third** statement. We add one to **i**. Then we execute the code in the main body. Now we don't alway have to use a counting variable to set up a **for loop**. We can set up a for loop using different types of conditions. As long as the second statement will eventually equal false. Otherwise we will have a loop that runs forever. Just like **nesting if statements**, we can also have nesting loops. Lets take a look at the example below:

```
#include <iostream>
#define cout std::cout
void main()
{
	bool bIsNested = true;
	while (bIsNested)
	{
		cout << "We entered our while loop.\n";
		for (int i = 1; i < 5; i++)
		{
```

```
            cout << "We have entered our for loop: " << i
<< " times.\n";
            cout << std::endl;
            bIsNested = false;
        }
        cout << "Goodbye.\n";
    }
}
```

The output will look this:

```
We entered our while loop.
We have entered our for loop: 1 times.

We have entered our for loop: 2 times.

We have entered our for loop: 3 times.

We have entered our for loop: 4 times.

Goodbye.
```

In the above example we can see that in the **main()** function we enter a while loop with a condition of **true**. We then output to the screen that we have indeed entered the while loop. Next we enter the **for loop** we iterate over our code for our maximum times of iterations. We output that we entered our for loop. And finally we set the while loop condition to **false**. We finally exit our for loop output Goodbye and exit the program. This is a **nested loop** and they can be very useful when working with arrays and other data types.

Arrays| Introduction

A **array** is data type that holds a bunch of values called elements. The array syntax looks like this **<type> Name[Array size];** Let us look at some example code declaring arrays of different types.

```
#include <iostream>
#include <string>
#define cout std::cout
#define string std::string
void main()
{
    //creating arrays
    int intArray[10]; // creates a default integer array with an array size of 10
    intArray[0] = 1; // setting the value in the 0th spot of the array to 1.
    string stringArray[4] = { "yes", "No", "Another", "Word" }; //setting all the variables spots with strings.
    double doubleArray[5];
    //the long of way initalizing all array spots.
    doubleArray[0] = 1.1f;
    doubleArray[1] = 1.7f;
}
```

The code above shows how to setup variables. Lets dissect what the code does a bit further. Inside the **main()** function we first start by creating a array of integers called intArray. We set the size of the array to 10. We access the arrays 0th position and set the value inside of intArray to 1. The next thing are code does is create an array of type string called stringArray setting the array size to 4. We then initialize all the positions inside of stringArray by using a **({})**. Inside the braces we set the first position to "yes", the second to "No", the third to "Another", and finally the fourth

to "word". After that we go ahead and create and array of doubles and show how long it would take to initialize all of them singly. We can also have **multidimensional arrays**, a multidimensional array's syntax looks like this. **<type>Array[ArraySizeOne][ArraySizeTwo]**. The best way to think about a multidimensional array is like columns and rows the first part is the column going up and down. While the second is the rows going left and right. Lets take a look at how we would go about setting up a multi-dimensional array.

```
#include <iostream>
#include <string>
#define cout std::cout
#define string std::string
void main()
{
    int multiArray[1][1];
    multiArray[0][0] = 1;
    float multiFArray[2][2] = { {0,1}, {1,0} }; // an
array with two rows, and two columns
}
```

The above code example shows how we would go about creating a multi-dimensional array. In the first part of the **main()** function we create a integer array with one column and one row. At position [0][0] we set the value to 1. We also create a float array that has two columns and two rows and set the values of those.

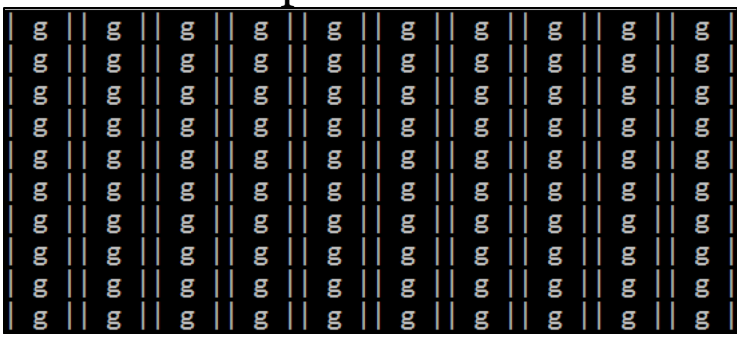
Arrays| Iterating

We can iterate over arrays to display the value, or even create a new array. Lets check out some example code:

```
#include <iostream>
#include <string>
#define cout std::cout
```

```
#define string std::string
void main()
{
	const int mapSizeX = 10;
	const int mapSizeY = 10;
	char arrayMap[mapSizeX][mapSizeY];
	//create map
	for (int x = 0; x < mapSizeX; x++)
	{
		for (int y = 0; y < mapSizeY; y++)
		{
			arrayMap[x][y] = 'g'; //g stands for
grassland.
		}
	}
	//display map
	for (int x = 0; x < mapSizeX; x++)
	{
		for (int y = 0; y < mapSizeY; y++)
		{
			cout << "| " << arrayMap[x][y] << " |";
		}
		cout << std::endl;
	}
}
```

What the output looks like:



In the above example we are creating a map. If we go in to the **main()** function we see that we first declare two constant integers to mapSizeX and mapSizeY. This is telling

our array how big we want our map to be. We then initialize our array to a type of char. We start by creating a nested for loop. In the first loop we want to loop for the mapSizeX size. Which is going up and down. The next loop we are setting the loop to loop for the size of mapSizeY. Inside of the second for loops body. We initialize our arrayMap to have a value of g. So if we are at arrayMap[0][0] it will equal g. We then basically recreate the same nested for loop to display our map. Except that when we are finished outputting the row on the screen. We go ahead and on the first for loop after the second for loop create a new line. So that we have columns going up and down.

Summary|

In this section we covered what relational operators are. How if loops work, and how we can use nested if loops. We also covered how for loops work, and how to use nested loops. We then covered arrays and how to iterate over them. In the next section we are going to get our character creation system for a rpg game really going once we get a introduction to how classes work.

Chapter 4

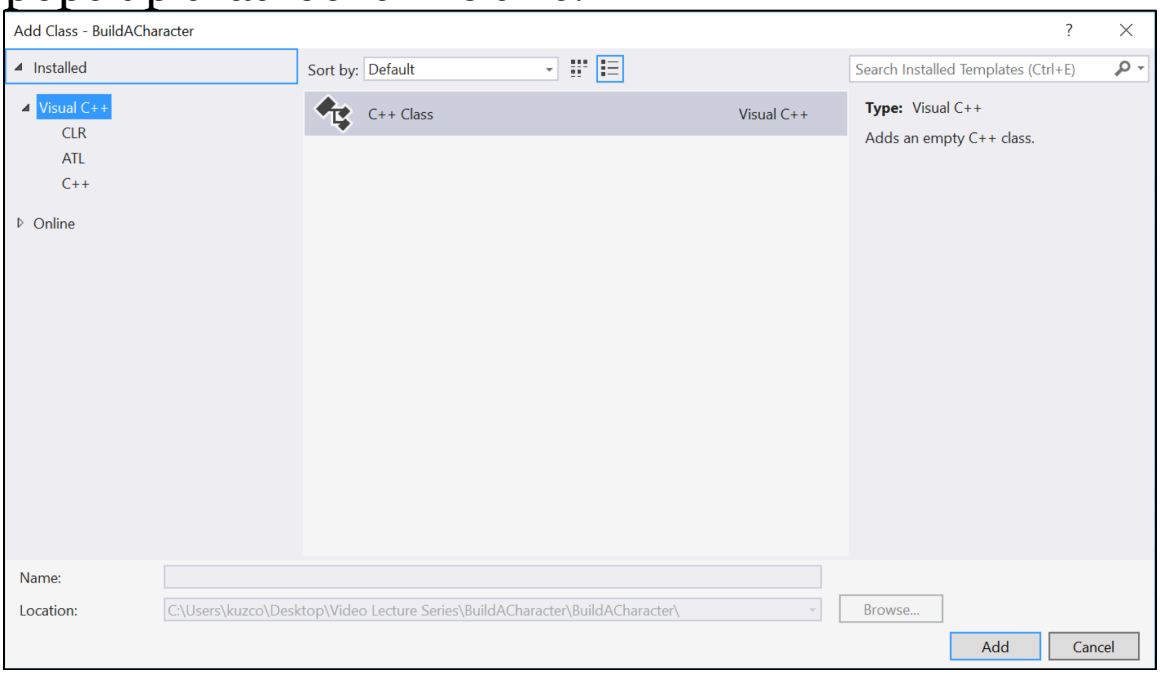
Classes & Character Creation System.

Introduction|

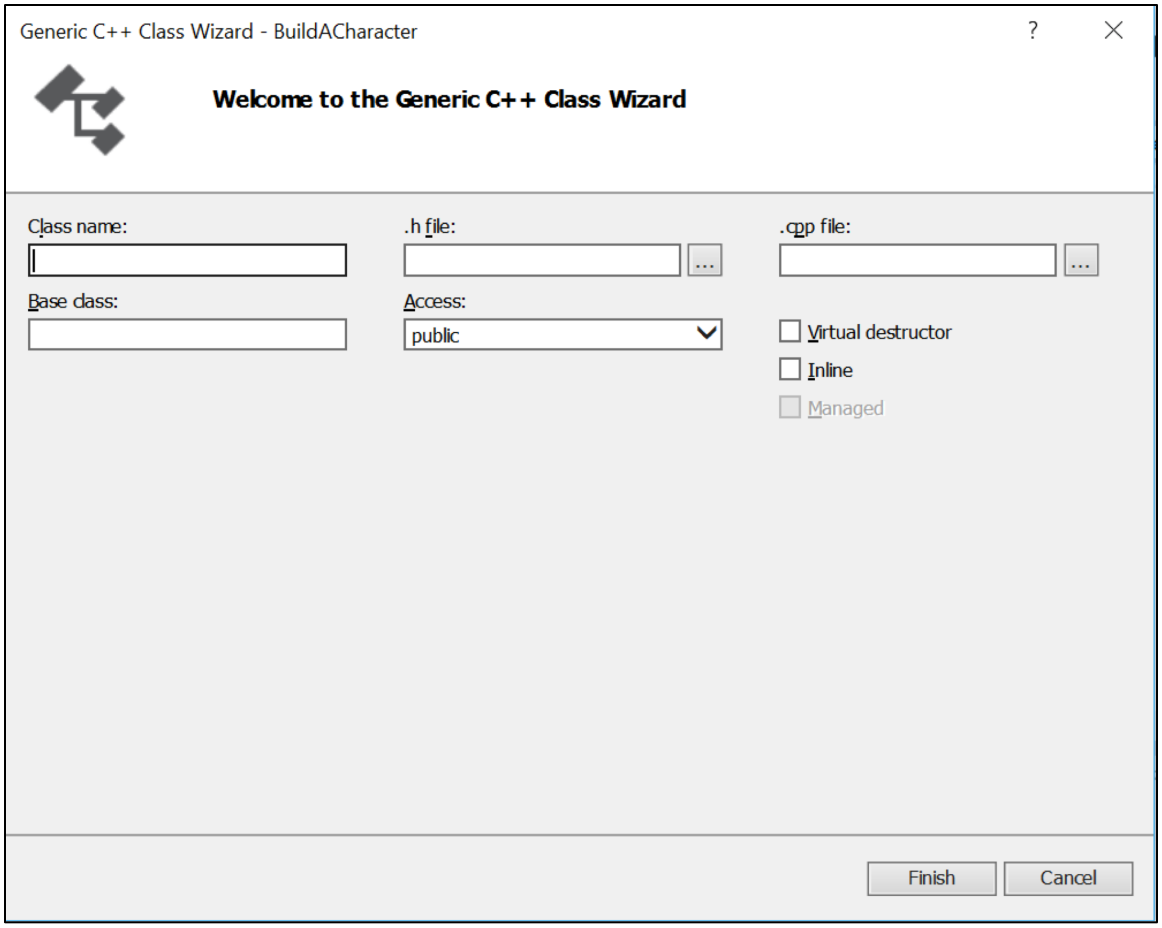
In section 4 we are going to cover **object oriented programming, classes,** and finally start getting our character creation system for a rpg game really going. **Object oriented programming**, means we are starting to think of our code files as a object or even **<type>**. We are creating our own variable, lets take for instance we want to create a variable of a character. We want to give that character a name, health, mana, and even stats. Let's start by creating a new class that is called **BaseCharacter**.

Visual Studio 2017| Creating A Class

There are five different way's to create a new class that we are going to cover. The first way is in the **solution explorer > right click on source files > add > class**. This can also be accomplished by going to the **top toolbar > selecting project > add class**. We will get a window that pops up that looks like this:

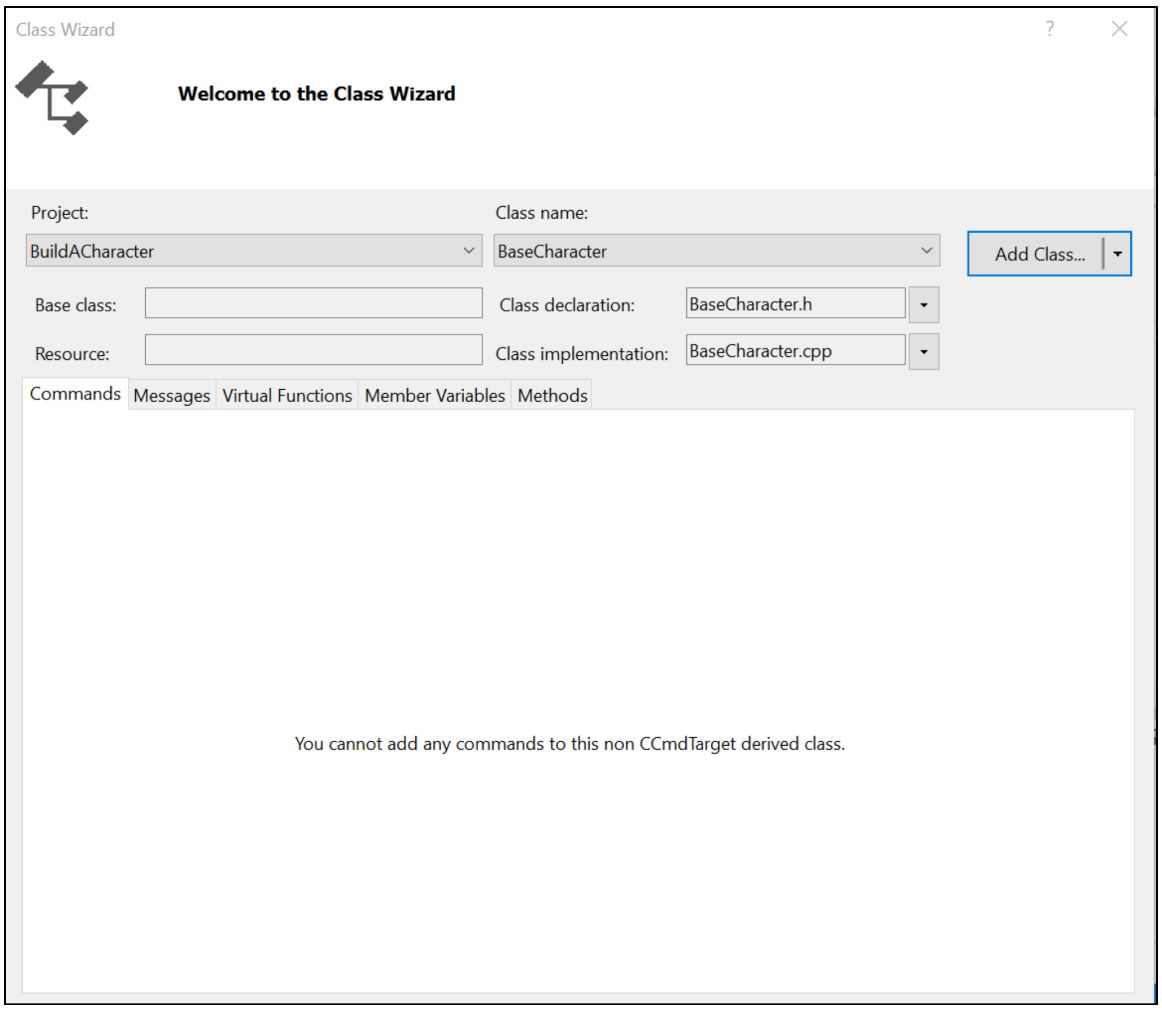


From here we want to select **C++ Class**, and the press the button **Add**. We are now looking at a window that should like this:

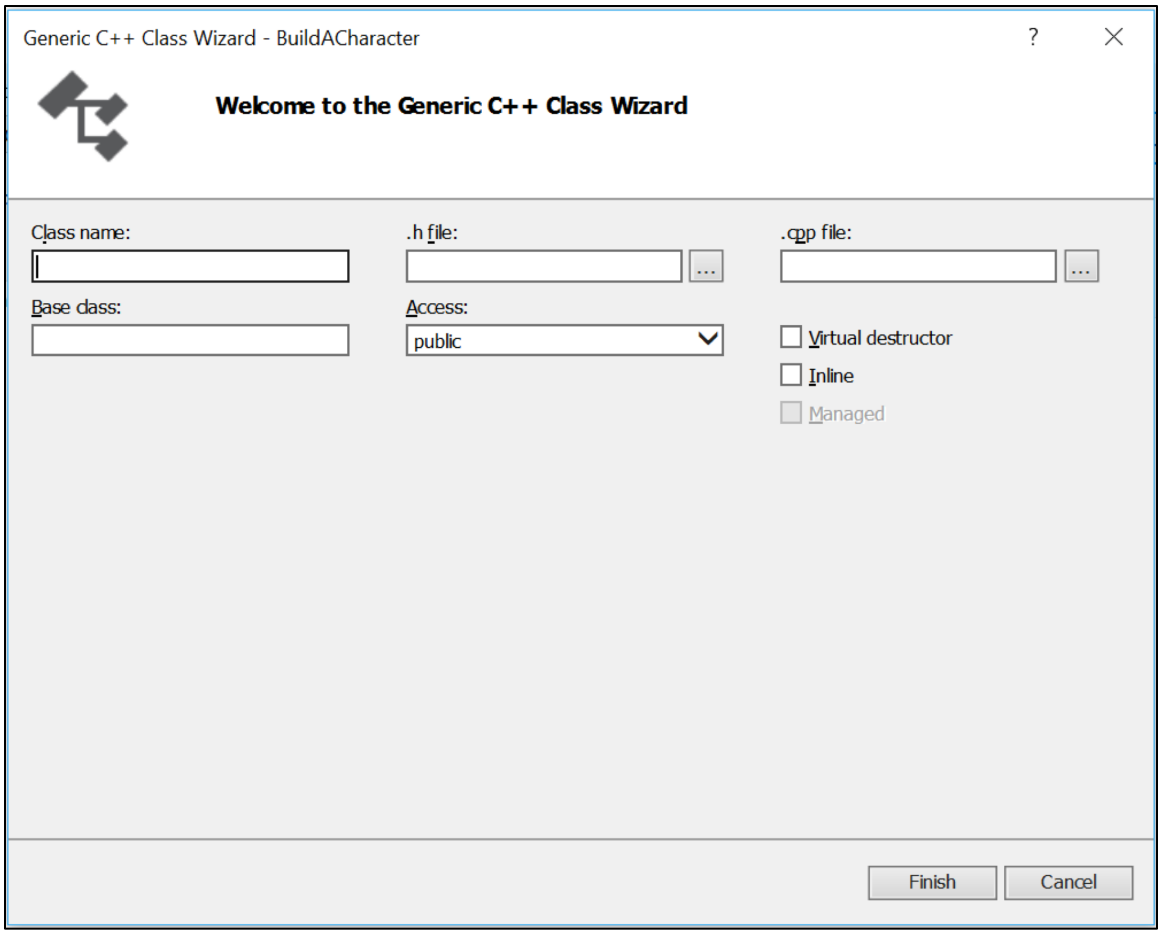


The first thing we want to add is a **class name**. A **class name** is what we will use when we want to create a new **<type>** in source code. We give our **class name**, a name of **BaseCharacter**. The next thing we will notice is the **.h file**. This is a header file, a header file contains the **function declarations or macros**, we can also store **variables**. **.cpp file** also commonly referred to as our **implementation file**, allows us to **implement** the functions from the **Header file**. It ask us for a **Base class**. A **base class** is a class we would be **deriving** or **inheriting** from. A example of this would be like a **Potion** which would have a **base class** of item. We would essentially be saying a

Potion is (inherited or derived) from a item. As our **BaseCharacter** class is going to be the base class. We will leave **Base class** blank, or empty. We will then select the **Button Finish**. We will have now created a new **BaseCharacter** class. I will show the last **three** ways to create the same class, and then we will cover what a class looks like. The final **three** ways that i am going to cover you can access by using a keyboard shortcut, **ctrl+shift+x.** Or right clicking within the **solution explorer > class wizard.** And finally going to the **top toolbar > project > class wizard.** By opening this we will get a window that look likes this:



We will want to **Select > Add Class** we will then be brought back to:



These are just a few of the ways you can create classes within in **Visual Studio 2017.**

Classes | Introduction

We will now go ahead and look at the **BaseCharacter.h** header file:

```
#pragma once
class BaseCharacter
{
public:
    BaseCharacter();
    ~BaseCharacter();
};
```

The first thing we will notice is a preprocessor directive **#pragma once**. The **pragma once** command tells our compiler that we only want to include this file once. In **unreal engine** you will notice they use these a lot. For now let's focus on the syntax of a **class**. The first part of a class is stated well with **class <name>** then we use **curly brackets({})** to open to our main body of our class. The next thing we will see is **public:** this will tells us that, this is openly accessible by anyone who **instantiates** this class. There is also **protected:** and **private:** that we can use. **protected:** allows only a class who inherits from **BaseCharacter** to use any functions or variables that we declare. **private:** allows only the class to have access to the functions or variables. We will see these more widely used later on. Next we see **BaseCharacter();** this is a **constructor**. When the class is first **instantiated** a **constructor** can be called. We can use the **constructor**, for setting up base functions, or variables we want the class to start off with. The next thing we will notice is **~BaseCharacter()**. This is called a **deconstructor**, It would be called when we are **deleting** the **instantiated class.** It would help with **memory cleanup**. Finally we will notice the class ends with a **(;)** semi-colon. Every class has to end with a semi-colon. The next thing we are going to look at is the **BaseCharacter.cpp implementation class file**:

```
#include "BaseCharacter.h"

BaseCharacter::BaseCharacter()
{
}

BaseCharacter::~BaseCharacter()
{
```

```
}
```

The first thing we will notice is we are calling the preprocessor directive of **#include "BaseCharacter.h"**. This is a way of stating we are looking for a local file. When we find the local file include it into our **implementation file**. We are including **BaseCharacter.h** because it declares the function declarations, and variables for our **implementation file.** As we go further down our file we notice **BaseCharacter::BaseCharacter(){}**. This is telling us that this **method** belongs to the **BaseCharacter** class. **(::) scope resolution operators** are used when we are telling the method that this belongs to that class. Let's go ahead and write out some code for our **BaseCharacter** class. We will begin in the **BaseCharacter.h header file:**

```
#pragma once
#include<string>
using FString = std::string;
using int32 = int;
class BaseCharacter
{
public:
    BaseCharacter();
    //setting variables
    void SetCharacterName(FString);
    void SetCharacterDescription(FString);
    void SetCurrentHealth(int32);
    void SetMaxHealth(int32);
    void SetMoney(int32);
    void SetIntelligence(int32);
    void SetEndurance(int32);
    void SetStamina(int32);
    void SetStrength(int32);
    //getting variables
    FString GetCharacterName() const;
    FString GetCharacterDescription() const;
```

```
    int32 GetCurrentHealth() const;
    int32 GetMaxHealth() const;
    int32 GetMoney() const;
    int32 GetIntelligence() const;
    int32 GetEndurance() const;
    int32 GetStamina() const;
    int32 GetStrength() const;

private:
    FString characterName;
    FString characterDescription;
    int32 currentHealth;
    int32 maxHealth;
    int32 characterMoney;
    int32 characterIntelligence;
    int32 characterEndurance;
    int32 characterStamina;
    int32 characterStrength;
};
```

In **BaseCharacter.h** in the **public:** we have add a bunch of function declarations. There are some for setting variables, and some for returning variables. Then we have a **private:** section where we have a bunch of variables we created. This is called **Data hiding**, we would hide data because we do not want to directly access the variables. We usually want to keep our variables private while allowing functions to do the public work. The reason for doing this is to keep our class maintained. Lets take a look at how these are implemented:

BaseCharacter.cpp:

```
#include "BaseCharacter.h"
//construct
BaseCharacter::BaseCharacter()
{
}
```

```
//set variable methods
void BaseCharacter::SetCharacterName(FString name)
{ characterName = name; }
void BaseCharacter::SetCharacterDescription(FString
description) { characterDescription = description; }
void BaseCharacter::SetCurrentHealth(int32 health)
{ currentHealth = health; }
void BaseCharacter::SetMaxHealth(int32 health) { maxHealth
= health; }
void BaseCharacter::SetMoney(int32 money) { characterMoney
= money; }
void BaseCharacter::SetIntelligence(int32 intelligence)
{ characterIntelligence = intelligence; }
void BaseCharacter::SetEndurance(int32 endurance)
{ characterEndurance = endurance; }
void BaseCharacter::SetStamina(int32 stamina)
{ characterStamina = stamina; }
void BaseCharacter::SetStrength(int32 strength)
{ characterStrength = strength; }
//get variables methods.
FString BaseCharacter::GetCharacterName() const { return
characterName; }
FString BaseCharacter::GetCharacterDescription() const
{ return characterDescription; }
int32 BaseCharacter::GetCurrentHealth() const { return
currentHealth; }
int32 BaseCharacter::GetMaxHealth() const { return
maxHealth; }
int32 BaseCharacter::GetMoney() const { return
characterMoney; }
int32 BaseCharacter::GetIntelligence() const { return
characterIntelligence; }
int32 BaseCharacter::GetEndurance() const { return
characterEndurance; }
int32 BaseCharacter::GetStamina() const { return
characterStamina; }
int32 BaseCharacter::GetStrength() const { return
characterStrength; }
```

Reading this will tell us that we are setting and getting the **private** variables in our class. We have now seen how a simple class is written. Lets go ahead and create a character for our game. We will start by **including BaseCharacter.h** and then instantiating our character. In our **main.cpp** file.

Main.cpp:

```
#include <iostream>
#include <string>
#include "BaseCharacter.h"
#define cout std::cout
#define cin std::cin
using int32 = int;
using FText = std::string;
int32 DisplayMenu(); //we will return a integer value
bool ChooseMenuOption(int32 choice = 0); // the first
overload we will only pass a choice.
void BeginPlay(); // We begin our game.
BaseCharacter character; //Instantiating our character.
void main()
{
    BeginPlay();
    return;
}
int32 DisplayMenu()
{
    {
        int32 getUserInput = 0;
        // Setting up our menu.
        cout << "1: Create a new character.\n";
        cout << "2: Character stats.\n";
        cout << "3: Exit.\n";
        cout << "Please input the menu item you would
like: ";
        cin >> getUserInput;
        cout << std::endl;
        return getUserInput;
    }
```

```
}
bool ChooseMenuOption(int32 choice)
{
    switch (choice)
    {
    case 1:
        cout << "Let's get started creating our
character.\n";
        //will loop forever.
        return true;
        break;
    case 2:
        cout << "These are our characters stats:\n";
        //will loop forever.
        return true;
        break;
    case 3:
        cout << "Thank you for playing our game.\n";
        return false;
        break;
    default:
        cout << "You did not put in a valid answer.\n";
        cout << std::endl;
        //will loop till we get a correct input answer.
        return true;
        break;
    }
}

void BeginPlay()
{
    bool bIsPlaying = true;
    do
    {
        bIsPlaying = ChooseMenuOption(DisplayMenu());
    } while (bIsPlaying);
    return;
}
```

As we can see we start by adding **#include "BaseCharacter.h"**, then we instantiate our character by **BaseCharacter character;**. Instantiating a character is basically the same as setting up any other variable. **<type> name;**. We can even **<type> name = Value**, as long as the value is of the same type of character. We will now go ahead and allow our user to create the character. In the **function ChooseMenuOption** within **main.cpp**, we are going to add the following code:

main.cpp

```
#include <iostream>
#include <string>
#include "BaseCharacter.h"
#define cout std::cout
#define cin std::cin
using int32 = int;
using FText = std::string;
char DisplayMenu(); //we will return a integer value
bool ChooseMenuOption(char choice = '0'); // the first
overload we will only pass a choice.
void BeginPlay(); // We begin our game.
BaseCharacter character; //Instantiating our character.
bool CheckingForErrors();
void CreateCharacter();
void main()
{
    BeginPlay();
    return;
}
char DisplayMenu()
{
    {
        FText getUserInput = "";
        // Setting up our menu.
        cout << "1: Create a new character.\n";
        cout << "2: Character stats.\n";
```

```
        cout << "3: Exit.\n";
        cout << "Please input the menu item you would
like: ";
        std::getline(cin, getUserInput);
        cout << std::endl;
        return getUserInput[0];
    }
}
bool ChooseMenuOption(char choice)
{
    bool bIsCharacterGood = false;
    switch (choice)
    {
    case '1':
        while (!bIsCharacterGood)
        {
            CreateCharacter();
            //Check to see if our user likes the
character
            bIsCharacterGood = CheckingForErrors();
        }
        return true;
        break;
    case '2':
        cout << "These are our characters stats:\n";
        //will loop forever.
        return true;
        break;
    case '3':
        cout << "Thank you for playing our game.\n";
        return false;
        break;
    default:
        cout << "You did not put in a valid answer.\n";
        cout << std::endl;
        //will loop till we get a correct input answer.
        return true;
        break;
```

```
    }
}

bool CheckingForErrors()
{
    FText input = "";
    bool bIsCheckingForErrors = true;
    while (bIsCheckingForErrors)
    {
        cout << "Do you like your character[y/n]: ";
        std::getline(cin, input);
        if (input[0] == 'y' || input[0] == 'Y')
        {
            bIsCheckingForErrors = false;
            return true;
        }
        else if (input[0] == 'n' || input[0] == 'N')
        {
            bIsCheckingForErrors = false;
            return false;
        }
        else
        {
            cout << std::endl;
            cout << "You have input a incorrect input.
\n\n";
            bIsCheckingForErrors = true;
        }
    }
}

void CreateCharacter()
{
    cout << "Let's get started creating our character.\n";
    cout << "What would you like to name your character:
";
    FText input = "";
    std::getline(cin, input);
```

```
    cout << std::endl;
    character.SetCharacterName(input);
    cout << "Your characters name: " <<
character.GetCharacterName();
    cout << std::endl;
}

void BeginPlay()
{
    bool bIsPlaying = true;
    do
    {
        bIsPlaying = ChooseMenuOption(DisplayMenu());
    } while (bIsPlaying);
    return;
}
```

We changed quite a bit of code in the **main.cpp** file. Let's go through some of the changes, and then discuss in further detail how we setup our character. First we changed **DisplayMenu()** from a **Int32** return value to a **char**. Following that we also changed the parameter in **ChooseMenuOption()** from a **int32** to a **char**. We added coded with the switch function within **ChooseMenuOption()** and created two new functions. The first function allows us to create our character. Lets take a look at our **CreateCharacter()** function. At the moment we are asking the user for the name of the **character**. We then set the name using **character.SetCharacterName(input);**. This is a very important statement, we just accesed a method within our class by using a **(.) dot operator**. The **dot operator** allows us to access all public methods or variables we have access to. We also use the **(.) dot operator** to return a value in the line **cout << "Your characters name: " <<**

character.GetCharacterName();. We can already see the power of a class. We stored a variable with the **character** class by just using a **(.) dot operator**, and a method supplied by the class we created. Let's go ahead and create another class, this time called **BaseCharacterClass**. We will then write the code:

BaseCharacterClass.h:

```
#pragma once
#include <string>
using FString = std::string;
using int32 = int;
class BaseCharacterClass
{
public:
	//set the private variables
	void SetClassName(FString);
	void SetClassDescription(FString);
	void SetClassHealth(int32);
	void SetClassEndurance(int32);
	void SetClassIntelligence(int32);
	void SetClassStamina(int32);
	void SetClassStrength(int32);
	//get the private variables
	FString GetClassName() const;
	FString GetClassDiscription() const;
	int32 GetClassHealth() const;
	int32 GetClassEndurance() const;
	int32 GetClassIntelligence() const;
	int32 GetClassStamina() const;
	int32 GetClassStrength() const;

private:
	FString className;
	FString classDescription;
	int32 classHealth;
	int32 classEndurance;
	int32 classIntelligence;
```

```
    int32 classStamina;
    int32 classStrength;
};
```

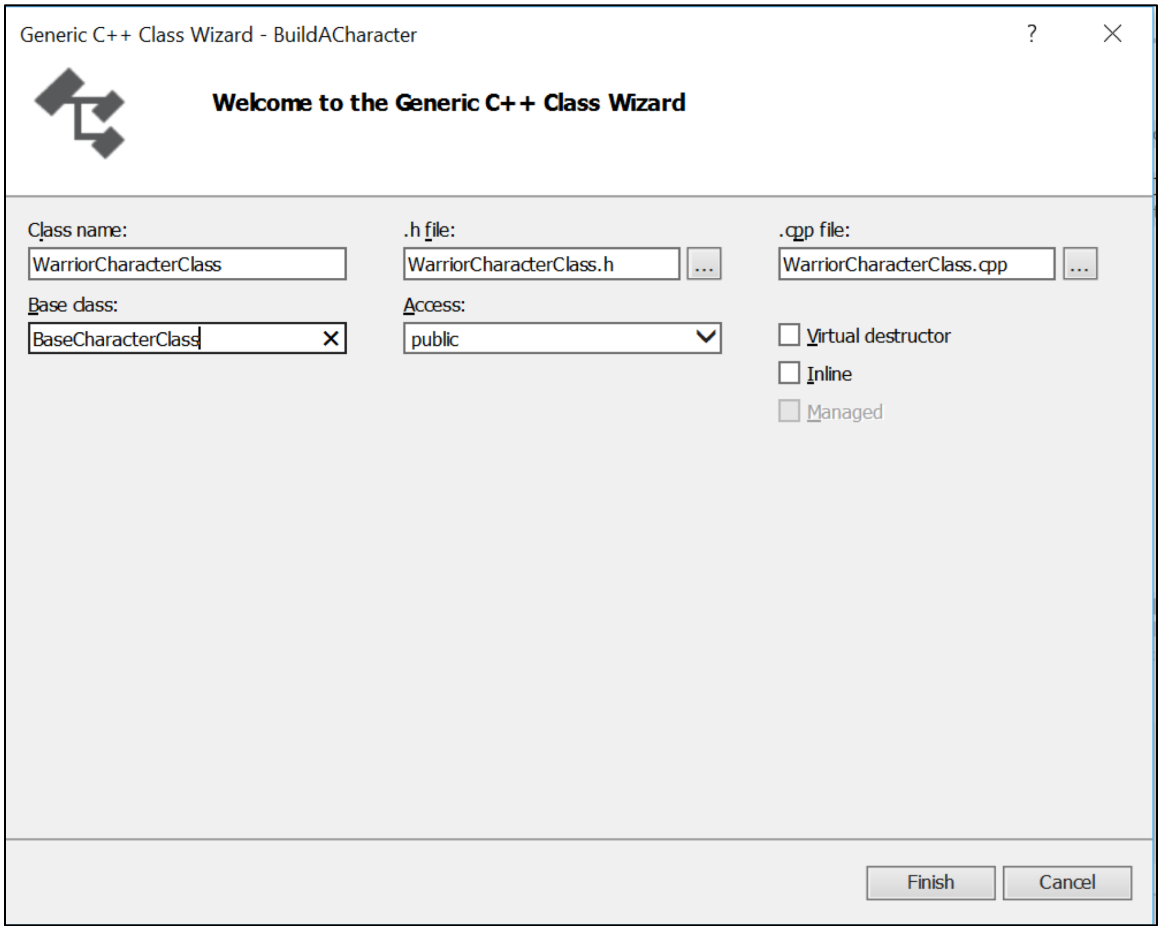
Nothing out of the ordinary yet. We are just creating some variables, and methods to set them. Lets go ahead and look at **BaseCharacterClass.cpp.**

BaseCharacterClass.cpp:

```
#include "BaseCharacterClass.h"
//set variable methods
void BaseCharacterClass::SetClassName(FString name)
{ className = name; }
void BaseCharacterClass::SetClassDescription(FString
description) { classDescription = description; }
void BaseCharacterClass::SetClassHealth(int32 health)
{ classHealth = health; }
void BaseCharacterClass::SetClassEndurance(int32
endurance) { classEndurance = endurance; }
void BaseCharacterClass::SetClassIntelligence(int32
intelligence) { classIntelligence = intelligence; }
void BaseCharacterClass::SetClassStamina(int32 stamina)
{ classStamina = stamina; }
void BaseCharacterClass::SetClassStrength(int32 strength)
{ classStrength = strength; }
//get variable methods
FString BaseCharacterClass::GetClassName() const { return
className; }
FString BaseCharacterClass::GetClassDiscription() const
{ return classDescription; }
int32 BaseCharacterClass::GetClassHealth() const { return
classHealth; }
int32 BaseCharacterClass::GetClassEndurance() const
{ return classEndurance; }
int32 BaseCharacterClass::GetClassIntelligence() const
{ return classIntelligence; }
int32 BaseCharacterClass::GetClassStamina() const { return
classStamina; }
```

```
int32 BaseCharacterClass::GetClassStrength() const
{ return classStrength; }
```

In this implementation file we just set up class to receive values. We are now going to derive from this class. Into a new class called **WarriorCharacterClass**. let's take a look at how this done. We would start by creating a class like normal but under **Base class**. We want to insert **BaseCharacterClass**. It should look like:



We then select finish. And we are now looking at **WarriorCharacterClass.h**.

WarriorCharacterClass.h:

```
#pragma once
#include "BaseCharacterClass.h"
class WarriorCharacterClass :
    public BaseCharacterClass
```

```
{
public:
    WarriorCharacterClass();
    ~WarriorCharacterClass();
};
```

We can now see that it automatically include the **BaseCharacterClass.h** file. Since we are inheriting from **BaseCharacterClass.h**, we need to include that. We then look at **class WarriorCharacterclass : public BaseCharacterClass**. This is telling us that we are inheriting from a public class called **BaseCharacterClass**. This allows us to use the methods within **BaseCharacterClass** and even extend the code in **BaseCharacterClass**. Now we will setup the **WarriorCharacterClass()**.

WarriorCharacterClass.h:

```
#pragma once
#include "BaseCharacterClass.h"
class WarriorCharacterClass :
    public BaseCharacterClass
{
public:
    WarriorCharacterClass();
};
```

All we did was remove the deconstructor from the header file. We are going to use the constructor to set up the class in give the BaseCharacterClass values.

WarriorCharacterClass.cpp:

```
#include "WarriorCharacterClass.h"

WarriorCharacterClass::WarriorCharacterClass()
{
    SetClassName("Warrior");
    SetClassDescription("A skilled sword fighter.");
```

```
    //a total of 8 skill points.
    SetClassHealth(2);
    SetClassIntelligence(1);
    SetClassEndurance(1);
    SetClassStamina(2);
    SetClassStrength(2);
}
```

We construct our warrior class and setup all the variables that we will use within the character levels up, and selects this class. We are now going to setup our character further. In **BaseCharacter()** we are going to add:

BaseCharacter.h:

```
#pragma once
#include<string>
#include "WarriorCharacterClass.h"
using FString = std::string;
using int32 = int;
class BaseCharacter
{
public:
    BaseCharacter();
    //setting variables
    void SetCharacterName(FString);
    void SetCharacterDescription(FString);
    void SetCurrentHealth(int32);
    void SetMaxHealth(int32);
    void SetMoney(int32);
    void SetIntelligence(int32);
    void SetEndurance(int32);
    void SetStamina(int32);
    void SetStrength(int32);
    void SetCharacterClass(BaseCharacterClass);
    //getting variables
    FString GetCharacterName() const;
    FString GetCharacterDescription() const;
    int32 GetCurrentHealth() const;
    int32 GetMaxHealth() const;
```

```
    int32 GetMoney() const;
    int32 GetIntelligence() const;
    int32 GetEndurance() const;
    int32 GetStamina() const;
    int32 GetStrength() const;
    BaseCharacterClass GetCharacterClass() const;

private:
    FString characterName;
    FString characterDescription;
    int32 currentHealth;
    int32 maxHealth;
    int32 characterMoney;
    int32 characterIntelligence;
    int32 characterEndurance;
    int32 characterStamina;
    int32 characterStrength;
    BaseCharacterClass characterClass;
    //Methods
    void CreateCharacter();
};
```

We have now included **WarriorCharacterClass** in **BaseCharacter.h**. Which has allowed us to create a new variable type **BaseCharacterClass** called **characterClass**. Remember that **WarriorCharacterClass** inherits from **BaseCharacterClass**. So all we are doing is using the **WarriorCharacterClass** base **class** as our **<type>**. We also added a function to create our character We are now going to set up our character in the constructor. We will set our base stat variables to 10. Then when our player has chosen a character class we call **CreateCharacter();**.

BaseCharacter.cpp:

```
#include "BaseCharacter.h"
//construct
BaseCharacter::BaseCharacter()
```

```
{
    //sets up our base stats.
    SetMaxHealth(10);
    SetCurrentHealth(maxHealth);
    SetEndurance(10);
    SetIntelligence(10);
    SetStamina(10);
    SetStrength(10);
    SetMoney(0);
}
//set variable methods
void BaseCharacter::SetCharacterName(FString name)
{ characterName = name; }
void BaseCharacter::SetCharacterDescription(FString
description) { characterDescription = description; }
void BaseCharacter::SetCurrentHealth(int32 health)
{ currentHealth = health; }
void BaseCharacter::SetMaxHealth(int32 health) { maxHealth
= health; }
void BaseCharacter::SetMoney(int32 money) { characterMoney
= money; }
void BaseCharacter::SetIntelligence(int32 intelligence)
{ characterIntelligence = intelligence; }
void BaseCharacter::SetEndurance(int32 endurance)
{ characterEndurance = endurance; }
void BaseCharacter::SetStamina(int32 stamina)
{ characterStamina = stamina; }
void BaseCharacter::SetStrength(int32 strength)
{ characterStrength = strength; }
void BaseCharacter::SetCharacterClass(BaseCharacterClass
classType) { characterClass = classType; }
//get variables methods.
FString BaseCharacter::GetCharacterName() const { return
characterName; }
FString BaseCharacter::GetCharacterDescription() const
{ return characterDescription; }
int32 BaseCharacter::GetCurrentHealth() const { return
currentHealth; }
```

```
int32 BaseCharacter::GetMaxHealth() const { return
maxHealth; }
int32 BaseCharacter::GetMoney() const { return
characterMoney; }
int32 BaseCharacter::GetIntelligence() const { return
characterIntelligence; }
int32 BaseCharacter::GetEndurance() const { return
characterEndurance; }
int32 BaseCharacter::GetStamina() const { return
characterStamina; }
int32 BaseCharacter::GetStrength() const { return
characterStrength; }
BaseCharacterClass BaseCharacter::GetCharacterClass()
const { return characterClass; }
void BaseCharacter::CreateCharacter()
{
    //set up our stats.
    SetMaxHealth(GetCharacterClass().GetClassHealth() +
GetMaxHealth());
    SetCurrentHealth(GetMaxHealth());
    SetEndurance(GetCharacterClass().GetClassEndurance() +
GetEndurance());

SetIntelligence(GetCharacterClass().GetClassIntelligence()
+ GetIntelligence());
    SetStamina(GetCharacterClass().GetClassStamina() +
GetStamina());
    SetStrength(GetCharacterClass().GetClassStrength() +
GetStrength());
}
```

We are now going to setup our character in **main.cpp**.

main.cpp:

```
#include <iostream>
#include <string>
#include "BaseCharacter.h"
#define cout std::cout
#define cin std::cin
```

```
using int32 = int;
using FText = std::string;
char DisplayMenu(); //we will return a integer value
bool ChooseMenuOption(char choice = '0'); // the first
overload we will only pass a choice.
void BeginPlay(); // We begin our game.
BaseCharacter character; //Instantiating our character.
bool CheckingForErrors();
void CreateCharacter();
FText ChooseName();
FText ChooseDescription();
BaseCharacterClass ChooseClass();
void DisplayStats();

void main()
{
    BeginPlay();
    return;
}

char DisplayMenu()
{
    {
        FText getUserInput = "";
        // Setting up our menu.
        cout << "1: Create a new character.\n";
        cout << "2: Character stats.\n";
        cout << "3: Exit.\n";
        cout << "Please input the menu item you would
like: ";
        std::getline(cin, getUserInput);
        cout << std::endl;
        return getUserInput[0];
    }
}

bool ChooseMenuOption(char choice)
{
```

```
    bool bIsCharacterGood = false;
    switch (choice)
    {
    case '1':
        while (!bIsCharacterGood)
        {
            CreateCharacter();
            //Check to see if our user likes the
character
            bIsCharacterGood = CheckingForErrors();
        }
        return true;
        break;
    case '2':
        cout << "These are our characters stats:\n";
        //will loop forever.
        DisplayStats();
        return true;
        break;
    case '3':
        cout << "Thank you for playing our game.\n";
        return false;
        break;
    default:
        cout << "You did not put in a valid answer.\n";
        cout << std::endl;
        //will loop till we get a correct input answer.
        return true;
        break;
    }
}

bool CheckingForErrors()
{
    FText input = "";
    bool bIsCheckingForErrors = true;
    while (bIsCheckingForErrors)
    {
```

```
            cout << "Do you like your character[y/n]: ";
            std::getline(cin, input);
            if (input[0] == 'y' || input[0] == 'Y')
            {
                bIsCheckingForErrors = false;
                return true;
            }
            else if (input[0] == 'n' || input[0] == 'N')
            {
                bIsCheckingForErrors = false;
                return false;
            }
            else
            {
                cout << std::endl;
                cout << "You have input a incorrect input.
\n\n";
                bIsCheckingForErrors = true;
            }
        }
}

void CreateCharacter()
{
    cout << "Let's get started creating our character.\n";

    character.SetCharacterName(ChooseName());

character.SetCharacterDescription(ChooseDescription());
    character.SetCharacterClass(ChooseClass());
    character.CreateCharacter();
    DisplayStats();
    //set new character stats.
    cout << std::endl;
}

FText ChooseName()
{
```

```
    cout << "What would you like to name your character:
";
    FText input = "";
    std::getline(cin, input);
    cout << std::endl;
    return input;
}

FText ChooseDescription()
{
    bool bHasChosen = false;
    while (!bHasChosen)
    {
        FText input = "";
        FText farmer = "You were raised on a farm. You do
not have any real battle prowess.";
        FText royal = "You were raised by royals. You
haven been trained in combat.";
        FText outsider = "The world has never accepted
you. You have trained by yourself for years on end.";
        cout << "Descriptions:\n";
        cout << "1: " << farmer << std::endl;
        cout << "2: " << royal << std::endl;
        cout << "3: " << outsider << std::endl;
        cout << "Please choose your description: ";
        std::getline(cin, input);
        cout << std::endl;
        switch (input[0])
        {
        case '1':
            bHasChosen = true;
            return farmer;
            break;
        case '2':
            bHasChosen = true;
            return royal;
            break;
        case '3':
```

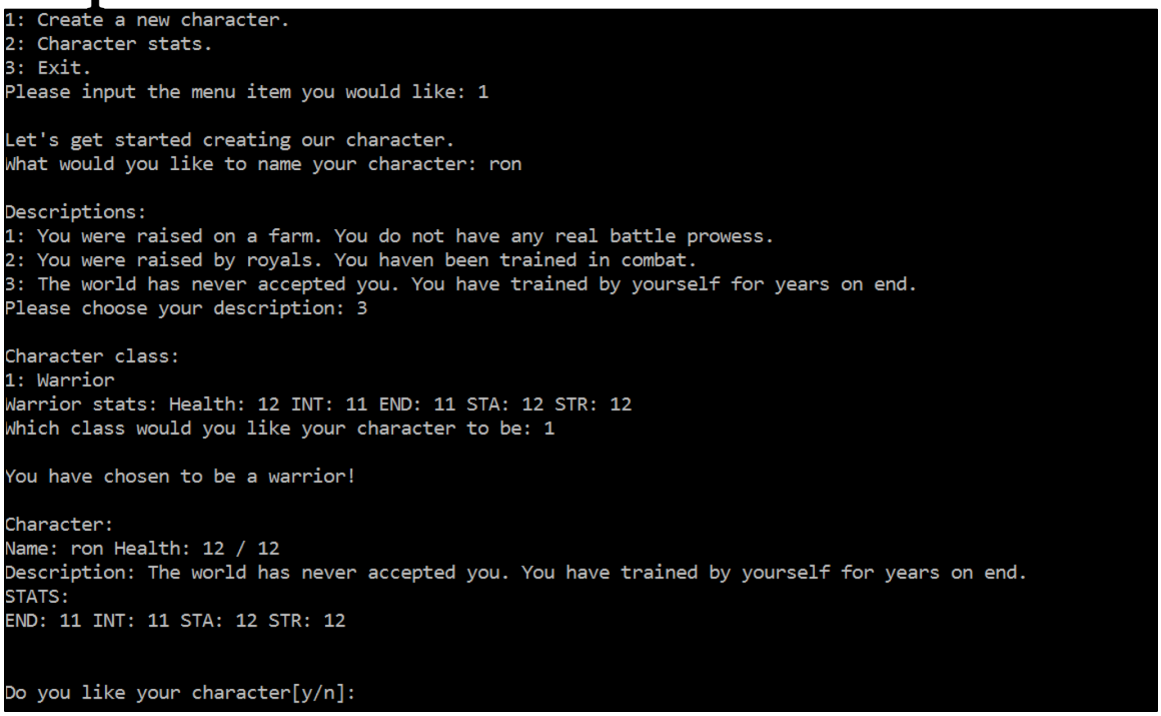
```
            bHasChosen = true;
            return outsider;
            break;
        default:
            cout << "You have not input a correct answer.
Please try again.\n\n";
            bHasChosen = false;
            break;
        }

    }
}

BaseCharacterClass ChooseClass()
{
    bool bIsClassSelected = false;
    while (!bIsClassSelected)
    {
        FText input = "";
        cout << "Character class:\n";
        cout << "1: Warrior\n";
        cout << "Warrior stats: Health: " <<
WarriorCharacterClass().GetClassHealth() +
BaseCharacter().GetCurrentHealth();
        cout << " INT: " <<
WarriorCharacterClass().GetClassIntelligence() +
BaseCharacter().GetIntelligence();
        cout << " END: " <<
WarriorCharacterClass().GetClassEndurance() +
BaseCharacter().GetEndurance();
        cout << " STA: " <<
WarriorCharacterClass().GetClassStamina() +
BaseCharacter().GetStamina();
        cout << " STR: " <<
WarriorCharacterClass().GetClassStrength() +
BaseCharacter().GetStrength();
        cout << std::endl;
```

```
        cout << "Which class would you like your
character to be: ";
        std::getline(cin, input);
        cout << std::endl;
        switch (input[0])
        {
        case '1':
            cout << "You have chosen to be a warrior!
\n\n";
            return WarriorCharacterClass();
            bIsClassSelected = true;
            break;
        default:
            cout << "You have chosen a wrong input.
Choose wisely.\n\n";
            bIsClassSelected = false;
            break;
        }
    }
}

void DisplayStats()
{
    cout << "Character:\n";
    cout << "Name: " << character.GetCharacterName();
    cout << " Health: " << character.GetCurrentHealth() <<
" / " << character.GetMaxHealth() << std::endl;
    cout << "Description: " <<
character.GetCharacterDescription() << std::endl;
    cout << "STATS:\n";
    cout << "END: " << character.GetEndurance();
    cout << " INT: " << character.GetIntelligence();
    cout << " STA: " << character.GetStamina();
    cout << " STR: " << character.GetStrength();
    cout << std::endl << std::endl;
}

void BeginPlay()
```

```
{
    bool bIsPlaying = true;
    do
    {
        bIsPlaying = ChooseMenuOption(DisplayMenu());
    } while (bIsPlaying);
    return;
}
```

We created a few new functions. **DisplayStats(), ChooseClass(), ChooseDescription(), ChooseName()**. We also update a few sections in our code to add these functions to them. In **ChooseMenuOptions()** we updated the switch statement to show our stats. And finally under **CreateCharacter()**. We finally finished our character creation program.

output:



```
Do you like your character[y/n]: y
1: Create a new character.
2: Character stats.
3: Exit.
Please input the menu item you would like: 2

These are our characters stats:
Character:
Name: ron Health: 12 / 12
Description: The world has never accepted you. You have trained by yourself for years on end.
STATS:
END: 11 INT: 11 STA: 12 STR: 12

1: Create a new character.
2: Character stats.
3: Exit.
Please input the menu item you would like:
```

Summary |

I know this chapter was really hard to follow, and if you have made it here. You should have a great understanding of how to program in c++. You will also have learned how to create your own character creation system. The next chapter will cover the very last part of c++, you should need to know before entering unreal engine.

Chapter 5

Pointers, References, and operator overloading

Introduction |

In this chapter we are going to cover the final bit of c++ we need to learn before we jump into unreal engine. We will cover **(*)Pointers** and how they work. **(&)References** and how they work. Then discuss **operator overloading**. After

we have covered this we will move on to a introduction in unreal engine.

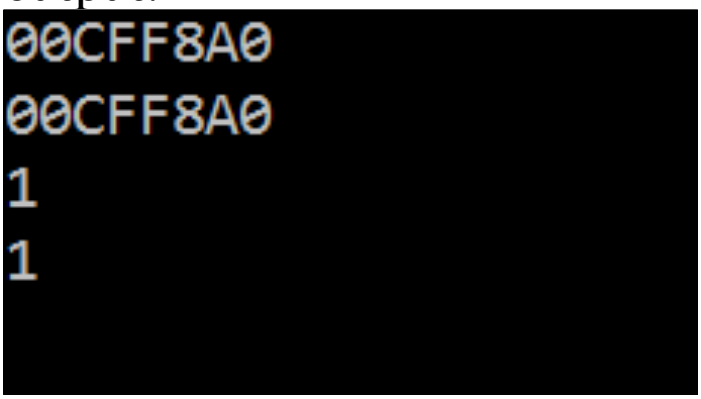
Pointers & References | Introduction

A **pointer** is a value that **points** to a space in **computer memory** using it's **memory address.** Let's take a look at the syntax for a pointer. **<type> *value = (address);** first we see that we create a type of variable then we add a **(*)** pointer symbol. The symbol can be written a few different ways i will only show two more ways of writing it. **<type>* value = (address);, <type> * value = (address);**. We also need to know what the **(&) address of operator** allows us to get. The **(&)**address of operator allows us get the the address of value. When using the address of operator we would write code that looks like this. **<type> *value = &value;**. We would first declare a **pointer type** then give it a name and give it the value of a **memory address** at **value**. We can also use the **(&)** symbol to create a **reference, <type> &value = value;**. It allows us to create a safer data type than a pointer. We are creating a reference to the value. Lets take a look at some example code:

```
#include <iostream>

void main()
{
    int variableOne = 1; // creating a normal variable.
    int &variableREF = variableOne; // Creating a
reference of variable one.
    int *variablePTR = &variableOne; //setting a pointer
equal to the memory address of our variable.
    std::cout << variablePTR << std::endl; //Shows the
memory address
```

```
    std::cout << &variableREF << std::endl; //Shows the
memory address.
    std::cout << variableREF << std::endl; //Shows the
value of variable one.
    std::cout << *variablePTR << std::endl; //
dereferencing shows the value of variable one.
    return;
}
```

output:

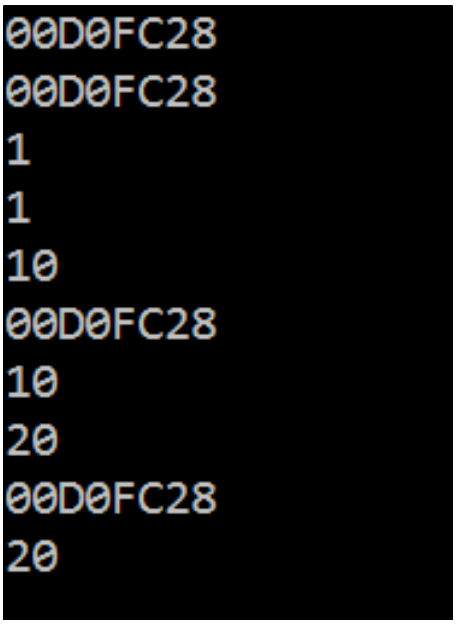


We can see that both **variablePTR & variableREF**, point to the same address of variable one. We can also see they have the same **value**. What this is allowing us to do is access the variable where the memory address is. We can change the value of the stored variable with the memory address and lots of other things. You may have noticed we added a **(*)variablePTR** in our cout statement. This is called dereferencing it will allow us to see or change what is stored within the memory address. Let look at some more example code:

```
#include <iostream>

void main()
{
    int variableOne = 1; // creating a normal variable.
    int &variableREF = variableOne; // Creating a
reference of variable one.
```

```
    int *variablePTR = &variableOne; //setting a pointer
equal to the memory address of our variable.
    std::cout << variablePTR << std::endl; //Shows the
memory address
    std::cout << &variableREF << std::endl; //Shows the
memory address.
    std::cout << variableREF << std::endl; //Shows the
value of variable one.
    std::cout << *variablePTR << std::endl; //
dereferencing shows the value of variable one.
    *variablePTR = 10;
    std::cout << *variablePTR << std::endl; //will display
the new value.
    std::cout << variablePTR << std::endl; //will display
the memory address.
    std::cout << variableOne << std::endl; //will display
the new value.
    variableREF = 20;
    std::cout << variableREF << std::endl; //will display
the new value.
    std::cout << &variableREF << std::endl; //will display
the memory address.
    std::cout << variableOne << std::endl; //will display
the new value.
    return;
}
```

Output:



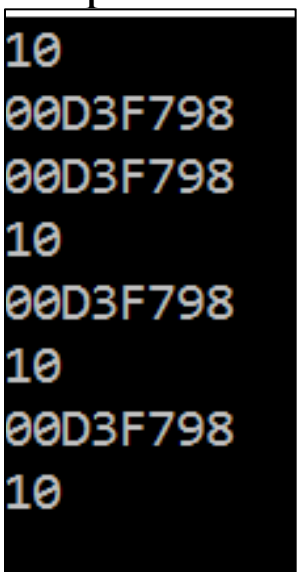
Now we can see that we have updated the value within the same memory address twice. We can see that both variable reference's and pointers, are very powerful. So what else can we do with them? We can use them while passing parameters. We can also use them to return more than one value from functions.

Passing parameters using pointers or references are very easy. We would do this when we want to keep the value the same. And use it across multiple functions, this will also help with keeping performance a lot faster. Let's take a look at passing via pointers:

```
#include <iostream>
void PassingPointerParameter(int *value)
{
    std::cout << value << std::endl; //will give us the
memory
    std::cout << *value << std::endl; // will give us the
value.
}
void main()
{
    int value = 10; //set our value.
```

```
    int *valuePTR = &value; //set our pointer to the
memory address of value.
    std::cout << value << std::endl; // display value
    std::cout << &value << std::endl; // address of value
    std::cout << valuePTR << std::endl; // display ptr
address
    std::cout << *valuePTR << std::endl; // display ptr
value
    PassingPointerParameter(valuePTR); //Display our
pointers value by passing valuePTR
    PassingPointerParameter(&value); //Display our
pointers value by passing the address of value.
    return;
}
```

Output:



From the example above we can see that we are passing in the memory address. We are also able to access the memory address and use it's value or change the value. Lets look at an example where we change the value within the function. Then display it back in main.

```
#include <iostream>
void PassingPointerParameter(int *value)
{
```

```
    std::cout << value << std::endl; //will give us the
memory
    std::cout << *value << std::endl; // will give us the
value.
    *value = 15;
}
void main()
{
    int value = 10; //set our value.
    int *valuePTR = &value; //set our pointer to the
memory address of value.
    std::cout << value << std::endl; // display value
    std::cout << &value << std::endl; // address of value
    std::cout << valuePTR << std::endl; // display ptr
address
    std::cout << *valuePTR << std::endl; // display ptr
value
    PassingPointerParameter(valuePTR); //Display our
pointers value by passing valuePTR
    std::cout << valuePTR << std::endl; //display the
memory address
    std::cout << *valuePTR << std::endl; //display the new
value changed from within the function
    PassingPointerParameter(&value); //Display our
pointers value by passing the address of value.
    return;
}
```

Output:

```
10
012FF96C
012FF96C
10
012FF96C
10
012FF96C
15
012FF96C
15
```

If we examine the code, we can tell that the original value starts of as **10**, and then we call the function **PassingPointerParameter**. We then change the value to **15**. We return back to main display the value and it is now **15**. When we pass pointer parameters it also changes the value. At the same time though imagine if we had to pass back multiple values from a function. All we would have to do is add more pointer parameters. We are now going to create a reference and get back more than one return value. Let us examine how a reference would work:

```
#include <iostream>
void PassingPointerParameter(int &outX, int &outY)
{
    std::cout << &outX << std::endl; //will give us the
memory
    std::cout << outX << std::endl; // will give us the
value.
    std::cout << &outY << std::endl; // will give us the
memory
    std::cout << outY << std::endl; // will give us the
value.
    outX = 15;
    outY = 50;
}
void main()
{
    int value = 10; //set our value.
    int valueTwo = 20;
    int &valueREF = value; //set a reference value.
    int &valueTwoREF = valueTwo; //set a reference of
valueTwo.
    std::cout << value << std::endl; // display value
    std::cout << &value << std::endl; // address of value
    std::cout << valueREF << std::endl; //value of
valueREF
```

```
    std::cout << &valueREF << std::endl; //address of
valueREF.
    std::cout << valueTwo << std::endl; //value of
valueTwo
    std::cout << &valueTwo << std::endl; //address of
valueTwo
    std::cout << valueTwoREF << std::endl; //value of
valueTwoREF
    std::cout << &valueTwoREF << std::endl; //address of
valueTwoREF.
    PassingPointerParameter(valueREF,valueTwoREF); //
Display our pointers value by passing valuePTR
    std::cout << valueREF << std::endl; //display the new
valueREF value.
    std::cout << valueTwoREF << std::endl; //display the
new valueTwoREF value.
    PassingPointerParameter(value, valueTwo); //Display
our pointers value by passing the address of value.
    std::cin >> value;
    return;
}
```

Output:

```
10
0118F784
10
0118F784
20
0118F778
20
0118F778
0118F784
10
0118F778
20
15
50
0118F784
15
0118F778
50
```

By looking at the above program we can tell we are, passing in to reference's and setting there value, when we return back to the main function we output the new value. We see that the new value was set at the memory address location. We should now have a basic enough understanding of pointers and references, to be able to understand what is going on in the unreal engine code.

Overloading Operators|

Operator overloading is the process in which we overload operators such as +,-,/,*, ==, +=, !=. We can set these up within code to do specific routines. Let's take a look at the syntax. **<type> operator+(parameters)**. We the use the **operator** keyword followed by the actual operator we want to define. We will see this used within unreal engine. I am not going to provide any code examples for this.

Summary|

We are now ready to move on to unreal engine, and get an introduction to the engine. In this chapter we learned about pointers, and references and how they work. We also took a small look at what an operator overload was.

Assignment|

1: Pointer Assignment & Reference Assignment
Set up some pointers, references , and mess around with them.
2:Messing around with operator overloads.

This a assignment is totally up to you. Build what ever you would like that uses operator overloads. Operator overloads are generally set up in classes and used to do something like add one value that is specified with in the class to another value not specified within the class. Or you can even add two of the same class types like, creating a vector class and passing in two vector parameters.

Chapter 6

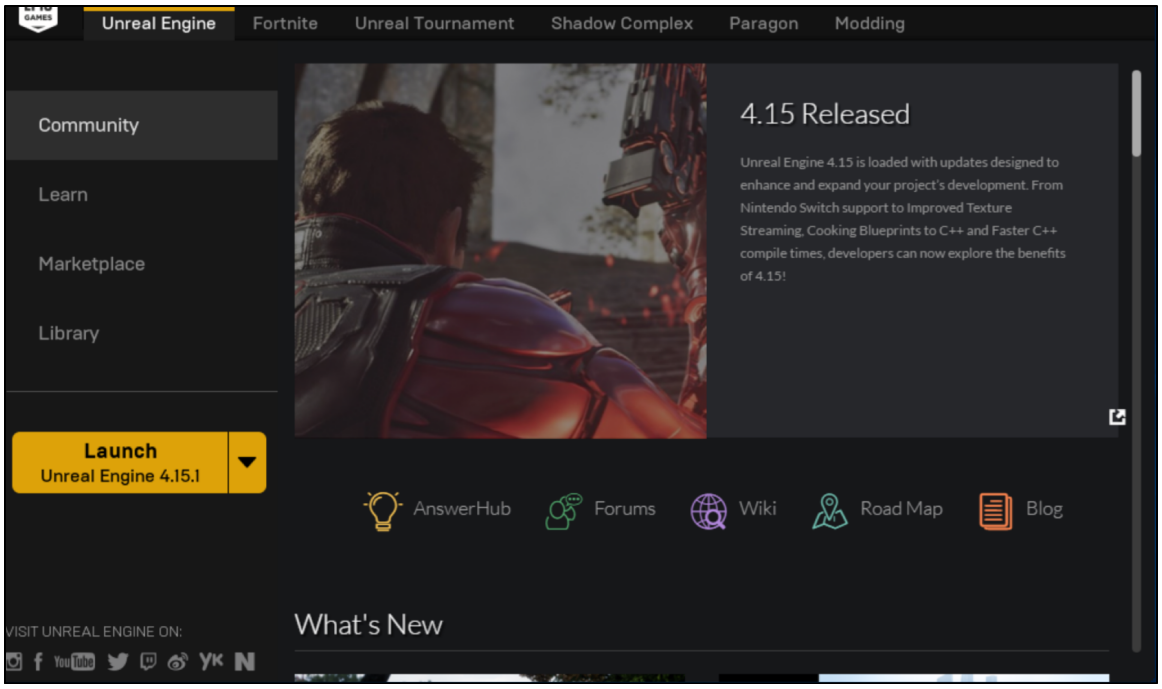
Introduction To Unreal Engine

Introduction |

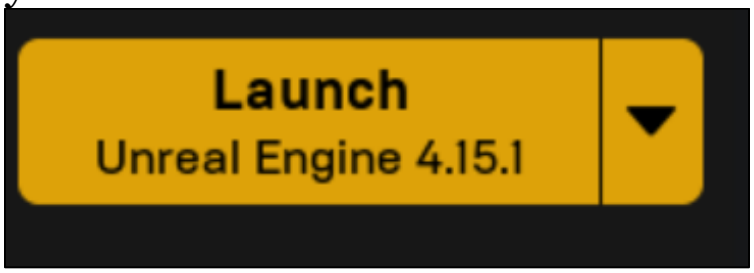
We are now ready to begin looking at the unreal engine interface, and getting a basic understanding of how it all works. Once you have downloaded unreal engine. Please open the shortcut.

Unreal Engine| Launch

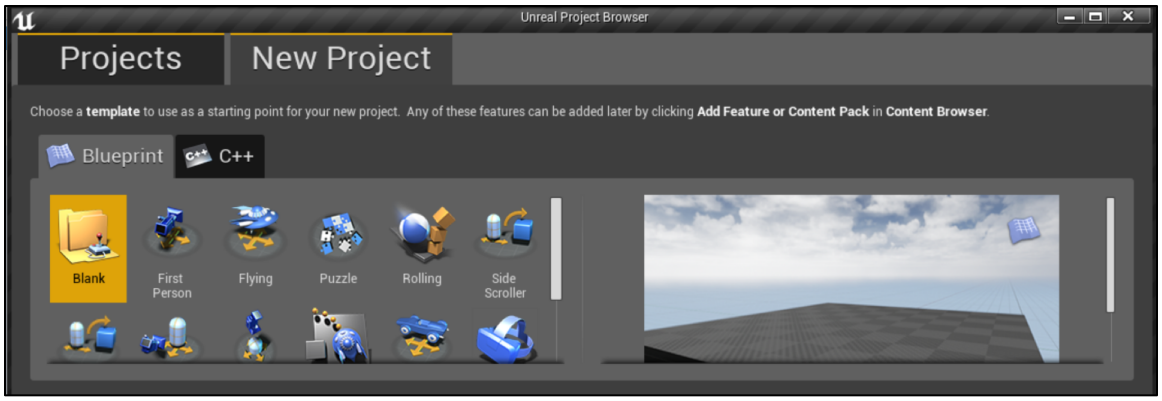
Once you have opened the shortcut to unreal engine you will get a **launch window** that looks like this as of 4.15.1:



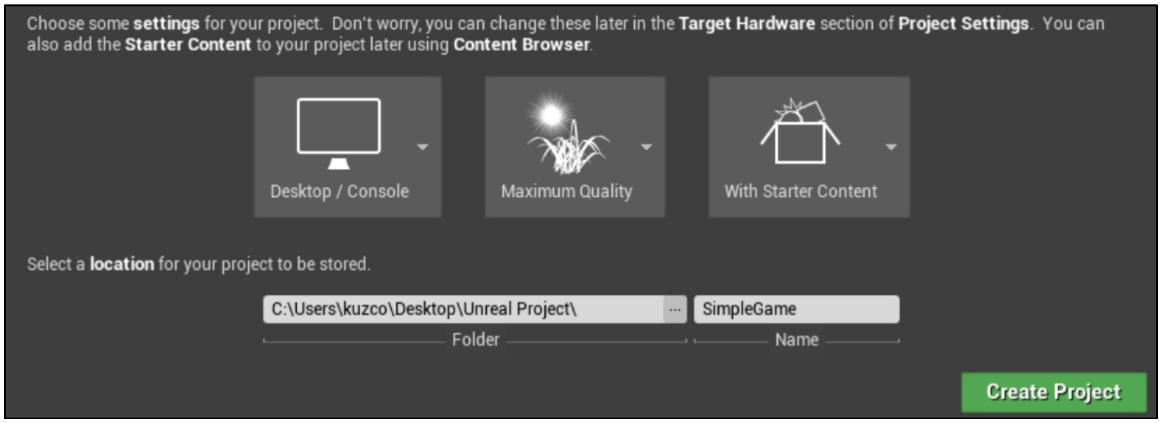
This is our **launch window**, within in this window we can see a lot of cool things. We have access to the newest information about the engine. Answerhub or forums if we need help developing a game or even to report a bug. We also have access to the **marketplace**, where you can buy/ download all kinds of cool stuff like examples, models, plugins, etc. For now we are going to go ahead and click the yellow **launch button**.



After we have clicked the yellow launch button, we are greeted by the **unreal project browser**. Within the **unreal project browser**. We can create a new project or even load a current one we are working on.

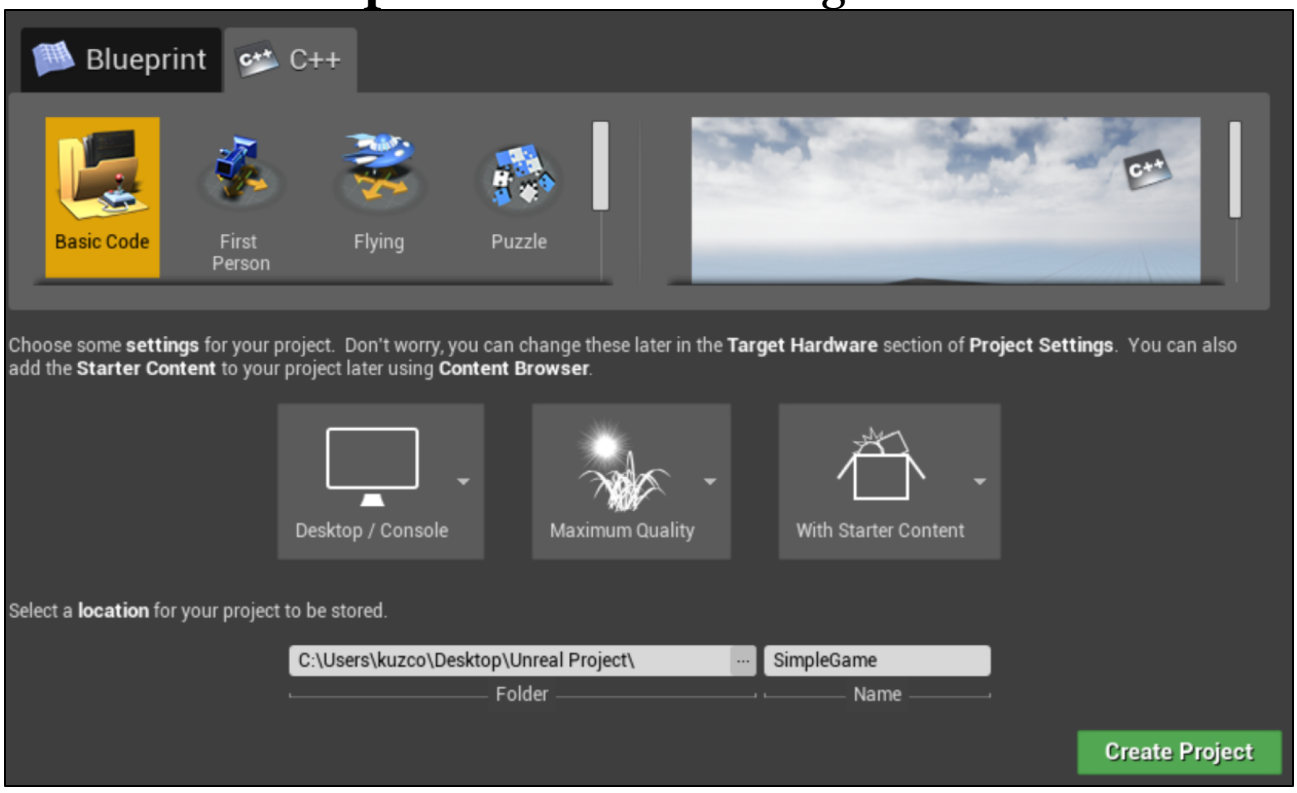


In the first half of the **unreal project browser**. We can determine what kind of project we would like to build. We have the option of a **blueprint project**, or a **c++ project**. We also can select what type of **project template**, we would like to start out with.



In this section we can choose what hardware we are going to be aim for. Whether that would be desktop/console or mobile/tablet. If we are not running very powerful hardware we can take down our quality. We can also choose wether we would like to start with starter content or not. Then we can specify the location of our project. Finally we can give our project a name. Then finally once we have set up all our settings we can click **create project**. The setting's we are going to use are going to be as follows. First we are going to select the **c++ tab**, then **basic code template**.

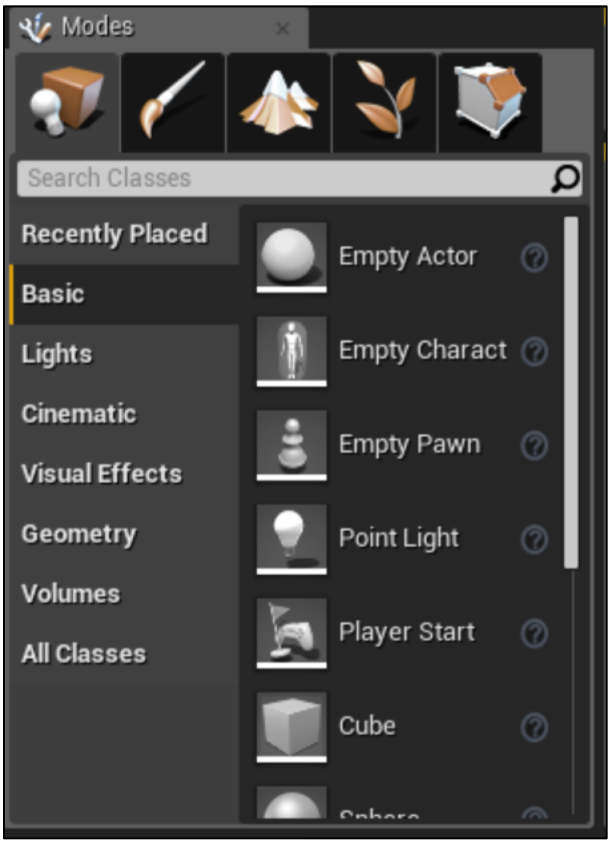
Next we are going to make a **Desktop / console** game. We can stick with **Maximum quality**. We will include the **starter content**. Finally we will put our game in a **folder** on our **desktop**, called **Unreal Project**. We will give our game the name of **SimpleGame**. You settings should look like:



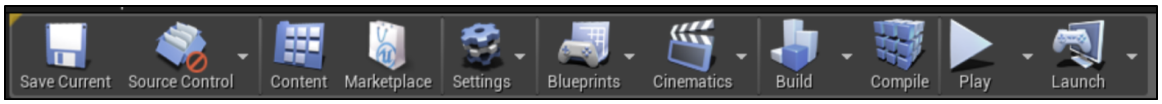
We will then mouse click on the button that states **Create Project**. We have now created our first unreal engine project. We will now take a look at how the interface looks.

Unreal Engine| Interface

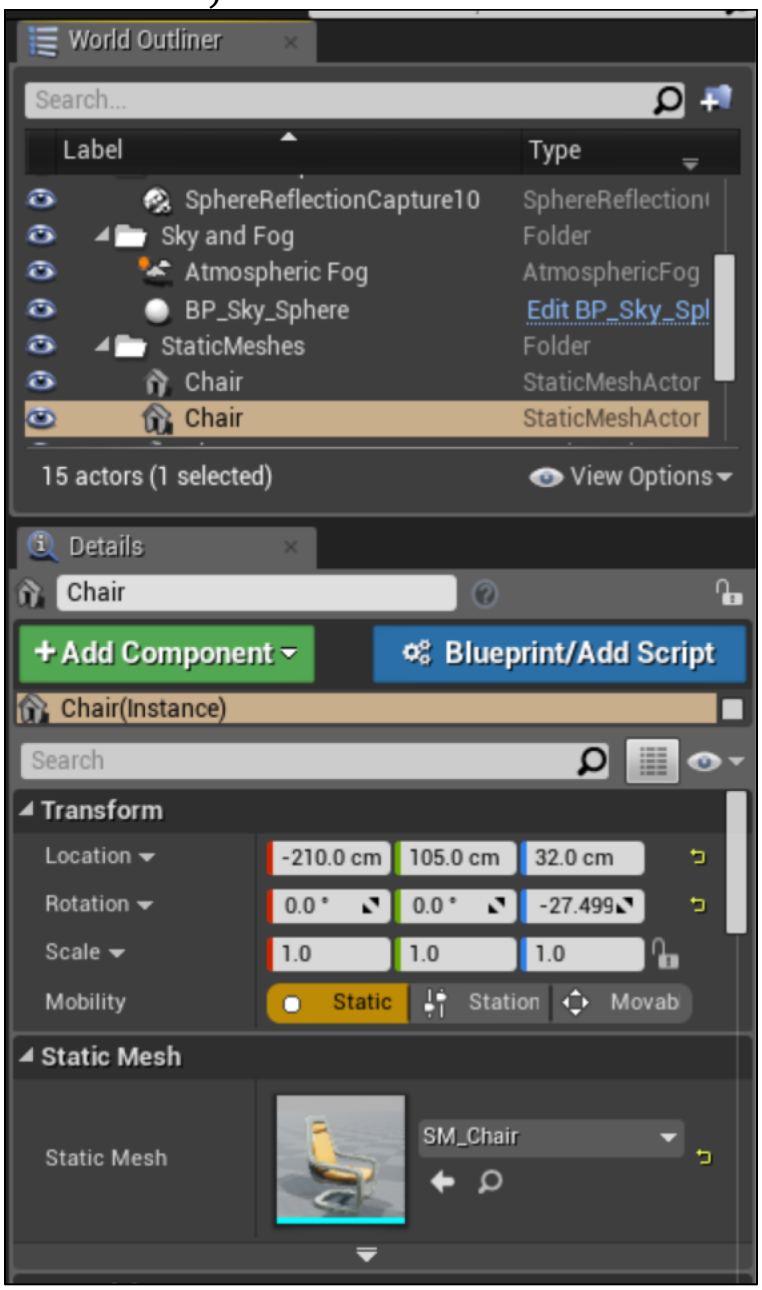
We will be greeted by **visual studio** opening and showing us our project files. For the time being we can close out of **visual studio**. This section is going to cover unreal engine. In **section 7** we will cover **visual studio** and how it works with unreal engine. When the **unreal engine editor** finally opens up. We will notice a level in the center with two chairs, and a table. A bunch of windows around it. We will start by breaking down each window and what they do. The first window we will cover will be the **modes window**.



In the **modes window** we can add objects to our scene. Whether we want to add actors, lights, visual effects, volumes, and a lot more. We can even edit geometry or make landscapes using the modes panel. The next window we will look at is the **toolbar window**.

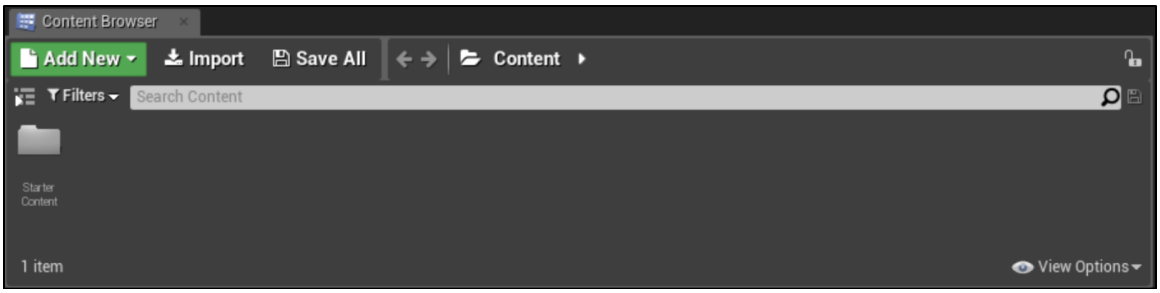


In the **toolbar window**, we can save what we are working, build stuff, compile our source code, play our game and even launch our game. The next window is the **world outliner, and details window**.



In the **world outliner**, we can see exactly what is within our scene. If we look at the **chair**, we can see that it is **static**

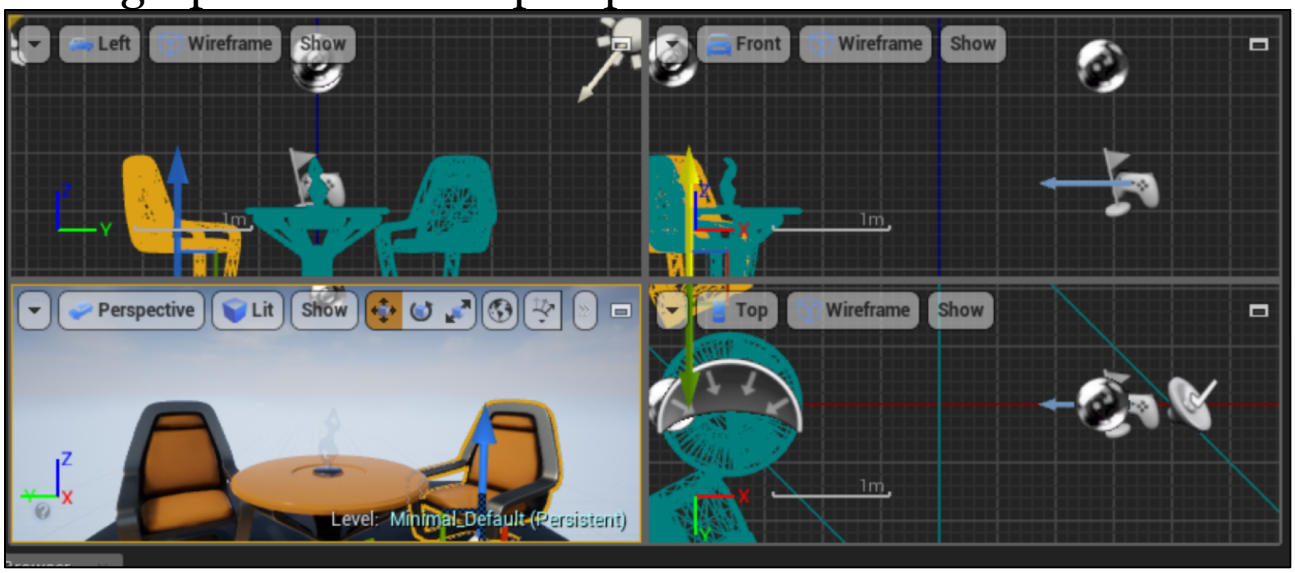
mesh and that it is currently visible. We can also see we have 15 actors within our scene. After we have selected the chair from our **world outliner**. We can see the details of where chairs location is at, the current rotation, and even scale. We can see the static mesh now called **SM_Chair**. There are lot's of other properties within, this window. We can edit, the material, physics, collision, lighting, and etc. The next window we are going to take a look at is the **content browser**.



From the content browser. We can see everything that is within our project. Right now all we have is the **starter content**. Finally we will take a look at our window with the scene in it.

In this window we navigate our scene, place objects, and basically build our game. We can tell that we our in a perspective view. Clicking on the button that states **perspective** will allow us to change our view. We can also

see we have a button that states **lit**. We can use this to change how our scene looks. Under **show** it allows us to select what we want to see within our scene. The next buttons, allow us to choose wether we want to **translate**, **rotate**, or even **scale** our objects. We then have a button that looks like a **world**. This allow us to choose wether we want to move our objects in world space, or local space. The next three buttons allow us to change how many increments we **translate, rotate,** or **scale** our objects. we also have a **camera speed** icon that allows us to determine how fast our camera moves. Finally we have a little box that changes our scene from one big perspective scene to four small scenes consisting of orthographic views and perspective.

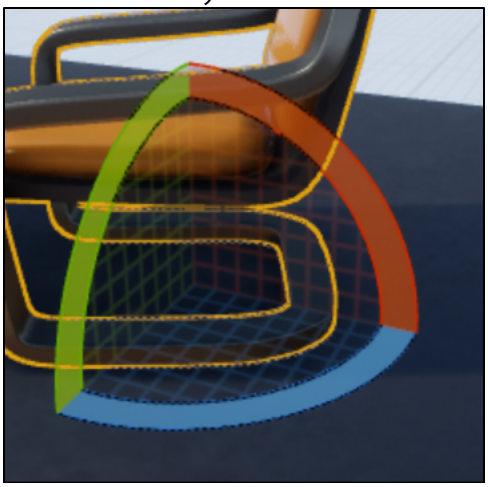


Unreal Engine| Navigation

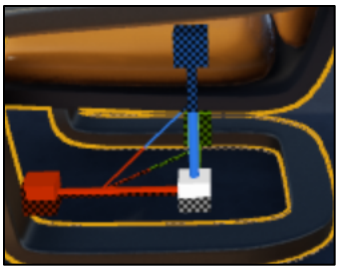
We can navigate our viewport a few different ways. Holding down the right click mouse button and pressing **W(forward),S(Backward),A(Left),D(right)**. Will move you forward, backward, left and right. Holding down the **right mouse button** and moving the mouse will allow us to rotate. We can also zoom in and out holding down alt and the

right mouse button, and then moving the mouse. Same is true with the left mouse button. Another way to move around in our scene is by pressing the forward key, to move forward, the left key, to move left, the down key, to move backwards, and finally, the right key, to move right. If we select a object in our scene and press f, we can focus in on the object. Holding down alt and the left mouse button will allow us to rotate around the object. Alt and the right mouse button will allow us to zoom in on the object. Finally to move objects in our scene. we will notice a **blue, green, and red transform tool**.

The translate transform tool. By selecting the green we will move the object along the y axis. The red will move along the x axis, and the blue will move the object along the z axis.



The rotate transform tool allows us to rotate along the axis.



The scale transform tool allow us to scale along the axis.

Unreal Engine | Summary

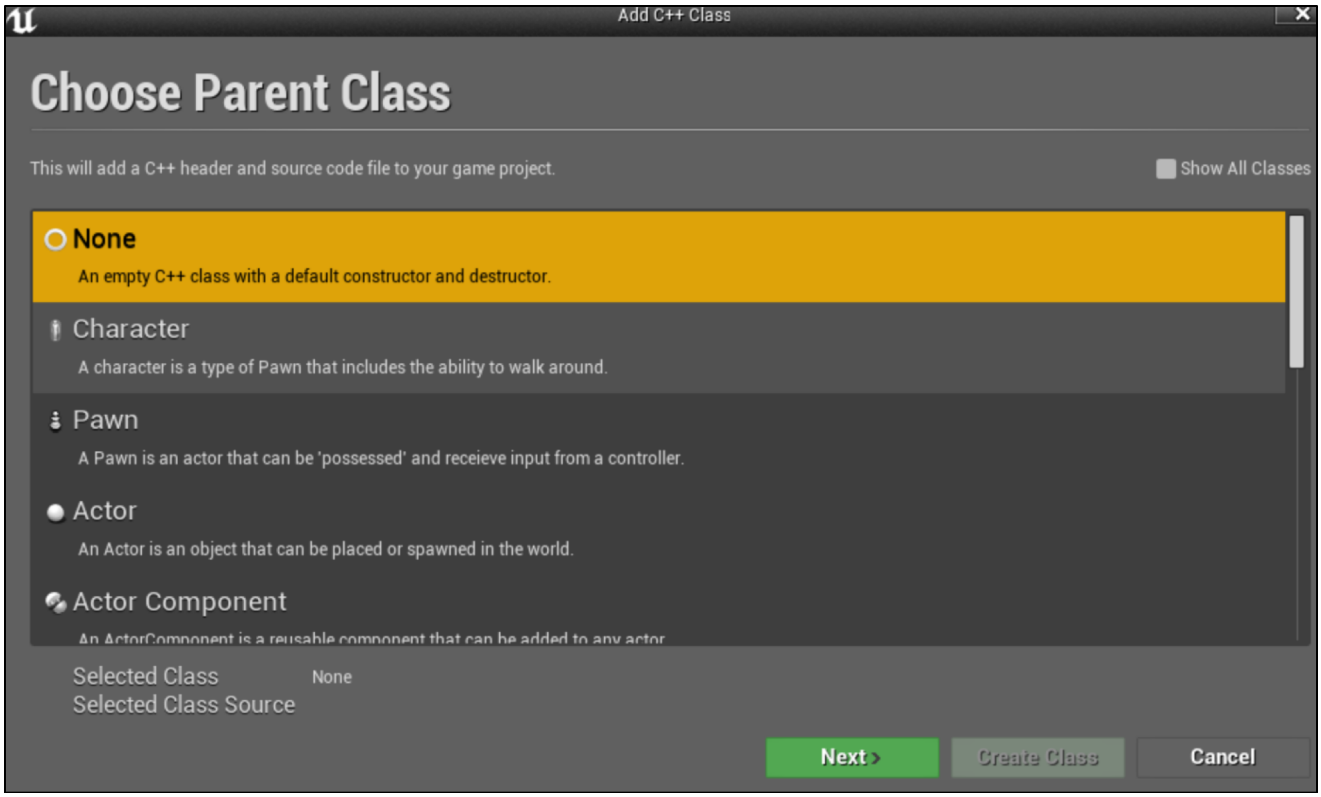
This was a basic overview of the unreal engine interface, in our next section we will start discussing how we create a code file, and what exactly is inside a code file.

Chapter 7

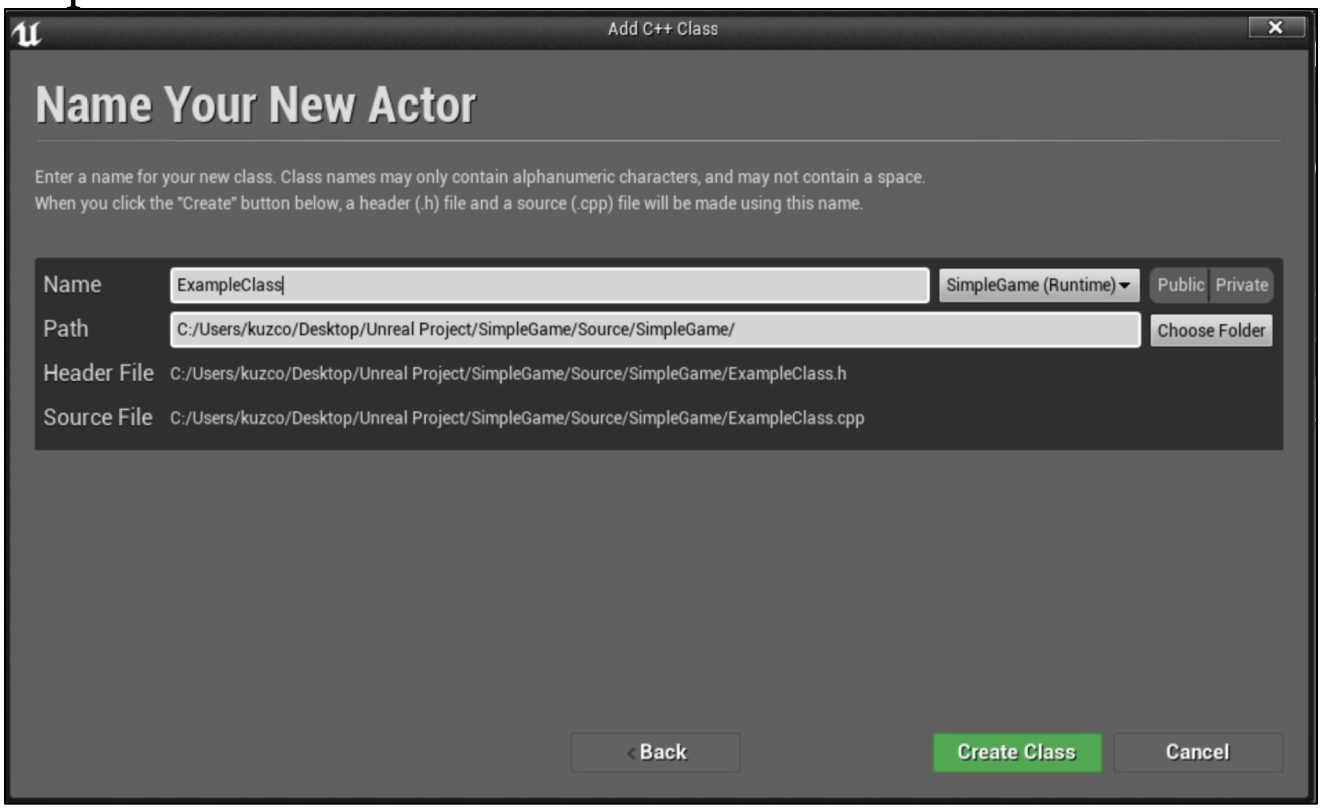
Introduction to Unreal Engine Coding

Unreal Engine | Creating A Class

To create a code class we can go to the **content browser > right click > new c++ class**. We can also create a new c++ class by going to **File > New c++ class**. Once we have done this we will get a screen that looks like this:



This allows us to choose a class we wish to inherit from. For now we can create a **actor** inherited class. We will then be provided with a screen that looks like this:



We will name our class **ExampleClass**. We will put it in our folder then we press the **create class** button. It will create the class and we will be greeted by visual studio.

Unreal Engine |Overview of the class

When visual studio loads up we can tell that we are within a header file called **ExampleClass**. We can also tell a whole lot of code was generated. Lets take a look at what the header file does.

ExampleClass.h

```
// Fill out your copyright notice in the Description page
of Project Settings.

#pragma once

#include "GameFramework/Actor.h"
#include "ExampleClass.generated.h"

UCLASS()
class SIMPLEGAME_API AExampleClass : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AExampleClass();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;
```

```
};
```

The two things we should take away from this class. Is that it has generated most of the code we need. We do not need to worry about the **UCLASS()** macro, this is just telling the engine how to represent the class. Then we have our class that inherits from a **AActor** class. Then it has a macro for **GENERATED_BODY()**. Finally we have a constructor called **AExampleClass()**, a **BeginPlay** function and a **tick** function. The begin play function is called everytime we start the game. While the tick function is called every frame in our game. So essentially we would use the begin play function to initialize variables, etc. While we would use the tick function to update everything that needs to be updated. **Virtual** tells us that we can override this class from other inherited classes. **Protected**, means we are keeping it only public to classes inherited from this class. Lets take a look at the **implementation file**.

ExampleClass.cpp

```
// Fill out your copyright notice in the Description page
of Project Settings.

#include "SimpleGame.h"
#include "ExampleClass.h"


// Sets default values
AExampleClass::AExampleClass()
{
    // Set this actor to call Tick() every frame.  You can
turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
```

```
}

// Called when the game starts or when spawned
void AExampleClass::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AExampleClass::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}
```

From here we can take a look at the constructor class. The first thing line of code within the constructor class. **PrimaryActorTick.bCanEverTick**, is set to true. This will tell us if we want to have this class update every frame. We then have our **BeginPlay** where we can implement what we want to initialize when the object is first spawned. Finally we have our **tick** function this tells us what we want to update every frame.

Unreal Engine |Overview of the class

Hopefully you have a good understanding of how a c++ class is created, and how we would go about working within a unreal engine c++ class.

Chapter 8

The End

The End

This book is a work in progress. It will be updated when the engine changes, or i find errors. Thank you for taking your time to make it through this book. If you have any comment's or criticism, i would be glad to read them. I hope this has given you a good **introduction to c++**, and **unreal engine**. In the next book, i am going to start creating a **simple rpg**, using c++ in unreal engine. For now this was just a simple introduction to unreal engine. I will be getting the chapters up as fast as possible. Again thank you for reading this book, and i hope you come back for book two.

Assignments

Chapter 1 Answers:

1. Get a characters name.

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

#define cout std::cout
using int32 = int;
using FText = std::string;

void main()
{
    FText getUserInput = "";
    cout << "Please input the characters name: ";
    std::getline(std::cin, getUserInput);
    cout << "The characters name is: " << getUserInput << ".";
}
```

2: Simple Experience Calculator.

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

 #define cout std::cout
 using int32 = int;
 using FText = std::string;

void main()
{
    int getUserInput = 0;
    cout << "Please input the characters level: ";
    std::cin >> getUserInput;
    int experienceNeeded = 50;
    getUserInput = getUserInput * experienceNeeded;
    cout << "The character needs: " << getUserInput << ".";
}
```

3: Define a macro.

#define cin std::cin

Chapter 2 Assignments:

1:Create a Level up function

```cpp
void LevelUp(int32 EXP);
void main()
{
    int32 experience = 0;
    cout << "How much experience would you like to give the character: ";
    cin >> experience;
    LevelUp(experience);
    return;
}
void LevelUp(int32 EXP)
{
    int EXPRequired = 50;
    int level = 1;
    while (EXP >= EXPRequired)
    {
        level++;
        EXP -= EXPRequired;
        EXPRequired *= level;
    }
    cout << "Your current level is: " << level;
    cin >> level;
}
```

2: Battle Menu

```
FText DisplayBattleMenu();
bool BattleMenuOptions(FText option);
void main()
{
    //Creating a battle menu.
    bool bIsMenuActive = true;
    while (bIsMenuActive)
    {
        bIsMenuActive = BattleMenuOptions(DisplayBattleMenu());
    }
    return;
}

FText DisplayBattleMenu()
{
    FText userOption = "";
    cout << "Battle Menu:\n";
    cout << "1: Attack\n";
    cout << "2: Run Away\n";
    cout << "Choose a option: ";
    std::getline(cin, userOption);
    cout << std::endl;
    return userOption;
}
```

```
bool BattleMenuOptions(FText option)
{
    switch (option[0])
    {
    case '1':
        cout << "You have attacked the enemy.\n";
        return true;
        break;
    case '2':
        cout << "You have chosen to run away.\n";
        return false;
        break;
    default:
        cout << "That option is not avaible.\n";
        return true;
        break;
    }
}
```

3: Potion function

```
FText DisplayPotionMenu();
bool PotionMenuOptions(FText option);
void healHealth();
void healMana();
void displayCurrentHealthAndMana();
int32 currentHealth = 0;
int32 currentMana = 0;
void main()
{
    //Creating a battle menu.
    bool bIsMenuActive = true;
    while (bIsMenuActive)
    {
        bIsMenuActive = PotionMenuOptions(DisplayPotionMenu());
    }
    return;
}
```

```
FText DisplayPotionMenu()
{
    FText userOption = "";
    cout << "Potion Menu:\n";
    cout << "1: Health Potion\n";
    cout << "2: Mana Potion\n";
    cout << "3: Exit.\n";
    cout << "Choose a potion: ";
    std::getline(cin, userOption);
    cout << std::endl;
    return userOption;
}
```

```
bool PotionMenuOptions(FText option)
{
    switch (option[0])
    {
    case '1':
        cout << "You have selected a health potion.\n";
        healHealth();
        displayCurrentHealthAndMana();
        return true;
        break;
    case '2':
        cout << "You have chosen a mana potion.\n";
        healMana();
        displayCurrentHealthAndMana();
        return true;
        break;
    case 3:
        cout << "You have decided to exit.\n";
        return false;
        break;
    default:
        cout << "That option is not avaible.\n";
        return true;
        break;
    }
}
```

```
void healHealth()
{
    currentHealth += 10;
}

void healMana()
{
    currentMana += 10;
}

void displayCurrentHealthAndMana()
{
    cout << "Your health: " << currentHealth << " Your mana: " << currentMana;
    cout << std::endl;
}
```