



**Game Engine Creation**

COSE40638

Student Name:

Student Number:

**Portfolio**

Contents

Chapter 1: Introduction

Chapter 2: Data Types

Chapter 3: Variables

Chapter 4: Operators

Chapter 5: Conditionals

Chapter 6: Loops

Chapter 7: Functions

Chapter 8: Arrays and vectors

Chapter 9: Strings

Chapter 10: Debugging

Chapter 11: Pointers

Chapter 12: References

Chapter 13: File Handling

Chapter 14: Object Orientated Programming – Structs

Chapter 15 OOP - Classes

Chapter 1: Introduction

To begin we will be going through the steps required to build a simple application in C++ using Visual Studio. This will be the traditional ‘Hello World’ program written by all new coders. This will go through the steps required to create a new program and how to add source files. If you followed along with the code-along then this section will just need a screenshot of your output.

NOTE: There is one function in the HelloWorld program. It is called main(). All programs must have a main function as this is where execution of the program begins. There can only be 1 main function in any given program.

Firstly, create a folder in your OneDrive and call it GitHub. Inside this create another folder called GEC. It is recommended you add folders for each week to make finding work easier, but this can be done later as needed.

**Program 1: Hello World**

1. To begin, start Visual Studio and you will be met with this window…

1. Click Create a new project and select the C++ Empty Project and click **Next**.
2. Name your Project HelloWorld and save to the location of your GEC folder within OneDrive that you created earlier, then click **Create**.
3. You should have the following window open. Right click the folder **Source Files** -> **Add** -> **New Item**. The default can be left as it is, it should have C++ File selected with the name **Source.cpp**. Click **Add** 
4. Now for some code! Replicate the following code: 
5. The first line of code adds the input/ output streams library that contains the cout object that allows us to print to screen. As mentioned above, every program needs a main function as a point of entry. As you can see this function is of type **int**, this means that the function expects a return type of type int, as seen on line 7 where we return 0. This return is reached only if the function is completed. On line 5 we start with **std::** this allows us to use the objects with the std namespace of the standard library, the use of **std::** can be omitted if **using namespace std;** is added below our include, but for now lets use best practice and include them. After cout we have two left chevrons << when outputting these will face left, putting what is on the right of them to the screen. If we were taking information in from a user, they would face right. We will cover this later. We then end with **endl;** which is the same if we were to add \n for a new line and of course the semicolon which is needed to end a line of code. Keep an eye out for these as they are a common result for program errors from new coders.
6. Now run your program using the green play button for Local Windows Debugger or F5. If you have followed along correctly you will be met with a console window containing “Hello World!”.
7. Screenshot this and add it below.

**Program 1 Screenshot:**

A computer screen capture

Description automatically generated with medium confidence

Chapter 2: Data Types

Before we can create variable to store our data, we need to learn what a data type is and how they work. There are a variety of built-in data types that we can use immediately, but it is also possible to create your own.

This table details the data types that are found in C++ and you will be using throughout your studies.

|  |  |
| --- | --- |
| **Type** | **Keyword** |
| Boolean | bool |
| Character | char |
| Integer | int |
| Floating point | float |
| Double floating point | double |
| Valueless | void |
| Wide character | wchar\_t |

Table 2.1: Data types

The keyword is how this type is written in C++. If you make a mistake typing this in, for example miss a letter or use an uppercase character where it should be lowercase you will be alerted to a syntax error.

Not: Syntax is the term used for how the compiler expects to receive the code. If something does not match, you will receive a syntax error.

**Data Type Modifiers**

Each type reserves a certain amount of memory from the computer. In some situations, you may find that you require the memory to be increased / decreased or you would like the value stored in a variable to be in a different range. To enable this there are data type modifiers.

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| signed | Use values in the negative and positive range. |
| unsigned | Use only the positive range. |
| short | Decrease the range, thus reducing the memory used. |
| long | Increase the range, which increases the memory used. |

Table 2.2: Data type modifiers

The following table details the data type, the amount of memory used and the range of values that can be stored in a variable of this type.

|  |  |  |
| --- | --- | --- |
| **Type** | **Typical Bit Width** | **Typical Range** |
| char | 1 byte | -127 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -127 to 127 |
| int | 4 bytes | -2,147,483,647 to 2,147,483,647 |
| unsigned int | 4 bytes | 0 to 4,294,967,295 |
| signed int | 4 bytes | -2,147,483,647 to 2,147,483,647 |
| short int | 2 bytes | -32,768 to 32,768 |
| unsigned short int | range | 0 to 65535 |
| signed short int | range | -32,768 to 32,768 |
| long int | 4 bytes | -2,147,483,647 to 2,147,483,647 |
| unsigned long int | 4 bytes | 0 to 4,294,967,295 |
| signed long int | 4 bytes | -2,147,483,647 to 2,147,483,647 |
| float | 4 bytes | +/- 3.4e +/- 38 (7 digits) |
| double | 8 bytes | +/- 1.7e +/- 308 (15 digits) |
| long double | 8 bytes | +/- 1.7e +/- 308 (15 digits) |
| wchar\_t | 2 or 4 bytes | 1 wide character |

Table 2.3: Data type memory and range

Note: We can use a built in function called **sizeof( type )** which takes one of the types listed above and returns the number of bytes in memory used.

**Program 2: Data Type Size**

1. To begin, start Visual Studio.
2. Create a new project but this time select **Console Application**, this will populate the basics for us such as the main function etc. Name this project “Program2\_dataTypeSize” and ensure you are saving to the right location.
3. You can clear the getting started comments should you wish and delete the Hello World code line. Then proceed to replicate the following:Text

   Description automatically generated

**Program 2 Screenshot:**

A screenshot of a computer

Description automatically generated with medium confidence

**Declaring User Data Types**

As mentioned above as well as the in-built types, C++ allows you to create your own data types by using the keyword **typedef**. The reason for doing is purely for the coder benefit and ease of reading. To the compiler your data type is no different form the original. The format for creating your own data type is as follows:

Graphical user interface, application

Description automatically generated

Note: this does not have to be int, this is just what is used for this example. Any data type from table 2.3 can be used.

Now instead of using the keyword **int** we could use our own defined type **date** like so:

Graphical user interface, text, application, website

Description automatically generated

The above code is exactly the same as using int currentDate;

If, however you required a data type to represent several values we can create an Enumeration. This allows you to create a new type and assign the constant value it represents. It follows this format:Graphical user interface, text

Description automatically generated

It will become clearer if we use an example. So, sticking with the date idea, in this example we would rather use the terms JANUARY, FEBRUARY, MARCH, etc to represent the month. We could create our own typedef month as shown above, but then in code we will always be using integer values. This is not very readable, so instead we will use an enumeration:

Text

Description automatically generated with low confidence

Note: Notice the syntax used when creating an enumerated type. After each type, a comma is used except for the last value and the semicolon is used after the curly brace.

As you can see **JANUARY** has been given the value of **1**, and all the following names will be incremented by 1. For example, **FEBRUARY** will hold the value **2**, **MARCH** the value **3**, and so on.

You can set the value of any name, but it is important to remember that the following names will be incremented by 1.

It is also possible to set none of the names to an initial value. In this case the first name will be assigned the value **0**, and the following names incremented in the usual way.

To use this enumerated data type, we create a variable of this type in the exact same way as we do any other.



Chapter 3: Variable Types

A variable is storage in memory that we can access using a name of our choosing. All variables must be of a data type described in the previous chapter. That is because the data type informs the compiler how much memory is to be reserved and what values can be stored. Refer to Table 2.1 for the data types that can be used along with the keywords that must entered for C++ to recognise the type.

When defining a variable, the user chooses what to name it, but there are restrictions. The name can be composed of letters, digits and underscores, but the first character must always be a letter or underscore. It is good practice to make the first letter of each new word uppercase. For example, **thisIsMyExampleVariableName**. This can also be seen in the previous chapter where we named our variables **currentDate** and **currentMonth**.

Also, the name chosen should be something informative. Variables of this sort: **a**, **b**, or **myVariable** do not help in any way, if you are dealing with user’s names call it **userName** or if dealing with ages call it **age**. Also, there may only ever be one variable of this name. duplicating names will produce syntax errors.

One final thing to note is that C++ is case sensitive, so when using your variables throughout your code ensure that they match. You will receive a syntax error if they do not.

The most straight forward way to declare a variable is with the following format:

Graphical user interface

Description automatically generated with low confidence

To assign a value to our variable **myNum** we use the assignment operator =. (Operators are covered in more depth in the next chapter.)

Graphical user interface, application

Description automatically generated

So, at this point wherever you use the variable myNum it will produce the value **5**. You could however choose to assign the value of **5** at the same time as declaring the variable. This reduces the number of lines of code and makes the code easier to read. Bare in mind that you will not always know the starting value, so cannot do this.

So far so good. We have one last addition to make to this defining a variable section before we will get on to a mini program, and this is the use of definition lists. We can define multiple variables on the same line provided they are of the same data type. For example:



Variables can be changed as often as required. There is no special code required for this, you simply reset the variable to hold a different value.

Graphical user interface, application

Description automatically generated

If you require a variable that will never change, then you can use the **const** modifier. This will make the value constant and can never be changed. You may think this unlikely to occur, but it is a feature that you will find useful more often than you may think. It is common to see a lowercase k before the variable name to signify that it is a constant variable throughout the code.

Graphical user interface, text

Description automatically generated

Note: Don’t forget if using float, it is good practice to add the f at the end of your value.

**Program 3: Defining Variables**

1. To start, open Visual Studio.
2. Create a new C++ console application as before and name it Program3\_definingVariables.
3. Next, replicate the following code (ensure when assigning a value to a char type you use single quotation marks).

Text

Description automatically generated

**Program 3 Screenshot:**A screenshot of a computer

Description automatically generated with medium confidence

**Program 4: Defining User-Defined Variables**

This program is for you to complete. Please pay attention to the instructions.

Create a program named Program4\_userDefinedVars, this will contain both a **typedef** and an **enum** type. If you still have Visual Studio open, you can use the shortcut ctrl + shift + n

1. Create a typedef of data type **int** named **health**.
2. Create an enumerated data type named **Weapons**, this list should contain the following weapons in this order.
   1. SWORD
   2. DAGGER
   3. MACE
   4. TWIN\_SWORDS
   5. SAMURAI
   6. WIZARD\_STAFF
   7. FIRE\_POTION
   8. ICE\_BLADE
   9. SMALL\_KNIFE
3. Next create a variable of data type **health** with the value of 13000, and a variable of data type **Weapons** with the value of SAMURAI.

Output the following to screen:  
My current health is: …

And the ID of my weapon of choice is: …

HINT: You will need to pass the variables to the cout statement just as you did with program 3. Output should read:

My current health is: 13000

And the ID of my weapon of choice is: 5

**Program 4 Source code:**

For testing purposes please follow these steps to adding code to this portfolio.

1. Copy the code from within Visual Studio
2. In word click **Insert** and then in the Text field click object
3. In the dropdown popup select OpenDocument Text and click OK
4. Paste your code in the new blank document that opens and close when done. ENSURE CURSOR IS IN THE RIGHT PLACE IN THIS DOCUMENT FOR WHERE YOU WANT YOUR CODE



**Program 4 Screenshot:**

As before this should be a screenshot of your output.

A screenshot of a computer

Description automatically generated with medium confidence

Chapter 4: Operators

Operators are special characters that represent mathematical or logical manipulations. C++ has a range of different operators, and we will be covering the following in this chapter: Mathematical Operators, Assignment Operators, Relational Operators, and Logical Operators.

**Mathematical Operators**

Looking at table 4.1 you will no doubt recognise the first four operators. These represent the usual mathematical operators we use in everyday life. As such we will not be explaining those, instead we will be focusing on the unfamiliar symbols.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Adds two operands |
| - | Subtracts second operand from first |
| \* | Multiplies both operands |
| / | Divides numerator by denominator |
| % | Remainder left after integer division |
| ++ | Increases integer by one |
| -- | Decreases integer by one |

Table 4.1: Mathematical Operators

**Modulus**

The following code snippet creates two integer variables which each store a number. The final line of code creates an integer variable named remainder which holds the remaining digits from a & b divided by a calculation.

Graphical user interface, text, application

Description automatically generated

The variable remainder will be equal to 1 as 3 goes into 10 3 times with the remainder of 1.

**Increment Operator**

The increment operator adds 1 to the current value stored in the variable. This is simple enough and will be shown below. Confusion can arise when the increment operator is placed before the variable.

The following code snippet assumes the existence of an integer variable named num, which stores the value 1. After the following line of code, num will now be equal to 2.

A picture containing text, meter, device

Description automatically generated

As stated above, the increment operator can be placed before the variable like so: ++num. This has the effect of changing the value stored in the variable in the same manner as before. But the effect takes place immediately. Assuming num started with a value of 1, in the above line of code, num will equal 2 as soon as it is reached, not on the following line.

Note: The placement of the Increment Operator may seem trivial or confusing now, but once we get to using loops in chapter 6 the placement will be crucial.

**Decrement Operator**

The decrement operator subtracts 1 from the current value stored in the variable. It works very much like the increment operator.

Again, the following code snippet assumes the existence of an integer variable named num, which stores the value 1. After the following line of code, num will now be equal to 0.

Graphical user interface, application

Description automatically generated

Just as with the increment operator, decrement can be placed before the variable. This has the effect of changing the value stored in the variable in the same manner as before. Assuming num started with a value of 1, in the above line of code, num will equal 0 as soon as it is reached, not on the following line.

**Program 5: Increment / Decrement Operators**

1. If you haven’t already open VS and start a new project called Project5\_IncrementDecrment
2. Replicate the following code.

Note: It is a good idea to comment your code. This means to add comments, which are only there for the programmer’s benefit. The compiler ignores them. This may seem trivial, but it really helps other programmers who may use your code, or even yourself if you have not looked at a program in a while. To comment you can either use a double slash // or surround the text with a /\* \*/ block. Both are demonstrated in the following code.

Text

Description automatically generated

**Program 5 Screenshot:**

A screenshot of a computer

Description automatically generated with medium confidence

**Program 6: Area of a Rectangle**

This program is for you to complete. Please pay attention to the instructions.

1. Create a program named Program6\_area, which creates variables of integer data types for **width** and **height**. Initialise **width** to the value of 15 and the **height** variable to the value of 25.
2. Next create an integer variable called **area** and set this to equal the width times by the height.
3. The output should be in the following format:

**Program 6 Source code:**

As before, please ensure to copy your code via the insert object format.



**Program 6 Screenshot:**

A screenshot of a computer

Description automatically generated with medium confidence

**Program 7: Converting from Fahrenheit to Celsius**

This program is for you to complete. Please pay attention to the instructions.

1. Create a program called Program7\_converting, in which you are required to write code that will convert Fahrenheit into Celsius
2. Create two floating-point variables named **fahrenheit** and **celsius**, set the value of Fahrenheit to equal 95.0f.
3. To calculate the Celsius, it must equal this formula: (Fahrenheit - 32) \* 0.5556
4. Output the results in the following format:



**Program 7 Source code:**

As before, please ensure to copy your code via the insert object format.



**Program 7 Screenshot:**

A screenshot of a computer

Description automatically generated with medium confidence

**Program 8: Area of a Circle**

This program is for you to complete. Please pay attention to the instructions.

1. Create a program called Program8\_pi, in this program we will calculate the area of a circle.
2. Create a constant floating-point variable named **pi**, this will hold the value 3.14159.
3. Next create two float variables, one called **area** and one called **radius.** Set the value of **radius** to equal **25.0f**. And set the value of **area** to equal this formula: **pi** \* **radius** \* **radius.**
4. Output the result in the following format:



**Program 8 Source code******

**Program 8 Screenshot:**

A screenshot of a computer

Description automatically generated with medium confidence

**Assignment Operators**

Now that we understand the mathematical operators, we can move on to assignment operators. These operators simply reduce the amount of code required to do a simple operation. Look at Table 4.2. Examples of each operator will follow below.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Assigns value from the right-side operand to the left side operand. |
| += | Adds right operand to the left operand and assigns the result to the left operand. |
| -= | Subtracts right operand to the left operand and assigns the result to the left operand. |
| \*= | Multiplies the right and left operands together and assigns the result to the left operand. |
| /= | Divides the left operand by the right operand and assigns the result to the left operand. |
| %= | Takes the modulus of two operands and assigns the result to the left operand. |

Table 4.2: Assignment Operators

**Equals Operator**

As has been shown previously the equals operator is used to assign a value to a variable. A variable must always be on the left-hand side of these operations. The following snippet assigns the value of 5 to the variable num.



You can also assign a variable to equal the value stored in a completely different variable provided they are of the same data type, like so:

Graphical user interface, text

Description automatically generated

**Calculate and assign**

A screen shot of a computer

Description automatically generated with medium confidence

**Rational Operators**

Relational Operators are used to return a result of true or false. These will be used all the time in conjunction with conditionals [chapter 5 next week]. Look at Table 4.3 and the corresponding examples below for an explanation of each operator.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | Checks if the values of the two operands are equal, if so, the condition becomes true. |
| != | Checks if the values of the two operands are not equal, if they are not equal then the condition becomes true. |
| > | Checks if the left operand is greater than the right operand, if so, the condition becomes true. |
| < | Checks if the left operand is less than the right operand, if so, the condition becomes true. |
| >= | Checks if the left operand is greater than or equal to the right operand, if so, the condition becomes true. |
| <= | Checks if the left operand is less than or equal to the right operand, if so, the condition becomes true. |

Table 4.3: Relational Operators

**Examples of Rational Operators**

If the variables being compared store the same value the result returned will be true, otherwise it will return false. Take notice of how a single = sign is for assignment and the double == is for comparison.

Text

Description automatically generated

**Logical Operators**

Logical Operators are used to return a result of true or false. Just like Relational Operators, these will be used all the time in conjunction with conditionals next week. Look at Table 4.4 and the corresponding examples below for an explanation of each operator.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| && | AND Operator  If both operands are non-zero [true], then the condition becomes true. |
| || | OR Operator  If any of the two operands is non-zero [true], then condition becomes true. |
| ! | NOT Operator  Used to reverse the logical state of an operand. If a condition is true, the NOT Operator will make it false. |

Table 4.4: Logical Operators

**Examples of Logical Operators**

Logical AND checks if both variables being checked are true the result will be true, otherwise false will be returned. Whereas the logical OR checks if either of the variables being checked are true. If one of them is true then the result returned will be true, otherwise false will be returned.

Text

Description automatically generated

The Logical NOT operator will reverse the result from the other two logical operators. This may seem a redundant operation, but using the ! (NOT) operator will come in useful. The following two examples are the exact same as the examples used for Logical AND except that we have included the ! (NOT) operator. Notice how the results have been reversed.

A screenshot of a computer

Description automatically generated with medium confidence

Chapter 7: Functions

Functions are a programming approach to reduce your codebase and stop the need to replicate the same code in multiple places. If approaching C++ from another language, you may be familiar with the term methods, which are very similar to functions, however; C++ standard does not use methods per se but slightly different variations of functions. The C++ equivalent being of a method being a *member function*. The only real difference is a function, or a *free function* is called by name anywhere in the code, and *a member function/method* is associated with an object. Each function is usually a block of code that does a single task. We have already seen one function that has been in every program we have written so far – the main() function.

Functions can take any number of parameters and can return any valid type. This includes void types which signifies that nothing is returned. When declaring functions various approaches can be taken.

• The first is to put the function ahead of the main() function in code. This will ensure the compiler knows of its existence before the main() function attempts to use it.

• The second, and preferred, is to use a prototype, which is placed ahead of the main() function and then add the function body below the main() function. The prototype gives the compiler all the information it requires to allow the main() function to use the function within its own body of code.

• The final approach is to use header files and source files. In the header the prototypes are declared, and in the source file the body of each function is added. You need to include the header file at the top of the source file that uses the functions contained within. We will be using this approach when we get to Chapter on Object Oriented Programming.

**Format of a function:**

**Text

Description automatically generated**

Note: Parameter is just another name for variables that are passed to the function.

The format of the prototype matches the function detailed above, but it ends with a semi-colon rather than the curly braces and code body. This is because it is only used to inform the compiler what the function name is, what the return type is and what the parameters are.

The format of a prototype is as follows:



It is important to note that any variables passed into a function as a parameter will be copied for use within the function. These are placed on the stack, and when the function returns the copies will be deleted. This is important to know because it means any variable passed into a function can is not changed within the function. We can solve this issue using pointers & references, but this will not be covered later weeks with Pointers and References.

**Example functions**

In all examples the programmer is expected to add the correct namespace at the top of the file or use the correct syntax.

**I love Coding Function**

Firstly, we must add the function prototype. This goes above the int main() function:



Next, we will add our function body. This will go under the main function after the last closing bracket.

Note: As we progress, we will be using header files and classes to construct our code and typically the main function and cpp file will be but a few lines of code.

A picture containing logo

Description automatically generated

Finally, we add code to the main function that uses our new function:

Text

Description automatically generated with medium confidence

Build and run your program to make sure everything has been done correctly.

**Passing Parameters**

This example will create a function that takes two integers, adds them together and outputs them to the screen. It does not require a return type, so it will be void. We will create some local integers to be passed in, we will pass through some constant integers and finally pass through a mixture of both. Note that the parameter name does not need to be the same as the integer name passed in.

First, add the function prototype at the top:



Next, add the function under the main like so:

A screenshot of a computer

Description automatically generated with medium confidence

Notice there is no return keyword. This is due to it being a void function and will automatically return at the end of the code.

And finally, we add our code calling the function to the main. We will do a couple of examples:

Text

Description automatically generated

Build and run your program and make sure there are no errors. If there is anything you don’t understand about how the function is working now is the time to speak to a lecturer.

**Return Values**

This example will create a function that takes two integers, adds them together and returns the result. The result is then output to the console.

Add the prototype:



Next, add the function body:

Text

Description automatically generated

And finally, add our code that calls the function in the main:

Text

Description automatically generated

**Program 17: Local Variables & Copies**

This program will demonstrate how variables do not get affected outside of functions regardless of what you do to the local copies a function makes.

1. Open Visual Studio
2. Create a new project called Program17\_LocalVariablesAndCopies
3. Create a function prototype called addTwo which takes one parameter of type int called num.
4. Add the function body below the main like so:  
   Text

   Description automatically generated
5. Now for the code within main:
   1. Create an int variable called myNum with the value of 6.
   2. Output to screen the value of myNum before the calling of the function  
      
   3. Call the function, passing myNum:  
      
   4. Now repeat step b but [after] the function call.
6. Build and run the program. If all is correct your outputs should be 6, 8, 6. The value myNum has not been changed but rather a copy of its value is used within the function scope.

**Program 17 Source code:**

****

**Program 17 Screenshot:**

Text

Description automatically generated

**Program 18: Smallest Number**

Write a program which asks the user to input two integers, three times. These values should then be passed into a function to determine which is the smallest. The function does **not** need to **return** the smallest value but should print it to the console. Then once that loop is complete, the program should ask three more times for a number to be multiplied by 10 and then divided by 3. The multiplication should take place in a function and **returned,** when returned, store the result in a float this is then passed to another function that divides the parameter by three. This dividing function does not need to return but merely print to console.

This is the function prototype you must use for SmallestNumber is below:



The multiplication function should return an int and expect an int as a parameter. The dividing function should **not** return anything but should expect a float as a parameter. These are you to workout.

Things to consider:

* Creating variables to hold user input
* Not sure how to test which is bigger? Look back at week 1 Conditionals
* Have your program ask for the three sets of numbers before exiting. Look back at week 3 Loops.
* How will you deal with numbers of the same value?

Note: This is a function exercise and the result MUST be determined within the function body and then returned from the function. Use the values below.

Run your program using the following values. The screenshot must show the use of these elements for the first loop:

1, 2

2, 1

5, 5

And use the digits 2, 6, and 4 in the second loop. The output should look like this:  
Text

Description automatically generated

**Program 18 Source Code:**

****

**Program 18 Screenshot:**

Text

Description automatically generated

**Program 19: Odds and Evens**

Write a program, which asks the user to input 10 integers. These values should be then passed individually to a function, which will determine whether the value is odd or even. This function should return a boolean value.

In the event of the value being odd, it should be added to an odd count, and in the event of it being even it should be added to the even count.

Your program should then call a function which will output how many odd numbers there were and the total, and in a similar manner for the even numbers. Use the following prototype for the output function:



Note: This is a function exercise and MUST demonstrate the use of functions as detailed above

**Things to consider:**

It might be useful to write some pseudo code in a notepad doc to plan out what the program needs,

* What local variables are needed?
* How will you ask for the 10 numbers?
* When will you call the Boolean is odd or even function?
* When will you call the outputResults function and what variables do you need to pass to it?
* The output should tell the user how many odd numbers were entered followed by the grand total of the odd numbers sum and the same with the even numbers.

**Program 19 Source code:**

****

**Program 19 Screenshot:**

Text

Description automatically generated

**Program 20: Simple Text Battle**

Below you have been provided with some simple pseudo code for you to work from. The aim is to recreate the program in working code using everything you have covered in lectures and tutorials so far. To keep things simple while you get to grips with functions you may use the listed global variables. This program is a good one to revisit when you have learnt pointers/references and edit it in such a way as to remove the global variables and access the vars from within main and edit them.

namespace

\*Global variables\*

two integers for player and enemy health set to 1000(p) and 2000(e)

boolean variable for playing set to true

\*Function prototypes\*

attack choice that returns nothing. Parameters required: one integer for choice

play state function that returns a boolean

\*Main Function\*

local variable to hold the players choice

A loop to run as long as playing is true

tell user they have met a troll and have 3 options of attack

1. use sword, 2. use magic, 3. use axe

tell user choices have different effects

store users choice

call attack choice function passing user choice

call play state function

return 0;

\*attack choice function\*

set constant ints for the passed user choice, sword damage = 300, magic\_damage = 650, axe\_damage = 450,

troll\_sword = 350, troll\_magic = 50, troll\_axe = 100;

switch statement(condition user choice)

case 1:

set enemy health to take sword damage

set player health to take troll sword damage

tell user they have hit the troll

tell user they have been hit

if player or enemy health is less than 0 set to 0 (prevents negative health)

tell user current player and enemy health

case 2:

set enemy health to take magic damage

set player health to take troll magic damage

tell user they have hit the troll

tell user they have been hit

if player or enemy health is less than 0 set to 0 (prevents negative health)

tell user current player and enemy health

case 3:

set enemy health to take axe damage

set player health to take troll axe damage

tell user they have hit the troll

tell user they have been hit

if player or enemy health is less than 0 set to 0 (prevents negative health)

tell user current player and enemy health

Any questions don’t hesitate to ask. Can you think of any improvements that could be made? Maybe check for wrong input etc? Ensure to full test your program and show this in your screenshots.

\*play state function\*

create a char for play again option

if enemy health is less than or equal 0

tell user they have killed the troll and won

ask to play again y/n

store choice

if they want to play again

reset enemy and player health

return playing true

else

return playing false

if player health less than or equal 0

tell user they have been killed

ask to play again y/n

store choice

if they want to play again

reset enemy and player health

return playing true

else

return playing false

return false;

**Program 20 Source Code:**

**Program 20 Screenshots:**