



**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

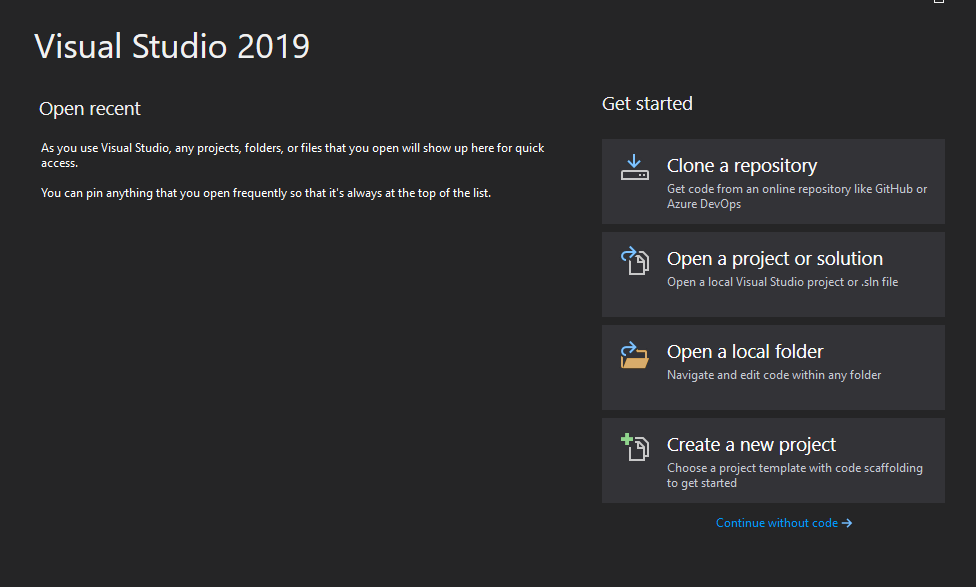
Chapter 1: Introduction

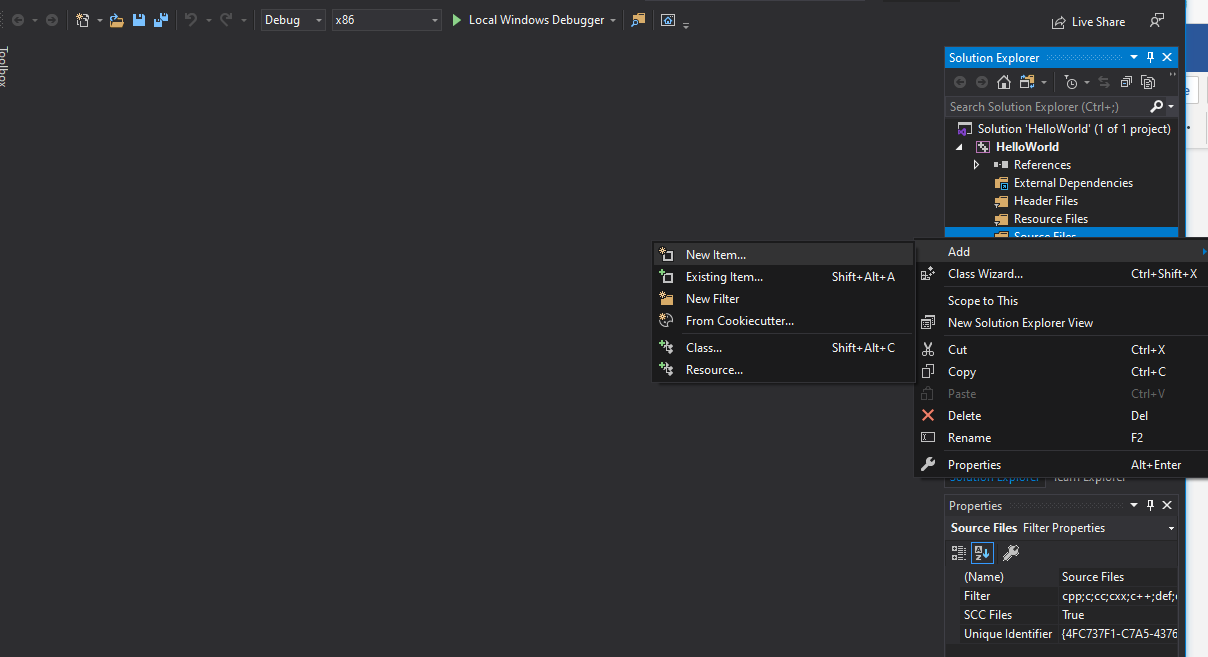
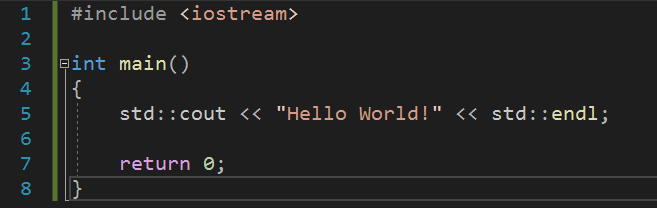
To begin we will be going through the steps required to build a simple application in C++ using Visual Studio. This will be the traditional ‘Hello World’ program written by all new coders. This will go through the steps required to create a new program and how to add source files. If you followed along 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:**

Text

Description automatically generated

Chapter 8: Arrays and Vectors

An array is a data structure that holds a fixed number of elements of the same data type. We can create an array named myNumbers, which holds n integers. All elements of the array can be accessed using the variable name myNumbers. This approach saves the programmer creating multiple individual variables for the same data. The array data structure also works well with loops.

The way this works behind the scenes is with the use of memory addresses. Arrays are nothing more than pointers and can be accessed in the same way. Do not worry about pointers for the moment, they will be covered in a later chapter. It is just something to bear in mind.

**One-Dimensional Arrays**

**Declaring Arrays**

The most straightforward way to declare a variable is the following format:



This creates an array with enough memory to hold size elements. The square brackets are important here.



**Initialising Arrays**

To set the value of an element in the array you follow the same process as any other variable. The only difference being that you need to inform the compiler which element you want to access. This is done using the square brackets again.

Text

Description automatically generated

Note: When accessing elements of an array the positions run from 0 to size -1. So, in the above example when accessing array elements, they run from 0 to 9. 0 will give you the first element, 1 will give you the second and so on.

Also, if you use an element position that is outside of the range, you will get an out of bounds runtime error.

If you know the values of the elements, then you can set this up at the declaration stage.



It is also possible to omit the size of the array if you are giving the element values at declaration. In this case an array of the required size will be created.



**Example array**

If you missed the lecture session, then please replicate the following program. This program will create an array and populate the elements. It will then output the array element and the value stored in a structured way using the setw() function. This function allows you to set how many characters to skip before moving onto the next output. It stands for set width and takes an integer as its only parameter.

Text

Description automatically generated

**Multi-Dimensional Arrays**

We can extend the concept of arrays to hold more elements. The most common is a two-dimensional array. You can imagine this as a table. It is entirely possible to create arrays with more dimensions, but this happens infrequently.

**Declaring 2D Arrays**

In much the same way as a single dimensional array is created, two dimensional arrays follow the same format, only now we add an additional size component.

****

This creates an array with enough memory to hold (*rows* x *columns)* elements. To visualise this in a table take a look at Table 7.1: Array as Table.

Table

Description automatically generated

Table 7.1: Array as Table

An example of an array called myNumbers that holds 9 integers in a 5x3 table would look like this:

****

**Initialising 2D Arrays**

To set the value of an element in the array you follow the same process as any other variable. The only difference being that you need to inform the compiler which element you want to access. This is done using the square brackets again. 

Note: Remember when accessing elements of an array the positions run from 0 to size-1. So, in the above example when accessing row 4 it is the 5th row. I.e. 0, 1, 2, 3, 4 is the 5th element when counting from zero.

If you know the values of the elements, then you can set this up at the declaration stage.

Text

Description automatically generated

It is also possible to omit the nested curly braces and declare a two-dimensional array in a single line. This is less intuitive, and I would recommend declaring arrays in the previous format.



**Example of 2D Array**

As with the single array above, if you missed the lecture please replicate the following program.

This program will define and initialise a 2D array and then output the results of the values held at each row and column.

Text

Description automatically generated

**Example of Vectors:**

If you missed the lecture, please replicate the following program to get some familiarity with the use of vectors. We will be using them more next week.

Text

Description automatically generated

**Program 21: Smallest Element**

Write a program, which asks the user to input 10 integers. These values should be stored in an **array** for use later. Once all numbers have been entered, the program should find and output the smallest value and its position in the array.

Note: This is an array exercise and the numbers MUST be stored in an array.

**Things to consider:**

* How will you ask the user for input?
* How will you store the smallest element and its position?
* How will you loop through the array and compare numbers?
* How will you handle array range not being inclusive? I.E to match the user expected position being 1-10 instead of 0-9.
* Nested for loops are one option, you can also use one loop with a conditional inside. Research how to solve the problem if stuck.

**Program 21 Source Code:**

****

**Program 21 Screenshot:**

A computer screen capture

Description automatically generated with medium confidence

**Program 22: Ordered Output.**

Write a simple C++ program that:

• Declares a one-dimensional array to hold 5 ints.

• Asks the user for 5 integers to fill the array.

• Outputs the array in descending order.

• Outputs the array ascending order.

• Outputs the largest element in the array and its position in the array.

**Things to consider:**

* Use a function to handle finding the biggest integer in the list.
* Research the C++ library <algorithm>. Sorting the ints into ascending and descending can be done several ways. A good programmer will always find and use the simplest method. Specifically read up on the **sort** method in the algorithm library.

**Program 22 Source Code:**

****

**Program 22 Screenshot:**

Text

Description automatically generated

**Program 23: Inventory**

For this program you will be given a several starting variables and a complete array containing strings. Your job as the programmer is to inform the user, they have picked up a wizard’s staff and ask them if they would like to swap it for the dagger in their inventory.

The program should have two outcomes:

1. The player chooses to keep the staff.
   1. Update the contents of their inventory accessing the correct element. (Look at indexing)
   2. Output the contents of their inventory to screen. (Loops are best here)
2. The player chooses to leave the staff behind.
   1. Inform the player they have left the item behind
   2. Output the contents of their inventory

**Variables and Array initialisation**

This is one way this could be done, there are numerous ways, choose the method that suits you best, but this an alternative method for initialising as an example.

Text

Description automatically generated

**Program 23 Source Code:**

****

**Program 23 Screenshots (must show both outputs):**

Text

Description automatically generated

Text

Description automatically generated

**Program 24: Matching Pairs (challenging task)**

The player selects 2 cards (one at a time) if they match the player gets a point and the card remain face up. The game continues until all cards have been turned.

Tips

1. Look at the below pseudo code to help guide you as to how to construct this program

2. Use the system(“cls”) function.

3. Use the square brackets as the cards and have a number for selection.

4. Use letters as your card faces.

**Example board:** [1] [2] [3] [4] [5]

[6] [7] [8] [9] [10]

**Example card faces:** [A][B][C][D][E]

**Pseudo Code:**

Create a char array for your 10 board cards A-E twice over mixed up.

create a bool array for cards found and set this to equal false

^Hint: You'll need curly braces

create a variable to hold the attempt count

create two prototype functions for drawing the game board. One

takes no variables the other requires the two guesses to be passed

drawBoard

system("cls"); //this will clear the board of any turns

Tell player attempt count

Create a for loop

if cards found[i]

cout "[" << boardCards[i] << "]"

else

cout "[" << i + 1 << "]"

if i equal 4 split board

\*\*\*Main\*\*\*

create variables for score and a bool for playing

while the bool variable is true

draw the game board

Two ints to hold guess 1 and guess 2

ask for a selection and store it

Call the draw board that requires parameters, pass the guess

ask for second selection

do as above passing second guess

//Check results

Construct a way to check if the guesses match in an if statement

^Hint: boardCards[selectionA -1]

if match score++

cards found [selectionA -1] = true

cards found [B -1] = true

else

no match...

if score is equal to 5, player has won and end game

else attempts to be increased

drawBoard with parameters

system("cls"); //this will clear the board of any turns

Tell player attempt count

Create a for loop

if cards found[i] or i+1 == guess 1 or i+1 == guess 2

cout "[" << boardCards[i] << "]"

else

cout "[" << i + 1 << "]"

if i equal 4 split board

**Program 24 Source Code:**



**Program 24 Screenshots:**

Initialisation:

Text

Description automatically generated

Half way through (Some cards revealed):

Text

Description automatically generated

Win Screen:

Text

Description automatically generated

Chapter 9: Strings

Last week, while creating your inventory array you were given code containing the key word “string”. This week we will be going into more depth about what a string is and how to use them, along with the use of char arrays. Before string was introduced as part of standard C++, the C-style char arrays were the common method used to create sentences.

If you have not yet completed your array work from last week, I suggest you do so before continuing, so you can get to grips with what we are making a little easier.

**Char Array**

The char array approach to strings comes from the C programming language. The way it works is that we create an array of char data type, which is terminated with a null character ‘\0’

To declare a string using this approach we have two options. The first is to categorically state what each character in the array is and add the terminating null character ourselves:

Shape

Description automatically generated

Note: The size of the array is 1 bigger than the size of the string. This is to cater for the null character.

You can of course miss out the size as in array declarations, but can then use the double quotes as shown here:



**Example Char Array**

If you missed the lecture this week, go ahead and replicate the code below to familiarise yourself with how to create char arrays.

Text

Description automatically generated

Within the string header (also known as cstring header), there are a variety of functions, which can be used on null terminated strings. We will cover a few as detailed in Table 9.1: String Functions and in the examples below, but please look at: [http://www.cplusplus.com/reference/cstring/](http://www.cplusplus.com/reference/cstring/%20) for further details.

|  |  |
| --- | --- |
| **Function** | **Description** |
| Strcpy\_s(s1, s2) | Copies string s2 into string s1. |
| Strcat\_s(s1, s2) | Concatenates s2 on to the end of s1. |
| strlen(s1) | Returns the length of s1. |
| strcmp(s1, s2) | Returns 0 is s1 and s2 match.  Returns less than 0 if s1 < s2.  Returns greater than 0 if s1 > s2. |

Table 9.1: String Functions

Note: In all the below examples it is assumed that the program has included the <string> header like so:



Note: As Visual Studio is updated often by Microsoft, functions can become depreciated. If you get a warning of a depreciated function, you will need to investigate how to use the replacement.

**Example of Copying Character Strings**

To copy the contents from one char array to another you can use the strcpy() function. The first char array parameter passed in will be the one copied to; the second char array passed in is the one to be duplicated.

Text

Description automatically generated

In this example we copied the entire string. It is possible to copy a portion of the string using the strncpy\_s() function.

**Concatenating Character Strings**

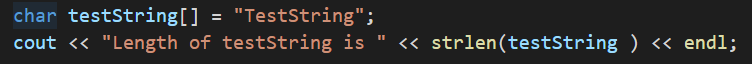
To copy the contents from one char array and attach it to the end of another you can use the strcat() function. The first char array parameter passed in will be the one copied to, the second char array passed in is the one to be duplicated.

Text

Description automatically generated

**Character String Length**

To determine the length of a string there is a function for that too. Use the strlen() function as shown below:



**Comparing Strings**

To determine whether two char arrays match simply pass the two strings through as parameters to the strcmp() function. This will only return a result of 0 if they are the same. Any other result means they do not match.

Text

Description automatically generated

**Program 25: Initials**

Write a program to prompt the user to enter in a single line their first name initial, followed by a space, their middle name initial followed by a space and the entirety of their surname. Store this in a char array.

The program should then output the first initial on one line, then the middle initial on a separate line and the surname on a line of its own.

Note: This program is to demonstrate the use of char arrays. Ensure that you complete this program using a char array and that the data input by the user is stored in a single char array. You can separate out the various parts of the name into separate arrays afterwards, but the initial read from the console should put the entirety of the input into a single char array.

**Tips**

* You will need to loop through your characters to find the first occurrence of a space. This should give you the information you require to be able to access the middle name initial.
* You will need to use **cin.getline** instead of **cin**, if cin is used then only characters leading up to the first space will be stored. To use it with a char array you call the function, and as parameters pass the array name, then size. Like so**: get.line(name, 50);** You can read up on this function here:

<http://cplusplus.com/reference/istream/istream/getline/>

**Program 25 Source Code:**

****

**Program 25 Screenshot:**

Text

Description automatically generated

**The String Approach**

The string approach is an alternative way of dealing with a string of characters. Using string you are in fact using an underlying class, which is found in the string header. You must include this header at the top of any source file that uses string. String introduces a host of new functions, some of which are described below, others that you will need to research on your own.

So, to create a variable of string type it is as simple as:

**Text

Description automatically generated**

Note: You MUST now use double quotes. Single quotes will create a syntax error.

When using a string type with the console, unlike the char array where you pass the array name and size to the getline function, you instead pass (cin, stringName) like so:



As before by just using cin alone, only characters that come before the first whitespace will be stored.

**Example of Input/Output with Strings**

Create a program and add the following code, run it twice to see what happens with cin and getline.

**Text

Description automatically generated**

Note: If you were to create a program where you use cin to take in one word followed by getline to take in numerous words; you would be faced with getline skipping input. This is due to the new line or **endl** not being flushed and counting as an input. There is a number of ways to fix this, with one being to use cin.ignore before getline. You can express what it is to be ignored by the console as a function parameter. The most common being: 

**Example of String Concatenation**

There are a couple of approaches to combining string objects. The first to be shown is the append() function, the second is using operators that have been overloaded such as + and +=.

**Text

Description automatically generated**

The above example will result in string1 containing the text “Superman”. The use of operators is shown next, which has the same result.



Alternatively, you could create a new string to hold the result:



**Note:** Overloading Operators refers to changing the functionality of an operator depending on the type being used. For example, int += int results in an int, string += string results in a string.

**Example of String Comparison**

There are also a couple of possible approaches to determining if two string types match. The first is the compare() method, the second is an overloaded == operator. Both examples below will give the same result.

Text

Description automatically generated

**Example of Swapping Strings**

**<string>** has a function called **swap()** and this allows you to swap the contents of one string for the contents of another.

Graphical user interface, text

Description automatically generated

**Example of Substrings**

Another useful function is the substr(), which allows you to retrieve a particular portion from a string. The first integer parameter is the index in the string to start at, and the second integer parameter is how many characters from the first index.

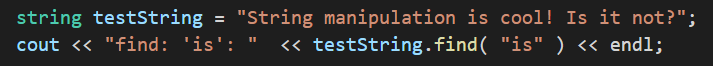
Text

Description automatically generated

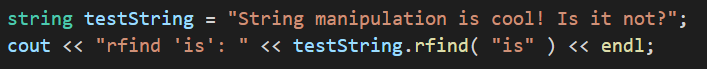
**Example of Substring – Finding**

At times you will need to locate a particular phrase or character. **<string>** provides functionality for this. There are a few variants of the **find()** function and an example of each is shown below.

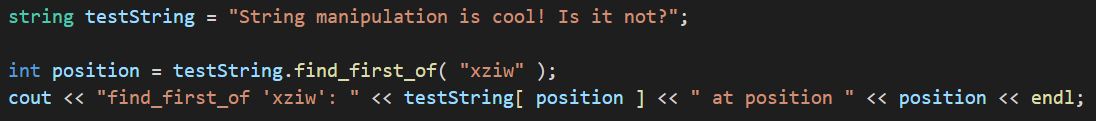
To find the location of the first occurrence of a phrase:



To find the location of the first occurrence of a phrase starting at the end of the string, use the rfind() method.



To locate the first / last occurrence of a character from a string you supply use the following examples.



The above example will find a ‘i’ at position 3.

**Example of Substring – Replacing**

If you want to replace a character in a string with a different character then it’s as simple as using the replace() function. You will no doubt need to use the find() function first to locate the position of the character you want to replace. The first integer parameter is the position to start overwriting, the second parameter is how many characters to replace and the third is the string to replace with.

Text

Description automatically generated

**Example of Substring – Inserting**

Replacing characters in a string is all well and good but what if you need to add some characters? That’s where the insert() function comes in useful.

Text

Description automatically generated

The above example will result in testString containing “123xx456”.

Note: Before we move onto the string programs for you to implement it is important to note that there are many overloaded versions of the functions covered in this chapter, and a lot of others that have not been covered. It is up to you to do your own research to discover what they are and how they work.

**Program 26: Replacing Characters**

Write a program to prompt the user to enter in a sentence, which is then stored in a variable of type string. Output the string they have entered to the console screen, then replace every occurrence of ‘e’ with an ‘X’.

The program should then output the altered sentence.

**Tips**

* You will need to keep an index of position
* You will need to use a loop to access the characters in the string using the find and replace
* Alternatively, do research on the functions available in the <algorithm> file.

**Program 26 Source Code:**

**Program 26 Screenshot:**

**Program 27: String Manipulation**

Write a program to prompt the user to enter their name, which is then stored in a variable of string data type. Insert their name in between the ‘XX’ in the following string:



Output this string to the console. The program should then remove the two X’s and then output the string to the screen again. (You should have two outputs at this point:

“Do you know who loves C++ XnameX does!” and “Do you know who loves C++ name does!”)

The program should then ask the user to input another name (this one **MUST** longer than the first). The program should then locate the first name in the test string and replace it with the new name.

Output the altered string with the second name in place of the first to the console and screenshot the entire process.

**Tips**

* Store the first and second name in a strings of their own.
* Don’t forget to test for a longer name on the second entry

**Program 27 Source Code:**

**Program 27 Screenshot:**

**Program 28: Strings and Vectors**

Below is the output of a program you have to try and replicate.

Text

Description automatically generated

The program must:

* Start with an empty vector of string
* Store the user’s choice in the aforementioned vector
* Decide if you are using cin, getline, or both
* Use a loop to ask for items until inventory is full. Set inventory max to 3.
* Give the user the contents of their inventory and replace their choice with the magic ring
* After being robbed you must delete the contents of the inventory and output the contents of the inventory one last time.
* If you are up to the challenge, proof check every entry to ensure no wrong entries. This is not a necessity.

**Program 28 Source Code:**

**Program 28 Screenshot:**

**Program 29: Rock, Paper, Scissors**

Write a game of Rock, Paper, Scissors; where the user is prompted to enter their choice as a string (e.g. Paper or paper) and the computer's choice is generated randomly. The first to score 3 wins, is the overall winner.

At the conclusion of the game your program should output the result (who won) and the scores of both players.

Those not familiar with the game can learn a little about it here (<https://en.wikipedia.org/wiki/Rock-paper-scissors)>

When you have completed your program, take a screen shot of the last three plays and the concluding output (see below for an example).

Text

Description automatically generated

**Tip**

* Research the use of Rand or Random Device within the <random> file. (Rand will suffice)

**Program 29 Source Code:**

**Program 29 Screenshot:**

**Chapter 10: Debugging**

Debugging is something that as a programmer you will do every time you sit down at a keyboard. It is not a task that is left until a program is written, it is a process that you must go through many hundreds of times before a program can be completed.

So, what does debugging mean? It is the process of searching out bugs in your code and fixing them. There are a variety of tools available within Visual Studio and we will be looking at these in this chapter. Before we get to the tools, we will first look at the different types of bugs you will encounter.

|  |  |
| --- | --- |
| **Bug Type** | **Description** |
| Syntax Error | Code that will not compile. |
| Functional Error | Code that compiles but gives unexpected results. |
| Memory Leak | Program will run for a period of time and then implode. |

Table 10.1: Types of bugs

**Example: Syntax Errors**

Examples of syntax errors are incorrectly spelled keywords or variables and missing semi colons. There are far more reasons for a syntax error, but regardless of the reason Visual Studio will not compile the code and should in most instances give a detailed reason for the issue in the output window. At times, the bug report will not be very helpful and will mean you have got to search for the source of the error yourself. Double clicking on a bug in the output window will take you directly to the area in code where Visual Studio thinks the issue is to be found.

Text

Description automatically generated

**Example: Functional Errors**

Examples of functional errors would be the use of a single = when you meant to use ==. This code would compile but would not give the results you expect. These types of bugs can be somewhat more difficult to find as the error might not become immediately apparent. That is to say, everything may run as expected until much later in the program execution.

A screenshot of a computer

Description automatically generated with medium confidence

**Example: Memory Leaks**

When assigning memory, it is vital that you free the memory once you are finished with it. Imagine a function that reserves memory for the task it is about to complete but does not free the memory at the end of the function. This function is then called several hundred times. Eventually your computer will run out of memory and the program WILL crash. We’ll cover these more when we get to Pointers as it these that are the most common reason for memory leaks.

**A screenshot of a computer

Description automatically generated with medium confidence**

**Breakpoints**

A breakpoint is a tool that allows you stop your code at run time. This allows you to look at the current values stored in variables and even step through the code and see where it goes.

There are two types of breakpoints that can be used in Visual Studio to help you track down functional errors.

The first and most used is the bog-standard breakpoint. To place a breakpoint simply click in the grey vertical bar that runs down the left-hand side of your code. A red dot should appear. With this in place run your program and you will see that execution is paused when the breakpoint is hit. To remove this breakpoint simply click on the red dot and it will be removed. You can also disable a breakpoint from either the breakpoint window or from the debug menu. To disable a breakpoint means to leave it in place, but to turn it off. It will grey out slightly if disabled. A disabled breakpoint will not stop code execution. If you right click on the red dot you will see that options are presented. From this menu you can add conditions to a breakpoint. For example, if you had a breakpoint within a loop, but you don’t want it to stop until it is on its 20th iteration, you can set this up from this menu.

The second type of breakpoint is a data breakpoint. This type of breakpoint is not placed on a line of code, but rather placed on a memory address. Whenever the contents of that memory address are altered the data breakpoint will stop execution of the code at the line of code that changed the memory. This is useful when allocating your own memory using pointers and you have a particularly difficult bug to track down. To assign a data breakpoint select breakpoints>data breakpoint from the debug menu. You must assign the memory address you are interested in monitoring.

When you have the program paused you can step through your code. The most commonly used functions are to step over a line of code, step into a line of code, or step out of a function. These are all available through the debug menu, the shortcut menu or shortcut keys. Familiarise yourself with these shortcut keys as you will be using them.

**Debug Windows**

As mentioned above there is a **Breakpoint Window**. This is where you can manage all breakpoints currently in your codebase. You can even remove them from you code from the breakpoint window.

The **Immediate Window** is where all local variables are shown. The values stored within these variables are shown to the right of the variable name. Any variable that changes during stepping through code will change red to show you it has changed.

There is a **Watch Window** available that allows you to add any variable names that you wish to watch. Once added you will be able to see the values of these variables in the same way as the immediate window detailed above.

**Program 30: Broken Code**

Enter the following program and fix all the syntax errors to get it working. Copy your fixed code in the relevant slot below along with a screenshot of the working program.

Tip: There are 6 errors (the double quotes don’t count if you have copy and pasted) that will stop this program running. Your output may say more, but once these 6 errors are fixed all compounded errors will disappear.

#include <iostream>

int mian()

{

int num1 = 5;

for( int i = 0: i < num1; i++ )

{

cout << “i = “ << i << endl;

}

int num1 = 15;

for( int i = 0; i < num2; i++ ){

cout << “i = “ << i << endl;

/\*Pause the output/

cin.get()

return 0;

}

**Program 30 Source Code:**

**Program 30 Screenshot:**

**Program 31: Unexpected Code**

Enter the following program and fix all the functional errors. This program will run, but it will not give the expected results. Copy your fixed code in the relevant slot below along with a screenshot of the working program.

The expected output is:

1 is an odd number

#include <iostream>

using namespace std;

int main()

{

int count = 3;

for( int i = 1; i > count; i++ )

{

cout << i;

if( count = 2 )

cout << “ is an even number” << endl;

else

cout << “ is an odd number” << endl;

}

// Pause the output

cin.get();

return 0;

}

2 is an even number

3 is an odd number

**Program 31 Source Code:**

**Program 31 Screenshot:**

**Program 32: Broken BMI**

For this program you have been supplied with a cpp file containing code for a quick BMI calculator. The program greets the user, takes their name, and then asks for their weight in stone and pounds, which is then converted into total pounds. Asks for their height in feet and then inches followed by calculating the height in total inches, and finally takes these values to calculate their BMI using the formula of: BMI = .

To get started, create a new project and either replace the default cpp file created with the one attached or copy and paste the code across, the choice is yours.

Below you will find a screenshot of the expected output your program needs to replicate, however, within the code supplied there are several errors and some code missing completely. You should be presented with 5 errors in the error list however there is in fact 9 changes to be made including lines of code that are needed.

Text

Description automatically generated

Expected output: Change name and values as you wish.

**Program 32 Source code:**

**Program 32 Screenshot:**

**Chapter 11: Structs**

At times as a programmer you need to put various data types together into a unified single type. An obvious example would be a 2D position on the screen. This position would be made up of two float data types, one called x, the other called y. How then do we do this then? Well a struct is the answer.

The struct statement will define a new data type that can be used throughout your program. The definition of a struct is as follows:

Text

Description automatically generated

**Example: Defining a Struct**

Using the position example given at the beginning, we would set this up as:

A screenshot of a computer screen

Description automatically generated with low confidence

**Example: Declaring a variable of Struct data type**

Now this creates a new data type, but how do we now use this? We initialise it as we would any other data type. So, to create a variable called characterPosition of data type Position, we could do one of the following:

1. Set the variable name as part of the struct statement.
2. Or alternatively we can define it alone like so:

Text

Description automatically generated

**Example: Accessing struct variables**

Now have our new data type Position, but how do we get access to the float variables hidden within. Well to do this we use the ‘.’ [dot] operator.

A screenshot of a computer

Description automatically generated with low confidence

**Example of structs and Functions**

It is also possible for a struct to contain functions, although this is not so common since the introduction of classes. We will be including functions here for completion sake. The most obvious function to include in any struct is a constructor. A constructor will be called on initialisation and can take parameters to set the internal variables. A constructor must have the same name as the struct name and NOT return any values.

This code snippet shows two constructors for the Position data type. The first takes no parameters and sets the internal variables to zero, the second takes in the starting values with which to set up the internal variables. You can have as many constructors as needed, but they must have different parameters data types for the compiler to know which one to call.

Text

Description automatically generated

**Example: Using constructors with structs**

There is a difference when declaring a variable of a struct data type using a constructor and it uses the following format for the first constructor and second detailed above:

A screenshot of a computer

Description automatically generated with medium confidence

**Program 33: Structs**

Write a program to prompt the user to enter their name [string], age [int] and telephone number [string]. This is to be stored in a struct data type that you have defined.

Output the struct details to the console using a function defined within the struct definition. Something along these lines:

****

Hint: If you get stuck look at the slides for member functions and possible whitespace issues using cin to string.

**Program 33 Source Code:**

**Program 33 Screenshot:**

**Chapter 12: Pointers**

Pointers are often seen as the most confusing aspect of C++. This is probably in partly because so many languages these days have managed memory. That is to say, the compiler takes care of allocating / de-allocating memory. C++ however gives you full control of this. This chapter will step you through the process. Ensure you understand each of the examples before continuing. It is easy to get lost on this topic.

**What is a pointer?**

Well a pointer is a variable like any other variable you have created up to now, but instead of it storing a value it stores a memory address. This memory address points to a location in memory where the actual value is stored.

**Why not just use variables?**

Using pointers allows us the ability to change the value from anywhere in the program that has access to the pointer. So, as we saw in the functions examples when you pass a value through as a parameter, any changes in the function are lost. If we access the memory address and pass by pointer, then any changes to the value through this pointer will persist.

Let us look at some examples...

The first thing we need to know is how to define a variable of pointer type. Well we still need to know what data type we are using, exactly as you would for a normal variable. This is because the compiler needs to reserve a chunk of memory large enough to hold the data.

**Example: Declaring pointers**

The only difference in declaration of a pointer variable when compared to a normal variable is the use of the’\*’ symbol. See below:



**Example: Initialising pointers**

Straight forward enough, but at this point we have only declared a pointer variable, we have not yet reserved the memory required to hold the value. To do this we need to use the new keyword. There are several ways to initialise the pointer. One is to use the new keyword followed by the data type, then a second line of code using the ‘\*’ symbol again (indirection operator) to dereference the pointer. That is to say, access the value stored at the address held in the pointer (sounds more confusing than it really is). Like so:



You may notice that there is a green line under our pointer. This is to tell us that assignment and declaration can be joined:

Graphical user interface, text

Description automatically generated with medium confidence

Alternatively, we can set the value at declaration as well:



This can also be written thusly:



It is also possible to use your pointer to point to a normal variable of the same type. So, this example creates an int called myNum and then creates an int pointer, which points at myNum. Both variables can be used to change the value. Note the use of the ‘&’ symbol. This retrieves the address of a variable.

Text

Description automatically generated

The second line in the above code snippet says, create enough memory to hold the address of an int and put the address of myNum in there.

Note: For every *new* keyword used in your program, you must have a matching *delete* keyword to remove it from memory. This prevents memory leaks etc.

Confused? Let’s look at a full code example using the above code snippets.

**Example: Starting with Pointers**

**Text

Description automatically generated**

**Output:**

**Text

Description automatically generated**

The above example is straightforward with the comments. This is showing you the different options you have to output to console just as seen in the lecture. Unlike normal variables, we have three options rather than two. Outputting the value of the pointer, just as we have any other variable so far (this would output the address of the variable it’s linked with), output the value at the address held using the ‘\*’ symbol, and outputting the personal address of the pointer using the ‘&’.

Note: If you were to run this program yourself, don’t worry if the addresses do not match. These change with each execution of the program.

Next, we will take a look at how we can use pointers and functions. As said above, we can change the value stored at the pointer address from anywhere in our program that has access to the pointer, which includes passing it into a function.

**Example: Passing by Pointer**

This is the same example that was used in Chapter 7: Functions, where a function is created that takes two integers and adds them together. In this version we will see how we can pass through the result pointer and see that it is permanently changed.

First, we create the prototype for the function above the main:



Next, let’s add code to the main function:

A screenshot of a computer

Description automatically generated with medium confidence

Finally, we add the function body below the main() function. Notice this time we store the result of the calculation in the pointer. It is important to remember to dereference the pointer in order to access the value stored at its address.

Text

Description automatically generated

Run the program and examine the output to help you understand what is happening.

**Example: Swapping Values with Pointers**

This code listing will demonstrate how you can swap the values stored in ordinary variable by accessing their memory address. This will be achieved by passing the addresses of the variable through a function that takes pointers as its parameters.

As before, we add a function prototype above the main:



Followed by our main:

Text

Description automatically generated

And then our function body constructed under the main:

Text

Description automatically generated

Run the program and examine the output to help you understand what is happening.

**Program 34: Replacing Commas with spaces**

Create a new program that will ask the user to enter some text (be sure to instruct them when adding text, in place of spaces add a comma). This text should then be passed to a function, which will replace any commas found with blank spaces. The function **must** be in the following format with this prototype:



The input text **must** be passed into the function and be changed within this. The altered text should be output to the console screen via the **main** function., i.e do not output the altered text in the comma2blank() function.

This can be achieved in a number of ways, feel free to include additional libraries.

**Program 34: Source code:**

**Program 34: Output Screenshot**

**Program 35: The Changing Pointer**

This might be a challenge for some. Write a program that has integer variables called num1 and num2.

num1 and num2 should have values assigned by request from the user. This **must** be done in a function called inputDetails().This function should have the following format:



Within the function it should ask the user to input two numbers and then populate the variables appropriately.

Hint: create two int variables as normal and store the answers as you have previously, just inside a function. Once you have these values your n1 and n2 need to equal these values…

Within the main() function create a pointer of int data type called pNum and point it to num1. Next call your inputDetails function passing the correct parameters.

Hint: you need to pass addresses…

Write another function called outputDetails() which takes num1, num2 and pNum as parameters.

This function should output the following details to the console screen:

1. num1 value.

2. num1 address in memory.

3. num2 value.

4. num2 address in memory.

5. pNum value (the address it currently holds)

6. pNum dereferenced value.

7. pNum address in memory.

Ensure the output is referring to num1, num2 and pNum and not local copies. This is where the function prototype is crucial. Your output must be clear. I’d recommend outputting some text to explain each of the above.

Back in the main() function call outputDetails. Next you are to reassign the pointer to point at num2 and output the same as above by calling your outputDetails() function. Might be useful to add a message stating the pointe change before calling outputDetails again.

Remember to set the pointer to point at nothing once you have finished with it.

**Program 35: Source Code**

**Program 35: Output Screenshot**

**Chapter 13: References**

References are another way of referring to a variable, but with a different name. The way this works is similar to pointers, but without all the messy syntax.

**So, what is a reference?**

A reference is an alias to an existing variable. The reference will store a memory address to the variable it is referring to and enable you to change its contents, unlike a pointer, a reference shares its address with the linked variable, whereas a pointer is its own separate entity. This is particularly useful with regards to functions.

**Why not just use pointers?**

Using pointers can get a little messy at times with all the dereferencing that you must do. References as you will see shortly can be used in the exact same way as you would the original variable. No dereferencing required. The benefit of using a pointer though is that you can reassign it to point at another object of the same data type. This is not the case with a reference. A reference can only be assigned the once and must remain referring to that variable.

The advice should be to use references when you can and use pointers when you must.

**Example: Declaring and initialising references.**

The only difference in declaration of a reference variable when compared to a normal variable is the use of the’&’ symbol. It must also be set to refer to another variable immediately. See below:

Text

Description automatically generated

Notice the naming convention; pointers tend to start with a small p, references start with a small r. This isn’t strictly necessary, but it is good practice so when reading code, it is easy to spot which is which. The other obvious change is the use of the & rather than the \* after the data type when declaring one.

**Example: Passing by Reference**

Text

Description automatically generated

**Output:**

****

Notice how the reference is used as though it were a normal variable.

**Example: Swapping by Reference**

This code listing will demonstrate how you can swap the values stored in ordinary variable by accessing their memory address. This will be achieved by passing the variable through a function that takes references as its parameters. You used these incidentally in program 35 for the output.

Text

Description automatically generated

Text

Description automatically generated

Run the program and examine the output to help you understand what is happening.

**Program 36: Using References**

Write a program that creates an int variable called num and has two functions, one for plus25, one for minus25. Next create a reference to num called rNum; All the below tasks **must** be done on the **reference**, with the output of **num** shown.

1. Ask the user to input a number and store it in num.

2. Output the value of num.

3. Add 25 to the current value stored in num by using rNum in the function.

4. Output num.

5. Ask the user to input another number and store this in num.

6. Output num.

7. Minus 25 from the current value of num by using rNum in the function.

8. Output num.

Remember, only cout num to show changes to rNum affect num.

**Program 36 Source Code:**

**Program 36 Output Screenshot:**

**Program 37: Designated Username**

This program does not require a lot of code; however, it does introduce new functions that will be used throughout your coding career. They are, srand() and rand(), and these are used to generate random numbers. Well, I say random, they aren’t truly random and creating a random device would be better (contact Jay if you would like an example of how to create a random device) but for what we need rand and srand are work perfectly well.

Srand is used to create a *seed*. If the *seed* is set to 1, the generator reinitialized to its original value and produces the same values as before any call to rand. We use time in our case via the **<ctime>** to create a new seed every time the program is run based of the time.

Rand produces a number generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses the seed we created with srand to generate the series of numbers.

Now, the program! Create a program that asks the user for a username and then states that their chosen name is taken and provide them with a new one.

Steps:

* Create a function prototype that takes a string reference.
* In main, create a string and ask the user for name which is stored in said string.
* Inform them the name is taken
* Call your function
* And output from main the string username value
  + The function will require the following:
  + Graphical user interface, text

    Description automatically generated
  + Using the reference, you must add the random numbers to the end of the name
    - Hint: ints don’t join strings without some help…

**Program 37 Source Code:**

**Program 37 Output Screenshot:**