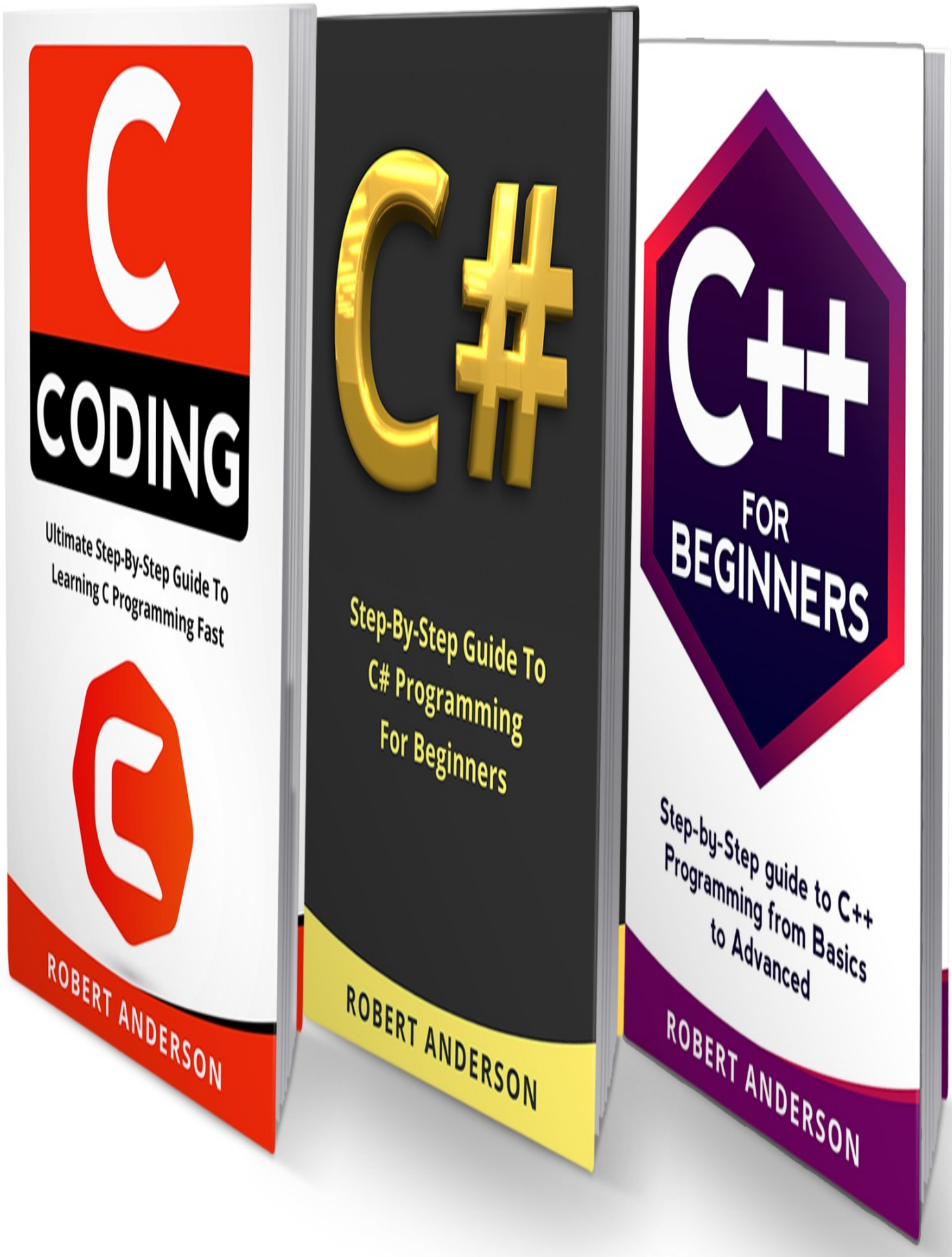

C
CODING
Ultimate Step-By-Step Guide To
Learning C Programming Fast
ROBERT ANDERSON
C#
Step-By-Step Guide To
C# Programming
For Beginners
ROBERT ANDERSON
C++
FOR
BEGINNERS
Step-by-Step guide to C++
Programming from Basics
to Advanced
ROBERT ANDERSON

PROGRAMMING

This book includes:

Robert Anderson

C
Coding

Ultimate Step-By-Step Guide To Learning C programming fast

Robert Anderson

TABLE OF CONTENTS

Chapter 1

Introduction and Installation

History of C

C is a general-purposed computer programming language, that first appeared as a concept in 1972. The lead developer of C was Dennis Ritchie. The origin of C is closely related to the creation of the Unix OS.

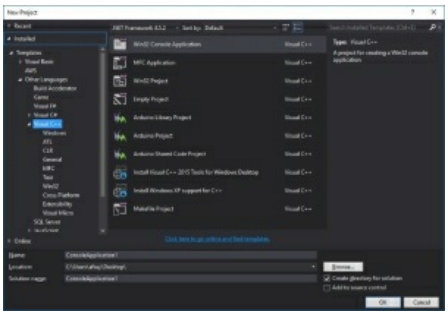
In this tutorial, we will go through the very basics of C right up to the intermediate level sections. The tutorials is designed to appeal to the first-time learners of a programming language.

Running C on Windows

Running C on Windows can be done with Visual Studio 2015. Visual Studio can be downloaded directly from Microsoft, through the installation process C++ needs to be selected as language. The menu should look like this:

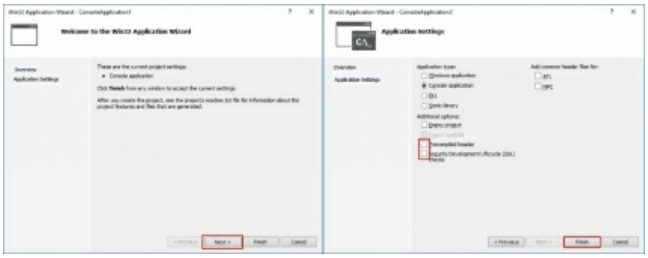
After you select ‘Visual C++’ fully install the program.

You should loads Visual Studio up and go to FILE -> New Project, you’ll be met by a Window like so:



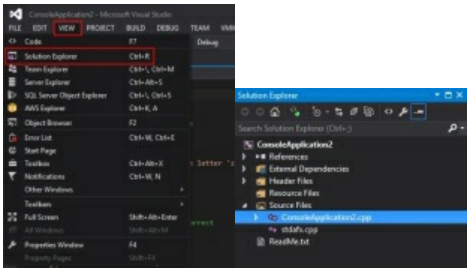
Select ‘Win32 Console Application’ and click “Ok”

A few menus will now appear, follow them through with the highlighted sections. (The highlighted tix-boxes need to be *unchecked)*:

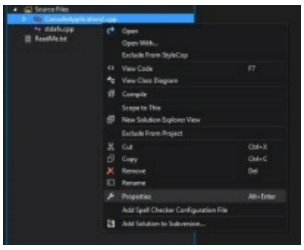


When the program template loads up you should notice you have not been provided with a C environment but a C++ one, you need to convert it to C:

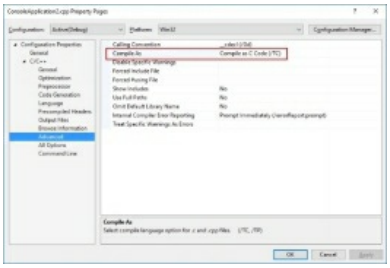
Go to the “Solution explorer” and it will open the “Solution Explorer” Menu



Right click on the **.cpp** file and click “Properties”



A menu will appear, expand the **C/C++** menu, go to “**Advanced**”, loop for the “**Compile As”** section and select from the drop down menu “**Compile as C Code (/TC)”**



The code is now ready to run! When the code you want to compile is reader click the green arrow on the top menu to run your console application.

Running C on Linux

Developing basic code on Linux is relatively easy first you’ll need the ‘*gcc’* compiler, if you don’t have it installed run:

```
apt-get install gcc-5
```

The next stage is to create your program, open up a text pad program (Leafpad in this case) use this basic program for testing purposes:

```
#import <stdio.h>
int main()
{
    printf("Hello, World!\n");

    return 0;
}
```

Save this file with the **.c** extension.

To compile and use the program you'll use *gcc* like so:

```
gcc -o <OutputProgramName> <C_FileName>
```

Where;

OutputProgtamName is the name of the executable you want
C_FileName is the name of your C file

To run the program just type the executables name followed by "./"

```
./<OutputProgramName>
```

Running C on Mac

Running C on a Mac is relatively easy, open up the *terminal* and run

```
clang --version
```

Clang is a compiler built by Apple and with deal with all the aspects of translating code into machine code (1's and 0's)

Start your notepad, write down your C code use this to start:

```
#import <stdio.h>
int main()
{
    printf("Hello, World!\n");
```

```
    return 0;
}
```

Save this code with the **.c** extension.

Open up a terminal in the folder where you saved you .c file, and type

```
make <filename>.c
```

Where filename is the name of the file you just saved. To run the code just run it like so:

```
./<filename>.c
```

Running C online

One of my favourite online IDE is https://www.codechef.com/ide it provides a clear clean place to produce code.

Prerequisites

Throughout this tutorial you will see code snippets dotted around. There's a few points that need explaining so you can get the most out of this tutorial:

Comments

Comments in code are signified by a "//" and the green colour, these are ignored by the computer and commenting is an important part of keeping computer code readable and easy to maintain. Starting coders tend to discard commenting and regret it later, don't be that programmer.

Basic structure

There is also a required basic structure of a C programme, the structure is below:

```
#include <stdio.h>

int main()
{
        //Code goes here
}
```

This is the structure to make your C programs run, don’t worry we’ll learn what it all means.

Chapter 2

Building Blocks

Variables and Types

The basis of a program is data, in the form of numbers and characters. The storing, manipulation and output of this data gives functionality. Variables hold this data, think of it as a box used to store a single item.

C is a heavily typed language where each variable must have a defined type. A type is an identifying keyword that defines what the variable can hold. The first type we will come across is the integer, this can hold real numbers (numbers without a decimal place), an integer is defined below:

```
int integerName = 3;
```

- **int** is the defined type keyword, we will learn about the difference possibilities later.

- "**integerName"** is the ID for the variable, this can be anything you want to call it, this is used to allow a variable to have a meaningful name, the variable could be defined as **"int pineapple = 3;"**, but it's good practise to make them relevant. However, there are a few exception to this as a variable cannot be a single digit i.e. '4' or cannot contain special characters (!"£$%^&*) etc.

- **"= 3;"** Is the assignment section, where the values of 3 is placed in the integer box for use later**. This also ends with a semi colon, this is used to signify the end of a line.**

This variable can now be used in valid areas of the program, like so:

```
int newInteger = integerName
```

The value of **intergerName** defined earlier (3) will now be placed in the **newInteger** variable and they will now both have the value of 3. The value of **integerName** doesn't change as it is just being copied and placed into **newInteger**.

<u>String</u>

The String is another crucial variable, this variable type is used to store a

series of characters, an example could be the word “batman”, the word is composed of characters that are stored in the String variable. It’s an array of characters (More on arrays later), but effectively it’s the single characters of the word stored next to each other in memory. The ‘\n’ is a special character that stands for a newline.

```
char word[] = "dog\n";

//And it can be used and printed it like so:
printf(word);
```

Boolean

Booleans are used as expressions, their values can only be **true (1)** or **false (0)** and are used to signify certain states or flag certain events, they can also hold the result for a conditional expression (More on conditional expressions later). These will make more sense when we go thought conditional statements.

```
_Bool falseIsDetected = 0; //False

_Bool trueIsDetected = 1; //True
```

Float/Double

Floating point variables are decimal values created with float point precision math, the technical elements of how this works are outside of the scope of this tutorial but can easily be explained through resources on the internet, just search “floating point precision”. Floats allows for a higher level of accuracy of a value by providing decimal precision. You can specify a ‘float’ value by putting an ‘f’ and the end of the value.

```
float decimalValue = 3.0f;
```

Void

This data type is special and is used to specify no value is available. This sounds counter intuitive, but we’ll see where this is used later.

Notable keywords/terms

const – This keyword turns the variable read-only, meaning it’s value cannot be changed after it is initialised. The keyword is used like so:

```
const float pi = 3.14f;
```

Global variable – This is a term describing a variable definition outside the main function. For example, the **int** friendCount is a global variable and the **int** currentMonth is not. Note the positions they're defined:

```
//Global
int friendCount = 0;
int main()
{
	//Not Global
	int currentMonth = 5;
}
```

This means the global variable can be used in **any** location in the program and can be dangerous if not used correctly. One correct way it to use it in conjunction with the **const** keyword above, this means functions can only reference the value and not change it.

Recap

- int Holds real numbers
- float / double Hold decimal numbers
- void Specifies there is not type
- boolean Holds either true or false
- string An array of characters making up a word or sentence
- global A variable that can be accessed anywhere
- const Means after a variable has been initialised its value cannot be changed

Arithmetic operations

Symbol	Use
+	int result = 1 + 2;
-	int result = 1 – 2;
/	float result = 1 / 2; (This is stored in a float because of the decimal)

*	Int result = 1 * 2;
%	Modulus operator returns the remainder of a division int remain = 4 % 2;

Conditionals

Comparing values to other values in a meaningful way is fundamental for conditional statements, below is a list of comparisons of values.

Name	**Symbol**	**Detail**
Greater than	>	Returns true if left-hand side is larger than the right
Greater than or equal than	>=	Returns true if the left-hand side is larger or equal to the right
Less than	<	Returns true if the left-hand side is smaller than the right side
Less than or equal than	<=	Returns true if the left-hand side is smaller or equal to the left side
Equal	==	A double equal sign is used to compare if the value of either side is equivalent and returns true if equivalent
Not Equal	!=	Returns true if the two values are not equivalent

The list of conditionals above can be used in certain circumstances to control the flow of the program.

<u>If statement</u>

There are situations where you need something to happen if a certain condition is the case, this is the role of the If statement and where conditional

statements come into the mix.

```
if (Condition)
{
    //Code will run here if Condition is True
}
//If the condition is false, the If statement is ignored and the program jumps here
```

A real-world example:

```
int compare = 10;
if (compare > 5)
{
    //printf() will print the string in the brackets
      printf("Code here will run!");
}
```

Output:
> Code here will run!

These allow a programmer to control the flow of the program and choose situations to happen when a possibility is true, we'll go onto more examples later.

Else Statement

Else's are optional but these can be added to the end of an if-statement and will only run if the if-statement condition is false

```
if (Condition)
{
    //Code will run here if Condition is True
}
else
{
    //Code will run here if Condition is False
}
```

Else statements can also become an else/if statement where a new if statement is attached, this look like this:

```
if (Condition1)
{
    //Code will run here if Condition1 is True
}
else if(Condition2)
{
    //Code will run here if Condition1 is False and Condition2 is True
}
//If neither are true no code will run
```

You can also chain another else statement onto an else statement effectively creating an infinite chain. If any conditions along the chain are true, the ones below are not checked, I'll demonstrate this below:

```
 int main()
{
        _Bool false = 0;
        _Bool true = 1;

        if (false)
        {
        printf("Condition 1");
        }
        else if (true) //This condition is true!
        {
        printf("Condition 2");
        }
        else if (true) //Ignored, due to previous else statement being true
        {
        printf("Condition 3");
        }
        else if (true) //Also ignored due to condition 2 being true
        {
        printf("Condition 4");
        }
```

```
}
```

Output
> Condition 2

After the body of condition 2 is hit and the **printf** statement is executed, the program does not go onto to check the other else statements.

Exercise
Create a program that prints out if a value is bigger or smaller than another, use this skeleton below program to get you started.

So in this case it should print “Value 1 is larger” (Note: Use **printf()** for printing

```
#include <stdio.h>

int main()
{
	//Values you change
	int value1 = 10;
	int value2 = 5;

	//PUT CODE HERE
}
```

Solution
This solution could be something like this:

```
#include <stdio.h>

int main()
{
	//Values you change
	int value1 = 10;
	int value2 = 10;


	if (value1 == value2)
	{
	printf("Values are equal!");
```

```
        }
        else if (value1 > value2)
        {
        printf("Value 1 is bigger!");
        }
        else
        {
        printf("Value 2 is bigger!");
        }
}
```

Just change the values of value1 and value2 before you run the program to test it.

Using multiple conditions

You can use more than one condition in a single statement, there're two ways of doing this **AND** signified by **&&** and **OR** signified by || (Double vertical bar).

AND checks if both conditions are true before triggering the body of the statement

```
if (Condition1 && Condition2)
{
    //Code will run here if Condition1 AND Condition2 are True
}
```

OR checks if one of the conditions are true before triggering the body of the statement

```
if (Condition1 || Condition2)
{
    //Code will run here if Condition1 OR Condition2 are True (Works if both are true)
}
```

Recap

- If statement Deals with conditional statements, code within it's body will run if the condition is true
- else statement Used as an extension to an if statement that is only checked if the if statement is false
- && Used to string two conditions together and will only return true if both are true
- || (Double Bar) Used to string two conditions together and will only return true if one of the conditions are true

Switch-Case

Switch cases are used to test a variable for equality with a constant expression without the need for multiple if-statements. One use for this structure is check users input string. Below is the basic structure of the switch case:

```
//Switch-Case
switch (expression)
{
//Case statement
case constant-expression:
        break; //Break isn't needed

//Any number of case statements
// |
// |
//\ /
// .

default:
        break;
}
```

The switch starts off with an 'expression', this is the variable that is to be compared to the 'constant-expressions', these constants are the literal values of the variable such as '1' or 'Z'. Breaks are optional but without them the code will 'flow-down' into other statements. There is an example below using a switch-case statement, the user inputs a character if the char is N or Y, 'No' and 'Yes' is output respectively but there's a default case that applies

for all situations, this needs to be at the end.

```
char character;

//Reads in user input (Explained in more detail later)
scanf("%s", &character);

        //Switch-Case
switch (character)
{
         case 'N':
        printf("NO\n");
        break;
        case 'Y':
        printf("YES\n");
        break;
        default:
        printf("Do not understand!\n");
        break;
}
```

If there is no **break** statement a ‘flow-down’ will occur, below is an example just like above but without the break statements and we’ll what the output is like:

```
char character;

//Reads in user input (Explained in more detail later)
scanf("%s", &character);

//Switch-Case
switch (character)
{
        case 'N':
        printf("NO\n");
        case 'Y':
        printf("YES\n");
        default:
        printf("Do not understand!\n");
}
```

If ‘N’ is the user input the Output will be:

```
>NO
>YES
>Do not understand!
```

This is because when one case statement is triggered it will continue down until a break statement is found to stop running the case body code.

Iteration

Iteration means looping, and looping quickly gives programs the ability to perform lots of similar operations very quickly, there’re two types of iteration: ‘for loops’ and ‘while loops/do-while loop’

<u>For Loop</u>

The for loop is given an end value and loops up until that value, while keeping track of what loop number it’s currently on, here is an example below:

```
for(int x = 0; x > 10; x++)
{
    printf("Looped!");
}
```

```
for(int x = 0; x > 10; x++)
```

Each part of the for-loop has a role

Red

This is the *Declaration* section to define the loop counter variable, this defines the start point of the counter

Green

This section is called the *Conditional* and it contains a conditional statement that is checked at the end of the loop to see if the if-statement should continue looping, so in this case the loop should continue looping if **x > 10,** if this condition becomes false the loop will not continue.

Blue

The blue statement is the *Increment* section where the loop counter is incremented (increased in value), the **x++** is shorthand for **x = x + 1** . This can also be **x--**, if there was a case requiring the counter to decrease.

- Declaration section defines the loop counter
- Conditional section continues the loop if true
- Increment section is where the loop counter is incremented

Conditional Loops

Conditional loops work like the for loop but don't have a loop counter and will only loop while a condition is True. This means you can create an infinite loop, like so:

```
while (TrueCondition)
{
     printf("This will not stop looping");
}
```

Note:
An infinite loop is normally constructed using a naked for-loop:

```
for(;;)
{
    printf("This will also never stop looping");
}
```

This loop will never end and your program will get stuck within the loop.

The example below shows if the condition is false the program will never reach the code within the brackets

```
while (FalseCondition)
{
    printf("This code will never run");
}
```

To use this loop effectively you can use it with a conditional statement (like the if statement) or you can use it with a bool variable, examples are below

```
int count = 0;
while (count > 10)
{
    printf("loop");

   //Remember this, it's the same as "count = count + 1;"
    count++;
}
```

As it says above you can also use a while loop directly with a Boolean variable:

```
int count = 0;
_Bool keepLooping = 1; //Bool is true (1) is true
while (keepLooping)
{
    printf("loop\n");
```

```
    //Remember this, it's the same as "count = count + 1;"
     count++;

     if(count == 3)
     {
       keepLooping = 0; //keepLooping is now false, and the loop will stop
     }
}
```

This is very similar to how a for loop operates, but it is important to understand the different uses for a while loop.

Note:
The section in the brackets of the while loop is checking if that condition is true, you can also write it like so:

```
while(!stopLooping) {}
```

This is effectively saying, keep looping while stopLooping is "not true", the "!" is symbol means "not".

Do-While

A Do-While loop is almost exactly the same as While loop but with one small difference, it checks if the condition is true after executing the code in the body of the loop, a while loop checks the if the condition is true before executing the body. The code snippet below shows this difference:

```
_Bool falseCondition = 0;

while(falseCondition)
{
    printf("While Loop\n");
}

do
{
    printf("Do Loop\n");
} while(falseCondition)
```

Output:
> Do Loop

Because even though the Boolean is false, the Do loop executed a single time because the check was at the end of the body.

Using a Do-Loop

A real-world example of a do-loop cool be checking for user input, it prints out and asks for input if all is okay there is no need for looping if not it will loop. An example looking for the user to enter a ‘z’ is below

```
#include <stdio.h>
int main()
{
	//Will loop if this is false
	_Bool correct = 1;
	do
	{
	printf("Please enter the letter 'z': ");

	//Takes in user input
	char z;
	scanf("%s", &z);

	//Checks if answer is correct
	if (z != 'z')
	{
	//Incorrect
	correct = 0;
	printf("Incorrect!\n");
	}
	else
	{
	//Correct
	correct = 1;
	}
	} while (!correct);

	//If the user has completed the task
	printf("Correct!");
}
```

Loop control keywords

Sometimes there are situations where you want to prematurely stop the entire

loop or a single iteration (loop), this is where loop control keywords come into use.

You have either a **break** or **continue** keyword:

break – Will stop the entire loop, this can be useful if an answer has been found and the rest of the planned iterations would be pointless.

```
for (int x = 0; x < 5; x++)
{
        if (x == 3)
        {
        break;
        }

        //The technicalities of this statement will be explained later
        printf("Loop value: %d", x);
}
```

Output:
> Loop value: 0
> Loop value: 1
> Loop value: 2

Now if the code is changed to not include the break:

```
for (int x = 0; x < 5; x++)
{
        //The technicals of this statement will be explained later
        printf("Loop value: %d", x);

}
```

Output:
> Loop value: 0
> Loop value: 1
> Loop value: 2
> Loop value: 3
> Loop value: 4

Continue – If we use the code from above but replace it for a **continue** the code will look like so:

```
for (int x = 0; x < 5; x++)
{
        if (x == 3)
        {
        continue;
        }

        //The technicals of this statement will be explained later
        printf("Loop value: %d", x);
}
```

The output it like so:

Output:
> Loop value: 0
> Loop value: 1
> Loop value: 2
> Loop value: 4

This shows that when ***x = 3*** the **continue** is executed and the loop is skipped and so is the **printf** statement is also skipped so there is no “Loop value: 3”

<u>Nested loops</u>
You can also place loops within loops to perform specific roles, in the example we are using for-loops but this can also be done with the while/do loop.
The example is printing out a 2D grid, the nested for-loop gives another dimension:

```
//Prints a 5x5 grid
for (int y = 0; y < 5; y++)
{
        for (int x = 0; x < 5; x++)
        {
        //Prints an element of the row
        printf("X ");
```

```
        }

        //Moves down a row
        printf("\n");
}
```

Output

```
> X X X X X
> X X X X X
> X X X X X
> X X X X X
> X X X X X
```

Functions

Functions are the building blocks of a program, they allow the reuse of code, the ability to keep it readable and stops the programmer repeating code. Repeating code is heavily advised against because bugs will be repeated multiple times and changes to code also need repeating. Functions give a centralised controlled area that deals with the distinct roles of the program.

A method has two elements, **parameters**:(the items passed into the function) and the **return type** (the variable being returned) these are both optional and you can have a function with neither.

A function must be defined above its call, like so;

```
//Function definition
void Print_Smile()
{
        printf(":)\n");
}

int main()
{
        //Method call
        Print_Smile();

        return 0;
```

```
}
```

And this way round would be incorrect:

```
int main()
{
	//Method call [ERROR HERE]
	Print_Smile();

	return 0;
}

//Function definition
void Print_Smile()
{
	printf(":)\n");
}
```

Function Parameters

Sometimes it might be useful to pass data into a program, there are two types of parameter passing, by **reference** and by **value.** Passing by reference is what it sounds like, it passes a direct reference to a variable not a copy, so any changes to that passed variable effect it back in the calling function. Passing by value is the passing of a copy of that variable, so any changes to that variable do not effect that passed variable.

Passing by reference is not directly supported by C, but the effect is possible when dealing with pointers (We will talk about this in the advanced section)

- Passing by reference means changes to the parameter effects the passed variable
- Passing by value means changes to parameter does not effect the passed variable

For the time being passing by value is done below;

```
void Add(int num1, int num2)
{
	//Adds values together
	int newValue = num1 + num2;

	//Prints result
```

```
	printf("The result is: %d", newValue);
}

int main()
{
	//Method call
	Add(10, 4);

	return 0;
}
```

Output:
> The result is: 14

In this case the function has two integer parameters that it takes both in and prints the result. Don't look to deep into the **printf** statement, we'll go into why "%d" is used and how to use **printf** later.

Below is another example but this time a string is used a parameter:

```
void PrintStr(char printData[])
{
	printf(printData);
	printf("\n");
}
```

"char printData[]" is the parameter variable and allows data to be used in that function, in this case it's printing out the char string. The method call looks like this:

```
PrintStr("Flying Squirrel");
```

This is an incredibly useful feature that allows us to make general purpose code and change its function output by what is put in as a parameter.

Returning values

Returning allows us to return data from a method, this lets us do computation within a function and get the function to automatically return the result. Let's take the one of the previous examples and adapt it so it returns the result instead of printing it:

```
int Add(int num1, int num2)
{
	//Adds values together
	int newValue = num1 + num2;

	//Returned keyword
	return newValue;
}

int main()
{
	//Method call
	int storeResult = Add(10, 4);
	return 0;
}
```

The areas that changed have been highlighted. When a value is to be returned, the "return" keyword is used, after this line has run it returns to the **line where the method was called** so any code under the return **will not run.** The returned result is then stored in 'storeResult' to be used later. Returning can happen with any variable type. I'll show you an example below that checks if the number is an even value (Using the modulus operator talked about above that finds the remainder of a division):

```
 _Bool EvenNumber(int value)
{
	if (value % 2 == 0)
	{
	//Return true
	return 1;
	}
	return 0;
```

```
}

int main()
{
	if (EvenNumber(2))
	{
	printf("Even number!");
	}

	if (EvenNumber(5))
	{
	printf("Even number!");
	}

	return 0;
}
```

Output:
>Even number!

This program uses the return variable form the method EvenNumber() as a conditional for the if-statement and if it's true it will print "Even number!". As you can see from the output the first one prints but the second does not.

Recursion

Recursion is a difficult concept and will only be lightly touched on here and its real-world uses and functionality explained in the advanced section.

Recursion is a definition of a functions commands involving a reference to itself, yes very confusing I know, but I use some examples to explain.

```
const int maxLoops = 5;
void Sequence(int previous, int now, size_t loopCount)
{
	//Works out next value
	int next = previous + now;
	//Prints new value
	printf("New value: %d\n", next);
```

```
        //Increments counts
        loopCount++;

        //Stopping condition to make sure infinite looping doesn't occur
        if (loopCount < maxLoops)
        {
        //Recursive call
        Sequence(now, next, loopCount);
        }
}

int main()
{
        Sequence(1, 1, 0);
}
```

Output:

```
>New value: 2
>New value: 3
>New value: 5
>New value: 8
>New value: 13
```

There're a few things to note, the lack of iteration loops, recursion in its essence causes looping. The second thing to not if the if statement labelled 'stopping condition', if recursive set-ups don't have conditions that stop them looping they will loop forever, so this is a crucial element for using recursion effectively. If you don't fully understand yet, don't worry we will go into more detail later in the advanced section of the tutorial.

Arrays

Arrays have been mentioned previously, it is a data structure that holds a set number of variables next to each other in memory. The array will be given a *type*, for example 'int'. Arrays are used to quickly define lots of variables and keep relevant variables together. An array is defined below:

A static size, with the size in the square brackets:

```
int lotsOfNumber[20];
```

Or you can define values at the definition, **Note**: a size does not need to be defined because it's automatically determined by the number of values you specify:

```
int lotsOfNumbers[] = {1,3,4};
```

- **Arrays start at 0**, so the first index has an identifying value of 0, the second is 1 and so on. This means when accessing values, you need to remember it is always one less than the number of values it contains

You can access an index like so:

```
int var = lotsOfNumbers[0];
```

This will grab the first index of the array and place it in 'var'.

Arrays are very useful to access the tightly related data very quickly, you can use a for-loop to loop through the indexes and use them according. An example is below:

```
int lotsOfNumbers[] = { 1, 3, 4, 10};
for (int x = 0; x < 4; x++)
{
        printf("%d\n", lotsOfNumbers[x]);
}
```

Output:

```
> 1
> 3
> 4
> 10
```

In C you cannot get the length directly and need to work it out, this can be done simply using the **sizeof()** keyword, this returns the size of the elements in the brackets, so for example on a 64bit machine a **int** should be represented using 4bytes so **sizeof** will return 4. The length of an array can be worked out as so:

```
int arrayLength = sizeof(lotsOfNumbers) / sizeof(lotsOfNumbers[0]);
```

This takes the entire size of the array, and divides it by the first element and the division gives how many indices the array holds.

Multi-dimensional arrays

You can also define a second dimension in the array or even a third, this gives more flexibility when working with arrays. For example, a 2D array could be used to store coordinates or positions of a grid. A 2D array is defined as so:

```
int arrayOfNumbers[][2] = { {1,1}, {1,1}};
```

You access an element by putting the values in the square brackets for the x and y coordinate

```
int element = arrayOfNumbers[X][Y]
```

The obvious difference is that the second dimension **needs** a value, this cannot be automatically resolved when creating an array and that the initialization requires nested curly brackets. Below is the 3D example:

```
int arrayOfNumbers[][][2] = {{{1,1},{1,1}},{{1,1},{1,1}}};
```

But at this stage it is starting to lose readability and it's much better practise to lay it out like so:

```
int arrayOfNumbers[][][2] =
{
        {{1,1},{1,1}},
        {{1,1},{1,1}}
};
```

Passing Arrays in functions

As we saw when we learned about functions above, passing variables in as parameters can be very useful and so can passing in lots of variables stored as an array. However, we learnt above how to determine the size of an array above but this **does not work** with an array passed as a parameter, so we must pass in a variable that represents the number of elements that array is holding. There is a special variable type used to hold count variables, it's called **size_t** and it is an unsigned integer value (Meaning it cannot become negative) used to hold values for a count.

An example of an array being passed as a parameter is below:

```
void Print_SingleDimenArray(size_t length, int ageArray[])
{
	//Length is used to dynamically determine the for loop length
	for (int i = 0; i < length; i++)
	{
	printf("%d\n",ageArray[i]);
	}
}

int main()
{
	//An array of ages
	int ages[] = { 32, 11, 12, 1, 8, 5, 10 };

	//Size is determined as shown previously
	size_t ageLen = sizeof(ages) / sizeof(ages[0]);

	//Method call
	Print_SingleDimenArray(ageLen, ages);
}
```

Output

```
>32
>11
>12
>1
```

```
>8
>5
>10
```

The array has been passed and used to print values. The length of the array is crucial to the program as it allows the for loop to work for arrays of varying length.

- Any changes to an array in any method will change the array variable in the calling method. This is what was mentioned earlier as **passing by reference,** the whole array isn't copied and passed by reference with a little trickery occurring.

Extra – Advanced

This isn't crucial to understanding but I'm going to explain why determining the size of an array after it's passed as a parameter will not give the correct result.

When an array is passed as a parameter the first element's memory position address is passed (This is a pointer, promise we will touch on these later) and when you use **sizeof** to grab the size of the 'array' you are just getting the size of the first element returned.

User input

Sometimes it is necessary to get an input from the user, in the form of a choice or name etc. There are a few ways of doing this but the most straight forward method is the **scanf()** and **printf()** functions.

Before we go into how they work we first need to learn the basic type identifiers for **scanf()** and **printf(): (This is not a comprehensive list)**

Formatting ID	**Valid Input**	**Type needed for argument**
%c	Single characters, reads the characters and so on so on	char
%f	Float values,	float
%d	**Decimal integer**	**int**

%o	Octal	int
%u	Unsigned decimal integer (Does not have a sign and is just a positive int)	int
%x	Hexadecimal	int
%s	**String of characters and will continue reading until a whitespace is found**	**char[]**

scanf() works by grabbing user input and placing in a variable:

```
char name[20];
scanf("%s", name);
```

And an example of grabbing an integer value:

```
int userInputInt;
scanf("%d", &userInputInt);
```

Note: The use of the “&” for userInputInt. This is saying that it’s an exact reference to the memory location of userInputInt and the value read in from the user is placed into that integer variable by adding it to that memory location.

printf is used to display information to the console for debugging and user interfaces. **printf** also uses the same formatting ID’s as **scanf** and these formatting ID’s are used as placeholders, the example below shows printing a string and integer respectively:

Integer

```
int number = 10;
printf("%d", number);
```

String

```
char word[] = "Lemon";
printf("%s", word);
```

These formatting ID work as place holders and **printf** can take multiple parameters depending on the number of formatting ID present in the print string. For example:

```
int number = 10;
char word[] = "Lemon";
printf("A fruit is: %s and here is a number: %d", word, number);
```

Each formatting ID is replaced in order by the parameters, in this case “%s” is replaced by *word* and “%d” is replaced by *number*. This can work with as many number of formatting ID’s as is needed.

fputs()

fputs() is the much simpler alternative when you just want to **print a string of characters** to the console. **fputs()** does not use the formatting IDs that **printf()** uses so it doesn’t need to check for formatting and is slightly quicker. It also automatically puts a ‘**\n**’ onto the end of the printed line.

Quiz chapter 2

This section is designed to keep you on your toes about the previous content, there will be 10 questions that you can answer to test your knowledge, below in the neighbouring section will be the answers.

1. Name one data type that is used to hold decimal numbers?
2. What two values can a boolean hold?
3. What is a condition statement?
4. Name the three sections that make up the for loop
5. What does the **continue** keyword achieve?
6. Can a valid function not return a value, and if so what datatype is used?
7. How many parameters can a function have?
8. What is the difference between **printf** and **puts**?
9. Which type is a valid input for this formatting id “%d”?
10. What code would you use to find how many values an array can hold?

Chapter 3

Advanced Basics

Pointers

Pointers have been mentioned previously and they are the memory address of a variable, this is much like a house address for a person. These can be passed around a parameter and allow the effects of **passing by reference** mentioned previously.

The program below will show the address of a variable:

```
#include <stdio.h>

int main()
{
	int var;

	printf("The address is: %x\n", &var);

	return 0;
}
```

Output

> The address is: 10ffa2c

Note: This will be different almost every time you run

Note the highlighted statement, the **'&'** (Reference Operator) is used to return the memory location of the variable and allows certain statements to access the data in that location. The '%x' is used because a memory location is in hexadecimal.

To create a pointer, we use the dereferencing operator (*), this will create a pointer variable that is designed to hold a memory locations address. Below is a program that creates a pointer and uses reference operator (&) to store another regular variable's address in the newly created pointer. The dereferencing operator (*) is also used to access or change the actual data in that memory location.

- Dereferencing operator (*) is used to create a pointer, it is also used when changing the actual value
- Referencing operator (&) is used when obtaining the memory location of a pointer

```
//Regular variable
int var = 10;

//Pointer
int* pointer;

//Storing of var memory location
pointer = &var;

//Pointer is now effecivly 'var' so
//things like this can happen
*pointer = 20;

printf("Var's value is now: %d", var);
```

Output
>Var's value is now: 20

NULL pointers

When you create a pointer is it initially not given anything to point at, this is dangerous because the pointer when created it references random memory and changing this data in the memory location can crash the program. To prevent this, when we create a pointer we assign it to NULL like so:

```
int* pointer = NULL;
```

This mean the pointer has an address of ‘0’, this is a reserved memory location to identify a null pointer. A null pointer can be checked by an if statement:

```
if (pointer){}
```

This will succeed if the pointer **isn’t** null.

Using pointers

Now some real-world uses of pointers are passing them as parameters and effectively passing them **by reference**. The example below will show the effect:

```
void Change_Value(int* reference)
{
	//Changes the value in the memory location
	*reference = 20;
}

int main()
{
	//Creates pointer to variable
	int var = 10;
	int* pointer = &var;

	printf("The value before call: %d\n", var);

	//Method call
	Change_Value(pointer);
	//Prints new value
	printf("The value after call: %d\n", var);

	return 0;
}
```

Output

```
>The value before call: 10
>The value after call: 20
```

This passes the memory location not the value of the variable meaning you have the location where you can make changes.

Note the parameter is **int*** this is the pointer type, so for an example of a pointer to a char would be **char*.**

Pointer Arithmetic

There is times when moving a pointer along to a another memory location might be useful, this is where pointer arithmetic comes into use. If we were to

execute say **ptr++** and the **ptr** was an integer pointer it would now move 4bytes (Size of an int) along, and we were to run it again, another 4bytes etc. This can mean pointer (if pointing to valid array structures) can act much like an array can. An example is below:

```
 int arrayInt[] = { 10, 20, 30 };
size_t arrayInt_Size = 3;

//Will point to the first array index
int* ptr = &arrayInt;

for (int i = 0; i < arrayInt_Size; i++)
{
        //Remember, *ptr gets the value in the memory location
        printf("Value of arrayInt[%d] = %d\n", i, *ptr);
        ptr++;
}
```

Output

```
>Value of arrayInt[0] = 10
>Value of arrayInt[1] = 20
>Value of arrayInt[2] = 30
```

This shows that a pointer to the first address of the array can be incremented along the addresses of the array (Remember each value of an array is stored in neighbouring memory locations)

You can do the opposite and decrement a pointer i.e. make the pointers value decrease.

There is also way to compare pointers using relational operators such as ==, < and >. The most common use for this is checking if two pointer point to the same location:

```
int value;

//Assigns ptr1 and prt2 the same value
int* ptr1 = &value;
int* ptr2 = &value;

//ptr3 is assigned another value
```

```
int* ptr3 = NULL;

if (ptr1 == ptr2)
{
        printf("ptr1 and ptr2 are equal!\n");
}

if (!ptr1 == ptr3)
{
        printf("ptr1 is not equal to ptr3\n");
}
```

This checks if the various pointers are equal. The same can be done with > and <.

Function pointers

Much like you can do with variables you can also do the same with functions, below is a snippet of code that shows a function pointer being defined. Key sections will be highlighted:

```
void printAddition(int value1, int value2)
{
	int result = value1 + value2;

	printf("The result is: %d", result);
}

int main()
{
	//Function pointer definition
	//<retrunType>(*<Name>)(<Parameters>)

	void(*functionPtr)(int, int);
	functionPtr = &printAddition;

	//Invoking call to pointer function
	(*functionPtr)(100, 200);

	return 0;
}
```

The basic structure for defining a function pointer is like so

```
<Return_Type> (*<Name>) (<Parameters>)
```

Where in this case:
<Return_Type> = void
<Name> = functionPtr
<Parameters> = int, int

This function pointer can now be passed as a parameter and used in situations where you would want to change the behaviour of code but with almost the same code.

An example could be dynamically choosing what operation a calculator should perform (Note: this is complex code and should be used a rough

example, so don’t worry if you don’t fully understand)

```
void calculator(int value1, int value2, int(*opp)(int,int))
{
	int result = (*opp)(value1, value2);
	printf("The result from the operation: %d\n", result);
}

//Adds two values
int add(int num1, int num2)
{
	return num1 + num2;
}

//Subtracts two values
int sub(int num1, int num2)
{
	return num1 - num2;
}

int main()
{
	calculator(10, 20, &add);
	calculator(10, 20, &sub);

	return 0;
}
```

Here what is happening we are passing the function ‘add’ and ‘sub’ as parameters for the function calculator, as you see from the highlight the function parameter is defined like it is above with the return type, name and parameters being defined, all that is passed into calculator is &add and &sub for the function pointers. The calculator function then goes on to invoke the pointer and passes in the values and returns the result.

Storage Classifications

In C each variable can be given a storage class that can define certain characteristics.

The classes are

- **Automatic variables**
- **Static variables**
- **Register variables**
- **External variables**

Automatic variables

Every variable we have defined so far has been an automatic variable, they are created when a function is called and automatically destroyed when a function exits. These variables are also known as *local variables*.

```
auto int value;
```

Is the same as

```
int value;
```

Static variables

Static is used when you want to keep the variable from being destroyed when it goes out of scope, this variable will persist until the program is complete. The static variable is created **only once** throughout the lifetime of the program. Below is an example of a static variable in use:

```
void tick()
{
	//This will run once
	static int count = 0;

	count++;
	printf("The count is now: %d\n", count);
}

int main()
{
```

```
        tick();
        tick();
        tick();
}
```

Output
>The count is now: 1
>The count is now: 2
>The count is now: 3

The area highlighted section is the static definition and will only run once.

Register variables

Register is used to define a variable that is to be store in register memory opposed to regular memory, the benefit register memory has is it is much much quicker to access however there is only space for a few variables.

Defining a register variable is done like so:

```
register int value;
```

External variable

We touched on this before but a global variable is a variable not defined in a scope and therefore can be used anywhere. An external variable Is a variable defined in a separate location like another file and the **extern** keyword is used to signify that the variable is in another file. You would include another file as reference by placing this at the top of your file:

```
#include "FileName.c"
```

This is to tell the program to reference this file as well. Note the files need to be in the same location.

Program 1 [File_2.c]

```
#include <stdio.h>
#include "C_TUT.c"

int main()
{
```

```
        extern int globalValue;
        printf("The global variable is: %d", globalValue);
}
```

Program 2 (It is small) [C_TUT.c]

```
#include <stdio.h>

int globalValue = 1032;
```

```
The Output of running File_2.c:
>The global variable is: 1032
```

This shows that the **globalValue** is referenced from C_TUT.c and used in another file by using the extern keyword.

File I/O

There will be cases where you'll need to retain data past the duration of the programs life time, this is where saving a loading to file comes into use.
There are two types of main files used
- Text Files
- Binary files

To start you will need to create a pointer of type 'FILE' tile will allow communication between file and program. This is done like so:

```
 FILE* ptr;
```

Opening a file

If this file already exists you can open it and read it's contents, this would be done by using:

```
ptr = fopen(char* filename, char* mode)
```

Where:

filename = to the filename of the file to open

mode = this is the mode to open the file in, the comprehensive list is below

Mode	Description

r	Opens the file for reading
w	Opens the file for writing, if the file does not exist a new file is created. The program will begin writing content from the start of the file.
a	Appending mode, if the file does not exist it is created. Any changes to an existing file will be added onto the end.
r+	Opens the file for reading and writing
w+	Opens the text file for both reading and writing. It first truncates (chops) the file length to zero if it exists, if it doesn't create a new file
a+	Once again opens the file for reading a writing. If the file does not exist a new one is created. Reading starts at the very start but writing is only appended onto the end of the file.

Closing a file

This one is nice and simple, just need to run:

```
fclose(ptr);
```

Writing to a file

There are two different ways of writing to a file **fprintf()** and **fputs()** (Like the console printing commands **printf()** and **puts()**. The only difference really is that **fprintf()** allows you to use the formatting ID like '%s' and **fputs()** does not. This mean **fputs()** does not need to check for formatting and just prints out the exact string making it faster. **fputs()** also automatically adds a newline character (\n) to the character string it is given, much like **puts()**. The example below shows to open, write-to and close a file:

```
#include <stdio.h>

int main()
{
```

```
        //Creates the file pointer
        FILE* ptr;
        //Opens the file
        ptr = fopen("C:/C_IO/example.txt", "w+");

        //Writes to the file
        fprintf(ptr, "fprint()\n");
        fputs("fputs()\n", ptr);

        //Closes file
        fclose(ptr);
}
```

It creates a file in the 'C_IO' directory directly on the C: drive (make this before running the program). It uses the 'w+' mode so every time the program is run the data in file is overridden.

You can also save values only using **fprintf** like so:

```
fprintf(ptr, "%d\n", value);
```

- Opening a file fopen()
- Writing to a file
    - fprintf()
    - fputs()
- Closing a file fclose()

Exercise

- Write a program to save the output of an add function save it to the C_IO on the C (Or your main drive) you made earlier and call it "addSave.txt", use 10 and 25 and your values.

Solution

Something like this, doesn't have to be exact there are always different ways of doing things.

```
#include <stdio.h>

int add(int num1, int num2)
{
```

```
        return num1 + num2;
}

int main()
{
        //Grabs the value to save
        int saveValue = add(10, 25);

        //Creates the file pointer
        FILE* ptr;
        //Opens the file
        ptr = fopen("C:/C_IO/addSave.txt", "w+");

        //Writes to the file – This is how printf works but without the ptrn
        fprintf(ptr, "%d\n", saveValue);

        //Closes file
        fclose(ptr);
}
```

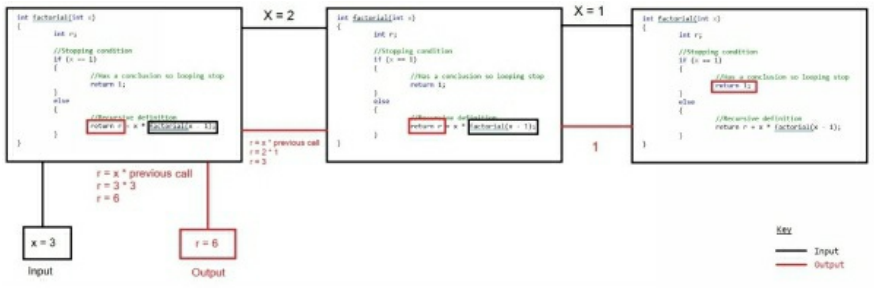
Recursion Continued

We touched on recursion in the basic section of the tutorial, and again it's the definition of a functions tasks with definition to itself. As promised there are a few more examples of recursion explained below:

```
int factorial(int x)
{
        int r;

        //Stopping condition
        if (x == 1)
        {
        //Has a conclusion so looping stop
        return 1;
        }
        else
        {
        //Recursive definition
        return r = x * factorial(x - 1);
```

```
        }
}

int main()
{
        puts("Please enter a number: ");

        //Reads in user input
        int a, b;
        scanf("%d", &a);

        //Starts the execution
        b = factorial(a);
        printf("The factorial is: %d", b);
}
```

This is the world famous example of recursion that is used to find a factorial of a number (3 factorial is 3x2x1). It works by the recursive return statement above, it stops by having a return statement without a recursive definition, i.e. when x = 1 the function just returns 1, this means the stack can unwind and find an answer. There is a flow diagram below:



Exercises chapter 3

1. What does a pointer variable hold?
2. What operator is used to signify a pointer?
3. What will "++" do to a pointer of type int where an integer is 4bytes?
4. How many times is a static variable initialised throughout the life of a program?
5. What happens when you define a variable as 'register'?
6. What are the two main functions used to write to a file?
7. What does the fopen() mode **w+** do?
8. In what case will the code below succeed. Pointer is an integer pointer

```
if(pointer)
{

}
```

9. What is '&' called and what does it do?
10. When you pass an array as a parameter what do you also need to path with it?

Chapter 4

Custom Structures

The customizable aspect of programming languages allows them to perform any role under the sun and the programmer can manipulate and craft structures designed to store and process data.

Structures

The first user-defined object we'll come across is the 'structure', this allows custom storing of data and are defined like so:

```
struct Structure-Name
{
	//Statements
};
```

Where "statements" is the variables the structure is to hold. The structure allows custom storing of data in a meaningful way, it much like the array, the only difference being that an array stores lots of variables of the **same type** the structure allows storing of several types, even other structures.

A real example is below:

```
struct Person
{
	int age;
	char firstName[15];
	char lastName[15];
	char favoriteColour [10];
};
```

This structure is designed to hold data about a person, a new person can be created like so:

```
struct Person p1;
```

It is much like defining a new variable, the example below shows that two

newly created people do contain separate and independent values from each other:

```
#include <stdio.h>

struct Person
{
	int age;
	char firstName[15];
	char lastName[15];
	char favoriteColour[10];
};

void PrintPerson(struct Person p)
{
	printf("First Name:	%s\n", p.firstName);
	printf("Last Name:	%s\n", p.lastName);
	printf("Favourite colour: %s\n", p.favoriteColour);
	printf("Age:		%d\n", p.age);
	puts("");
};

int main()
{
	//Person 1
	struct Person p1;

	p1.age = 10;
	strcpy(p1.firstName,"John");
	strcpy(p1.lastName, "Doe");
	strcpy(p1.favoriteColour, "Red");

	PrintPerson(p1);

	//Person 2
	struct Person p2;

	p2.age = 25;
	strcpy(p2.firstName, "Lucy");
	strcpy(p2.lastName, "Brown");
```

```
        strcpy(p2.favoriteColour, "Yellow");

        PrintPerson(p2);
}
```

Output:

```
>First Name:        John
>Last Name:         Doe
>Favourite colour:  Red
>Age:               10

>First Name:        Lucy
>Last Name:         Brown
>Favourite colour:  Yellow
>Age:               25
```

To copy a string value into a structure variable you need to use **strcpy()**. This code shows that each time a new Person is created so does a whole new set of variables that go along with that Person. This give a very easy 'cookie-cutter' way of creating lots of meaningfully variables very quickly. Also note how each variable is accessed, it's by using a full stop (Access operator) :

```
personName.Variable
```

```
p1.age = 10;
```

Nested structures

Structures can also hold other structures, they can either be: internally defined or externally defined.

Internal definition

This is where you define a structures definition inside another's structures variable definition. If we use the person example and turn the first and last name into a structure.

```
struct Person
{
        int age;
```

```
        char favoriteColour[10];
        struct Name
        {
        char firstName[15];
        char lastName[15];
        } name;
};
```

Where the first name is access like so:

```
p1.name.firstName;
```

External definition
This is like the same but the definition is not within another a structure:

```
struct Name
{
        char firstName[15];
        char lastName[15];
};

struct Person
{
        int age;
        char favoriteColour[10];
        struct Name name;
};
```

Where first and last name are accessed exactly the same as the internal definition.

- Internal definition means you cannot recreate the structure in other locations but can use it internally
- External definition means you can use it internally and in other locations

This gives a much more modular and readable code.

TypeDef

Typedef is a keyword provided so you can customize the name of build in and user defined variables and structures. I show an example below:

```
typedef int INTEGER;
```

This is taking the data type **int** and giving it another persona as INTEGER, now an INTEGER is defined like normal:

```
INTEGER value = 10;
```

Real-World Example

A real world example of where this could be useful is backwards compatibility in situations where integer values are different. So you would define the int like so:

```
typedef int int32;
```

Use int32 like int normally and on the other machine where you would need a larger int value you would change the definition to be:

```
typedef long int32;
```

And the code would still work with a very small amount of maintaining.

Enums

Enums stands for Enumerated types. An enum is a user defined type that allows creations of customs data types that hold custom values. An example is below:

```
enum Condition
{
	Working,
	NotWorking,
	Finished,
	Unknown
};
```

This defines an enum called Condition where each of the possible types are defined as possible states. You create and use an enum like this:

```
#include <stdio.h>

enum Condition
{
	Working,
	NotWorking,
	Finished,
	Unknown
};

int main()
{
	//Created
	enum Condition programCondition;

	//Assigned a value
	programCondition = Working;
	//Comparison
	if (programCondition == Working)
	{
	puts("All is good!");
	}
}
```

In this program is shows how you can create an enum, how you can assign it a value and how you can compare it’s value.

Enums are most commonly used like boolean with extra context and features. It improves readability and adds extra states opposed to the binary nature of booleans values. Enums are very good at keeping track of data with limited well-defined values, like the current month or the day of the week.

Exercise

Create an enum that deals with the days of the week and prints out whatever day has been assigned (Hint: A switch-case is very useful for checking current day)

Solution

Something like this:

```
#include <stdio.h>

enum WeekDay
{
        Monday,
        Tuesday,
        Wednesday,
        Thursday,
        Friday
};
enum Condition currentDay;

int main()
{
        currentDay = Friday;

        switch (currentDay)
        {
        case Monday:
        puts("It's Monday!");
        case Tuesday:
        puts("It's Tuesday!");
        break;
```

```
        case Wednesday:
        puts("It's Wednesday!");
        break;
        case Thursday:
        puts("It's Thursday!");
        break;
        case Friday:
        puts("It's Friday!");
        break;
        }
}
```

Unions

Unions are a special data type that allow the programmer to store different data types in the **same memory location**. You can define a unions with many data members, **but only one** variable can hold a value at one time. These are an efficient way of using the same memory location for different uses.

The structure of a union is like so:

```
union Name
{
        //Variables
        int i;
        int y;
        char word[20];
        _Bool alive;
};
```

And multiple copies can be created much like how you can make versions of a structure or an enum. This is done like this:

```
union Name t1;
```

This **size** of the unions is as **large as the largest variable**, not as big as all the values together giving an efficient way to store variables one at a time opposed to a structure. Comparing the two the union above is *20 bytes* in size, a structure with exactly the same variables would be *32 bytes*. This would amount to a big space reduction if used on a much larger scale.

The variables in a union are accessed using the member access operator (.) and used like this:

```
union Name example;

example.i = 10;
printf("i before assigned another variable a value: %d\n", example.i);

example.y = 25;
printf("i after assigned another variable a value: %d\n", example.i);
printf("y value for reference: %d\n", example.y);
```

```
strcpy(example.word, "Hello!");
printf("i after assigned a string a value: %d\n", example.i);
printf("y after assigned a string a value: %d\n", example.y);
printf("String variable: %s", example.word);
```

Output

```
>i before assigned another variable a value: 10
>i after assigned another variable a value: 25
>y value for reference: 25
>i after assigned a string a value: 1819043144
>y after assigned a string a value: 1819043144
>String variable: Hello!
```

As you can see changing one value effects every other variable, this is the downfall of unions if not used correctly and need to managed very carefully so errors do not occur due to changing variables.

Variable argument lists

You might stumble upon a problem that might require a function with many parameters, this is where variable arguments comes it because it allows a dynamic amount of variables passed as a parameters.

Note:
A new included is required for this:

```
#include <stdarg.h>
```

Add this when using variable arguments.

You design the method like so:

```
void function(int noOfVariables, ...)
..
..
```

Where there is always one variable defining the number of variables followed by the variables. The extra variables are assigned to something called a *va_list,* these lists are manipulated with these functions:

```
va_start(va_list valist, int numberOfVaribles);
```

va_start effectively takes the extra variables and places them in the va_list

```
va_arg(valist, type);
```

va_arg takes the next variable and returns it, it doesn't know if the current integer is the last so the program setup needs to set up to make sure it doesn't overflow.

```
va_end(valist);
```

va_end simply cleans up the memory for the va_list.

Below is an example that finds the largest value:

```
#include <stdio.h>
#include <stdarg.h>

int max(int n, ...)
{
	int largest = 0;
	//Creates an assigns
	va_list valist;
	va_start(valist, n);

	//loops through variable list
	for (int i = 0; i < n; i++)
	{
	//Grabs the next arg
	int nextVar = va_arg(valist, int);

	//Compares size of values || The first value is assigned to be the largest value
	if (nextVar > largest || i == 0)
	{
	largest = nextVar;
	}
	}

	//Frees up memory
	va_end(valist);
```

```
        return largest;
}

int main() {
        printf("Largest: %d\n", max(6, -2,3,4,5,66,10));
         printf("Largest: %d\n", max(3, 7, 2, 1));
}
```

Output

```
> Largest: 66
> Largest: 7
```

This program shows how the dynamic nature of the variable list can be very useful.

Exercise

Create a program using variable lists that returns the averages (product of all numbers/how many numbers) of **int** variables provided.
Create a method called ‘average’ that returns a *float*, this will be your variable function. And don’t forget to include:

```
#include <stdarg.h>
```

Solution

Something like this:

```
#include <stdio.h>
#include <stdarg.h>

double average(int n, ...)
{
        double total = 0.0;

        //Creates variable list
        va_list vaList;
        va_start(vaList, n);

        //Grabs each variable
        for (int i = 0; i < n; i++)
        {
```

```
        //Same as: "total = total + va_arg(vaList, int);"
        total += va_arg(vaList, int);
        }

        //Clean up!
        va_end(vaList);

        double avg = total / n;
        return avg;
}

int main()
{
        //%f for a float
        printf("Average: %f\n", average(4, 1,2,77,4534));
}
```

This is very similar to the max value example above, it has the va_start, va_arg and va_end.

Exercises chapter 4

1. What function is needed to copy a string into a suitable variable?
2. What are the two ways a structure can be nested?
3. The keyword ‘typedef’, what is it used for?
4. What is Enum short for?
5. What is the drawback of a union structure?
6. What determines the size of a union?
7. What does a va_list hold?
8. What is used to show a function is to use an argument list?
9. What operator is used access members of a structure?
10. How does a structure and array differ?

Chapter 5

Advanced Features

Header files

A header file is a list of function, variable and macro definitions that can be included and used in different files. A header file is specified by the **.h** filename extensions. They are included in other files by using the **#include** (pre-processor directives) and the name of the file ("header.h"). There are custom header files that programmers can create and also built in header files that come with the compiler, much like "**stdio.h**" that we have seen in the previous section.

To create a header file all you need to do is create a file with the **.h** file extension, this can be done in something as simple as notepad, this header file can now be included with another file to link and use the features of the header file.

Below is an example:

header.h

```
int x = 10;

int Function()
{
	return x;
}
```

Can he included and used like so:

```
#include <stdio.h>
#include "header.h"

int main()
{
	printf("%d", Function());
}
```

Output

```
>10
```

This allows programmers to make modules that can be reused between projects.

A possible use of headers is below:

```
int add(int num1, int num2)
{
	return num1 + num2;
}
int sub(int num1, int num2)
{
	return num1 - num2;
}
int div(int num1, int num2)
{
	return num1 / num2;
}
int mul(int num1, int num2)
{
	return num1 * num2;
}
```

Once again, the calculator example returns, a header could be a list of functions like this that could then be included to allow the use of these functions.

Pre-Processor Directives

Pre-processing are used to give the compiler a type of command, you come across a command quite a few times already is the '#include' directive that tells the compiler to include a certain header file.

Below is a list of the processor directives, we will go through how they all work:

Directive	Description
#define	Used to define a macro

#include	Inserts a particular header from another file.
#undef	Removes the effect of #define
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends pre-processor conditional.

#define and #undef

These commands are used when global values are used to increase readability, for example:

```
#define LOOP_NUMBER 2;
```

Could be defined and used like so:

```
for (int i = 0; i < LOOP_NUMBER; i++)
{
	//Loop code
}
```

And the for loop will loop twice, but you cannot change the value like you would with a normal variable, it's **constant through the life of the program** unless **#undef** is used that will undefine the value set.

#undef is normally used to overwrite build in directives where you would undefine a value and then set a value of your choosing.

#ifdef and #if

#ifdef and **#ifndef** will return true and false respectively if a macro is defined, this most common use for this is checking if the *-DDEBUG* flag has

been set and running code in DUBUG mode that has no use in the finished program. It's used like this:

```
#include <stdio.h>

int main()
{
#ifdef DEBUG
        puts("DEBUG!");
#endif
}
```

This code will only run if the code is flagged as debug mode. This can be very useful for a programmer.

#if, #else and #elif work exactly the same as if statements but are used to check the macro with arithmetic expressions rather that checking for its existence.

Error Handling

Error handling is a very important section, because a programmer needs to make sure his program is prepared to deal with expected and unexpected errors. C does not provide direct support for error handling but allows access to some low level functions. Most functions will return -1 or NULL if there is an error and set the **errno** error code, **errno** is a global variable that holds the last returned error code.

You will need to include the <errno.h> header file to use these error handling functions.

```
#include <errno.h>
```

There are a few functions that allow you to use an understand error codes.

perror()
Function displays the string you pass and attaches to the end the textual representation of the error code stored in **errno.**

strerror()

Returns the pointer to the textual representation of the **errno** value. This can be used to save the error feedback.

stderr file stream
stderr is used to output an error to the console.

Usage is below:

```c
#include <stdio.h>
#include <errno.h>

extern int errno;

int main() {

	FILE * file;
	int errnum;

	//Looks for file
	file = fopen("youWillNotFindMe.txt", "rb");

	//If the file returned an error
	if (file == NULL)
	{
	//Grabs error code
	errnum = errno;

	//Prints error code
	fprintf(stderr, "Value of errno: %d\n", errno);

	//Returns string provided + : and Error code message
	perror("Error printed by perror");

	//Grabs the error text
	char* errorMsg[] = { strerror() };
	printf("Error msg test print: %s", *errorMsg);
	}
	else
	{

	fclose(file);
	}

	return 0;
}
```

Above is usage of some of the error handling examples. Go over and understand it error handling is very important.

Below is another example of preventing the classic divide by zero error:

```
#include <stdio.h>
#include <errno.h>

extern int errno;

int divide(int x, int y)
{
        if (y == 0)
        {
        //Prints error
        fprintf(stderr, "Diving by zero error..!");

        //Returns error code
        return -1;
        }
        else
        {
        //If valid returns even
        return x / y;
        }
}

int main()
{
        int returnCode = divide(0, 2);
}
```

Type casting

Variables has defined types, but there are situations where you'll need to convert from one type to another. Very commonly would be the conversion between *integer* to a *float* or *double*

The general format is like so:

```
(type)varToCast
```

This can be used like so:

```
int main()
{
	int integer = 3;
	float decimal = 1.5f;

	int result = (int)decimal + integer;
	printf("%d\n", result);
}
```

Output
> 4

What happens is any decimal values are truncated (removed) and the remainder is added to the integer value.

You can also upgrade a value, this example shows when a integer is upgraded to a float:

```
int main()
{
	int integer = 3;
	float decimal = 1.5;

	//Explicit casting
	float result = decimal + (float)integer;
	printf("%f\n", result);

	 //Implicit casting
	float result = decimal + integer;
	printf("%f\n", result);
}
```

This can be done either implicitly or explicitly. Implicit is when the complier **automatically** converts the variable and explicitly is when you use the casting operator. It is however very good practice to specify a cast wherever it is necessary.

Memory management

C allows programmers to dynamically manage memory. This functionality is provided by:

```
#include <stdlib.h>
```

Memory management requires the use of a few functions

- **void *calloc(int num, int size);**
  - This function allocates an array (size specified by **num**) and the size of each allocated specified by **size**
- **void free(void *address);**
  - This function releases memory, the location is specified by **address.**
- **void *malloc(int num);**
  - This function works like **calloc** but leaves the locations uninitialized.
- **void *realloc(void *address, int newsize)**
  - This function re-allocates memory to the size specified in **newsize**

Below is an example of using **malloc** or **calloc** to allocate memory for a string:

```
#include <stdio.h>
#include <stdlib.h>

#define WORD_SIZE 20

int main()
{
        char* word;

        //Allocates memory
```

```
        word = malloc(WORD_SIZE * sizeof(char));

        //or

        word = calloc(WORD_SIZE, sizeof(char));

        //Copies values over
        strcpy(word, "Hello nice to meet you!");

        //Printing
        printf("The string is: \"%s\"", word);

        //Deletes memory
        free(word);

}
```

This can be used to do quite complex things, like take in a user input and store it in an array exactly the size for that string:

```
#include <stdio.h>
#include <stdlib.h>

#define WORD_SIZE 20

char* getInput()
{
	//Will be deleted
	char temp[50];

	//User input
	printf("Please enter your first name: ");
	scanf("%s", temp);
	puts("");

	//strlen() finds lenght of a string
	int len = strlen(temp);
	//Dynamic allocation of memory
	char* perfectSizeWord;

	//Allocates memory!
	perfectSizeWord = calloc(len, sizeof(char));

	//This command would not work, it takes the memory location of temp and just copies it so when
	//this function goes out of scope so does perfectSizeWord
	///perfectSizeWord = temp

	//Copies the values properly
	for (int i = 0; i < len; i++)
	{
	perfectSizeWord[i] = temp[i];
	}

	//Size comparason
	printf("Size of temp var:		%d\n", sizeof(temp));
	printf("Size of new perfect var:	%d\n", sizeof(perfectSizeWord));
```

```
        return perfectSizeWord;
}

int main()
{
        //Gets returned variables
        char* word = getInput();

        //Prints
        printf("The word is:            %s\n", word);
        //Delets memory
        free(word);
}
```

The program above is large, take your time and look through. Try and implement it and understand how it works.

Exercise

Use error handling techniques above to handle the error for allocating memory with **malloc** or **calloc**. To simulate a an error just make *word = NULL;* after the dynamic memory allocation.

Use this program frame below to help:

```
#include <stdio.h>
#include <errno.h>

int main()
{
        char* word;

        //Allocates memory
        word = malloc(20 * sizeof(char));

        //ERROR SIMULATE
        word = NULL;

        //-----------------PUT ERROR CODE HERE----------------------

        //Deletes memory
```

```
        free(word);
}
```

Solution

Very simply all that needs including is:

```
#include <stdio.h>
#include <errno.h>

int main()
{
	char* word;

	//Allocates memory
	word = malloc(20 * sizeof(char));

	//ERROR SIMULATE
	word = NULL;

	if (word == NULL)
	{
	fprintf(stderr, "Error: Unable to allocate the memory!");
	}
	//Deletes memory
	free(word);

}
```

Exercises chapter 5

1. How do you tell the compiler to use other files and libraries?
2. What file extension does a header file use?
3. What is #define used for?
4. What #include is required for error handling?
5. What is contained in the global variable **errno**?
6. What does strerror() return?
7. What file stream is required to print out an error?
8. Typecasting; what is it used for?
9. What is the different between **calloc** and **malloc**?
10. What normally will a function that encounters an error return?

Answers Chapter 2

1. Float or Double
2. True (1) and False (0)
3. A statement designed to check if a condition is true or false
4. Declaration, Conditional and Iteration.
5. Skips a full iteration
6. Yes it is, and by using the void keyword
7. Theoretically unlimited
8. Printf uses formatting id's, puts does not using formatting id's and puts automatically adds a '\n' add the end of the string
9. *Integer value*
10. *int arrayLength = sizeof(lotsOfNumbers) / sizeof(lotsOfNumbers[0]);*

Answers chapter 3

1. A memory location for a certain variable type.
2. The dereferencing operator (*).
3. It will move 4bytes along to the next memory location.
4. Only once.
5. The variable is stored in the much faster to access register memory.
6. fprintf() and fputs().
7. Opens the file for both reading and writing and chops the file size down to zero if it exists and creates it if it does not exist.
8. If the pointer is not NULL.
9. *Is called the reference operator and is used to access or return the raw memory address.*
10. *The size of the array preferably as a* ***size_t*** *variable.*

Answers chapter 4

1. strcpy() is needed.
2. Internally and externally defined.
3. Used to give variables and user defined structures custom names.
4. Enumerated type.
5. Only on variable in a unions can hold a value at one time.
6. A union is as big as its biggest variable.
7. The list of variables passed into a variable function.
8. An ellipsis at the end of the parameter list.
9. *The member operator denoted by a full stop (.)*
10. *An array stores variable of the same type, a structure stores variables of any type.*

Answers chapter 5

1. Using the #include directive
2. A header file uses “.h”
3. #define is used to create global definitions of constant values
4. #include <errno.h>
5. The last error code that was thrown will be stored there.
6. The textual representation of the error code stored in errno
7. The file stream is: “stderr”
8. Used to change one variable into another
9. *When **calloc** allocates memory it initialises values, **malloc** does not*
10. *NULL or -1*

C#

Step-By-Step Guide To C# Programming For Beginners

TABLE OF CONTENT

Chapter 1

Setup and Introduction

History of C#

C# first appeared in 2000 when it was announced by Microsoft, C# is a Microsoft owned language stemming from the C family. It was designed by a team run by Anders Hejlsberg. Initially it proposed the language would be called "*Cool"* standing for *"C-Like Object Orientated Language"* with Microsoft finally deciding to call it C#. The hash symbol is used but pronounced as 'sharp', this was chosen on purpose because of the lack of sharp symbol on many of the keyboards at the time. The sharp symbol also resembles four plus symbols in a 2x2 grid, strongly implying that C# is a later increment of its brother language C++.

Below is a timeline of the C# language and what inclusions each increment added or improved (All features will be explained later in the tutorial):

Jan 2002
C# 1.0
- -Type safe programming language
- -Modern
- -OOP (Object Orientated Programming)

Nov 2005
C# 2005
- -Generics
- -Partial classes
- -Anonymous types
- -Iterators
- -Nullable types
- -Static classes

Nov 2007
C# 2007
- -Improvements of previous additions
- -Extensions methods

-Expression trees
-Partial methods

April 2010
C# 4.0

-Dynamic binding
-Naming and optional arguments

Aug 2012
C# 5.0

-Asynchronous Programming

A huge addition allowing programmers to create multithreaded applications

-Caller info attributes

July 2015
C# 6.0 (Version in this tutorial)

-Exception filters
-String interpolation
-Static type members into namespaces
-Complier as a service

Why C#?

C# has many advantages over other similar languages like Java and Python.

- C# is incredibly type safe meaning when a variable is created a type must be defined, that means another type cannot be housed in this variable.
- C# has full control over memory leaks that are a big worry for many C++ programmers.
- C# also provides a rich library of features that's allows implementing extra features easy and straightforward
- Any system with the .NET framework will run this application. However, the framework is mainly supported on Windows, meaning Linux and Mac support is thin or entirely non-existent.

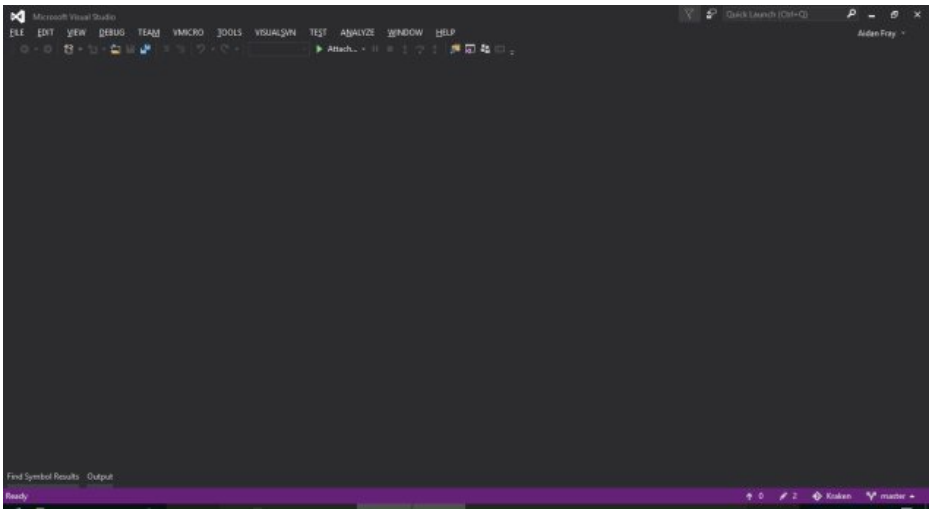
Running C# on Windows

Running C# in my opinion is the best way and pretty much only way to code and work with C#, it a Microsoft created language and they have done everything they can to make it easy to work with on a Windows machine, even if it means ignoring other platforms entirely.

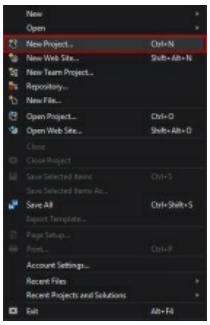
Mac Users:
There has recently been a release of Visual Studio for Mac, but it is rudimental and does not have the features that Windows Visual Studio has, it is therefore heavily advised to work with C# on a Windows machine. But, there are ways to have a Windows OS to run on a Mac, look up “Virtual Box” by Oracle, it is a program that allows you to run a virtual Windows machine.

To work with C#, you will need to download **Visual Studio**, it can be downloaded directly from Microsoft. Please run the installer and install Visual Studio.

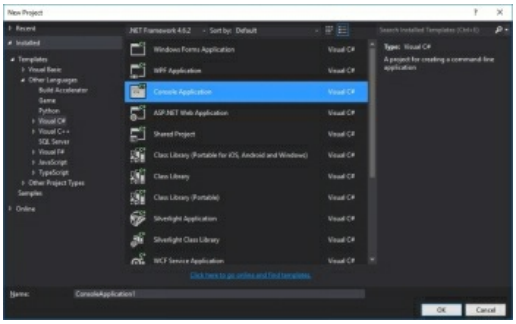
When you have a fully working instillation you should be greeted with this screen:



Click File and then select “New Project”



You'll be presented with the screen below, select ConsoleApplication and give it a name below:



Click "Ok". You'll then be presented with some code where you'll be ready to program.

Chapter 2

Classes and Object Orientated Programming

Basic classes

Classes are used to store variable and functionality relevant to an object, for example there can be a class created for a ‘Book’, it would have information such as number of pages, genre and cover colour and functionality like ‘loan book’ or ‘return book’. This is where the whole useful and real world related nature comes from with classes, and the word **object** in the OOP.

Before we go into how to create a class there are accessor keywords that need to be understood:

11. private — Cannot be accessed outside the scope of the class
12. public — Can be accessed anywhere
13. protected — Can be accessed from inherited classes

You define a class like so:

```
class Book
{

}
```

With all its variables and functions defined within the brackets, a fleshed-out version is below:

```
class Book
{
      public Book(int numberOfPages, int currentPage, string colour, string
name)
         {
            _numberOfPages = numberOfPages;
            _currentPage = currentPage;
```

```
            _colour = colour;
            _name = name;
        }

    private int _numberOfPages;
    private int _currentPage;
    private string _colour;
    private string _name;

    public string changeMe;

    public int returnCurrentPage()
        {
        return _currentPage;
        }

}
```

There are a few things to note. The function in the box is knowns as a Constructor (Explained in this chapter), it is called when creating a new object. This class contains all the member variables and member functions, they are both accessed by using the member operator (.). In this case all the variables are marked as ‘private’ meaning they cannot be accessed from outside the class, this is where getters and setters (Accessor methods) come into use, the function ‘returnCurrentPage’ is a getter and allows access to the currentPage value, this is the recommended way of dealing with variable access, it allows a platform for the programmer to control access to variables.

An object is created below, with some of its functions and variables being accessed:

```
class Program
{
    static void Main(string[] args)
        {
        Book book1 = new Book(200, 32, "Red", "Harry Potter");

        book1.changeMe = "Test";
        Console.WriteLine(book1.changeMe);

        int currentPage = book1.returnCurrentPage();
```

```
        Console.WriteLine(currentPage);
        }
}
```

Output
>Test
>32

Recap

11. Classes provide an intuitive way to store data on real world objects
12. They allow functionality to be built up around the data
13. Are used as a cookie-cutter way of making lots of containers for data
14. Accessor methods allow access to private variables
    a. Getters return the variables value
    b. Setters allow to program to change the value

Properties

As mentioned in the previous chapter were Accessor methods, properties allow an easily readable solution to accessing member variables in a tiny compact syntax. Properties also contain the variable definition. They are defined like so:

```
public int currentPage
{
        get
            {
            return currentPage;
            }
        set
            {
            currentPage = value;
            }
}
```

And called like this:

```
Book book1 = new Book();
```

```
book1.currentPage = 2; //2 is 'value' in the set
Console.WriteLine(book1.currentPage);
```

This performs exactly the same role as the accessors in the previous chapter but with a much more concise syntax, for comparison:

```
private int _currentPage;
public int returnCurrentPage()
{
        return _currentPage;
}
```

Is equal to:

```
public int currentPage { get; }
```

As you can see it's a much smaller set of code.

A more advanced use of the setter could be to validate the input. For example, you can't have a current page that is larger than the total number of pages. This code could look like this:

```
class Book
{
      int totalPages = 500;
      public int currentPage
        {
         get
            {
            return currentPage;
            }
         set
            {
            //Validation check
            if (value < totalPages)
                {
               currentPage = value;
                }
             else if(value == totalPages)
                {
               Console.WriteLine("You've finished the book!");
                }
```

```
            else
                {
                //Error message
                Console.WriteLine("Invalid current page!");
                }
            }
        }

}
```

This gives the programmer the ability to write code that accounts for many possibilities.

Static

As mentioned in previous chapters member functions allow classes to hold functionality that is somehow related to the data contained within, this allows classes to be almost self-contained programs this along with member variables gives classes their functionality.

Member functions and variables can be defined as ‘static’, meaning the object is ‘always there’. This is used when behaviour will not change between all instances of the class and it means that the behaviour or variable is shared between them all.

Note: static functions can only access static variables

```
using System;

namespace ConsoleApplication1
{
   //
   class Program
    {
      static void Main(string[] args)
         {
          Example e = new Example();
              e.Instance();
          e.Static() //Error

          Example.Instance(); //Error
          Example.Static();
         }
    }


   class Example
    {
      public void Instance()
         {
          Console.WriteLine("Instance!");
         }
      public static void Static()
         {
```

```
            Console.WriteLine("Static!");
            }
        }
}
```

The example above shows how the different types of functions are accessed and dealt with.

Below is an example of static working within a ‘real-world’ example:

```
using System;

namespace ConsoleApplication1
{
    //
    class Program
     {
       static void Main(string[] args)
          {
           int i = Person.WorkOutAge(new DateTime(1990, 10, 15));
           Console.WriteLine(i);
          }
     }

    class Person
     {
       public Person(string _name, int _age)
          {
              name = _name;
              age = _age;
          }

       public string name { get; }
       public int age { get; }

       public static int WorkOutAge(DateTime birthday)
          {
           //Time since
           TimeSpan difference = DateTime.Now - birthday;

           double years = difference.TotalDays / 365;
```

```
            return (int)years;
            }
        }
}
```

This program uses a static method to work out the age of somebody using their birthday, this allows the method to hold relevant functions without a separate one being created per instance.

Constructors and Destructors

When creating an instance of a class a constructor is used, the language provides a default constructor, but a custom constructor can be created, you may have seen the functions without a return type and with exactly the same name as the class, to continue with our person class example:

```
class Person
{
      public Person()
        {
           Console.WriteLine("Person created");

        }
}
```

This is a custom constructor that just prints when a Person is made. Normally constructors are used to pass in data to member variables when an instance of the class is created.

You can also have as many constructors as you like, however they all need varying parameters types and number of parameters (this is known as a function signature) so with the example above I could not define another constructor with no parameters, this is known as **overloading.**

For example:

```
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
        {
         Person p1 = new Person("John", 32);
         Person p2 = new Person("James", 50, "Train Driver");
        }
    }

   class Person
    {
      public Person(string _name, int _age)
```

```
        {
            name = _name;
            age = _age;
         job = "N/A";
        }
      public Person(string _name, int _age, string _job)
        {
            name = _name;
            age = _age;
            job = _job;
        }

      public string name { get; }
      public int age { get; }
      public string job { get; }

    }
}
```

The highlighted sections show the use of the overloading, the compiler automatically detects which constructor to use depending on the types in the parameter list.

Just like the constructor there is a destructor that performs the opposite role of the constructor that instead of dealing with creation code the destructor deals with clean-up code.

A destructor is created like so:

```
class Person
{
        ~Person()
        {
         Console.WriteLine("Person deleted!");
        }
}
```

This method **cannot** be called manually but is managed by the internal Garbage Collector that is a built-in part of the compiler designed for automatic memory management.

This can be demonstrated like this, if a method is called that creates an instance to Person and then the program goes out of scope for the main method the garbage collector will be called to clean up the memory

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
        {
            Function();

                    }

      static void Function()
        {
         Person p1 = new Person();
        }
    }

   class Person
    {
        ~Person()
        {
         Console.WriteLine("Person deleted!");
         Console.ReadKey();
        }
    }
}
```

Output

```
>Person deleted!
```

Exercise

Create a class that holds data for Pets, the class will need a Name, Age and Type of animal. There also will need to be a Constructor, Destructor and a member function that prints all the data for the pet.

Test it by having a dog called Jose that is 1 year old, A cat called Besse that is 5 and a parrot called James that is 10.

Solution

Something like this:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
          Pet p1 = new Pet("Jose", 1, "Dog");
             p1.Print();

          Pet p2 = new Pet("Besse", 5, "Cat");
             p2.Print();

          Pet p3 = new Pet("James", 10, "Parrot");
             p3.Print();

          Console.ReadKey();
         }
    }

   class Pet
    {
      public Pet(string _name, int _age, string _type)
         {
             name = _name;
             age = _age;
             type = _type;
         }

      string name { get; }
      int age { get; }
      string type { get; }

      public void Print()
         {
```

```
            //{name} is string interpolation
            Console.WriteLine($"Name:   {name}",name);
            Console.WriteLine($"Age:    {age}",age);
            Console.WriteLine($"Type:   {type}",type);
            Console.WriteLine();
           }
    }
}
```

Inheritance

Inheritance is a language function that allows classes to take on characteristics of another, inheritance is done like so:

```
class Dog : Animal
{

}
```

This means that any function or variable marked as “Public” or “Protected” (Protected is explained in the next chapter) Dog will have its own version of or access to, this allows quick creation of new types of classes without having to repeat function and variable definitions.

The example below should how it can be used:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
          Dog d1 = new Dog("Rocko", 3, "Blue One", "Red");

          d1.PrintName();
         }
    }

   class Animal
    {
      public Animal(string _name, int _age)
         {
             name = _name;
             age = _age;
         }

      public string name;
      public int age;

      public void PrintName()
```

```
        {
         Console.WriteLine(name);
        }
    }

   class Dog : Animal
    {
      public Dog(string _name, int _age, string favBone, string kColour) :
base(_name, _age)
        {
            FavouriteBone = favBone;
            kColour = KennelColour;
        }

      private string FavouriteBone;
      private string KennelColour;
    }
}
```

There are a few points to mention, the first is the example showing how functions are inherited, Dog does not have a definition for “PrintName” but because it inherits “Animal” it has its own definition.

The other point it is the “base(_name, _age)”, this is how you call the base constructor, in this case it’s calling the Animal constructor above.

Encapsulation

Encapsulating is the process of enclosing one or more items within a physical or logical package and in the context of OOP is how access to member variables is managed.

We’ve seen three of these before but this section will go into more detail about the sections below:

11. Public
12. Protected
13. Private
14. Internal
15. Protected internal

Public

Public simply allows full access to all member objects.

For example:

```
class Program
{
        static void Main(string[] args)
           {
            Public p = new Public();

            //value can be accessed and changed no problem
                p.value = 3;
           }
}

class Public
{
        public int value;
}
```

Private

Is the complete opposite of public where on internal calls to a variable are allowed, this means if we want to change variable we need to use accessor function:

```
class Program
{
        static void Main(string[] args)
           {
            Private p = new Private();

            //Error here!
                p.value = 3;
           }
}

class Private
{
        private int value;
```

```
}
```

Protected

Protected is a little different, protected involves inheritance is when a class inherits functions and variable from another (Mentioned previously).

Internal

This allows a class to exposed both member functions and variables to objects in the current assembly, essentially this is just any functions contained within the application (.exe or .dll)

Protected Internal

Protected internal is very like internal but with a few differences, it does involve inheritance much like protected.

Note:
You may be thinking what if an access modifier isn't specified? Well, If that is the case (and it's completely valid), it will default to **private**.

End of chapter Quiz

11. What does OOP stand for?
12. What are functions called that allow changes to be made to private member variables?
13. Explain the role of a property?
14. Defining a classes member variable as 'static' achieves what?
15. What can a static function only access?
16. How is the destructor called by the programmer?
17. What section of the complier automatically deals with memory management?
18. Where can you access a variable marked as internal?
19. What is the technical name for a newly created version of a class?
20. What is the name for the use of multiple constructors?

Answers

11. Object Orientated Programming.
12. Getters (Accessor functions).

13. A small more concise syntax for getter, setters and variable definition.
14. This means the variables presence is constant between every instance of the class.
15. Static variables.
16. It is not, the destructor cannot be manually called.
17. The garbage collector
18. In the same assembly, i.e. in the same .exe file or .dll
19. An instance of a class
20. Overloading a class

Chapter 3

Advanced Class structures

Polymorphism

The word polymorphism means having many forms, in the context of OOP is often described as “one interface, many functions”.
There are two types of polymorphic design, **static** and **dynamic** polymorphism.

Static Polymorphism

Static polymorphism means the response to a function is determined at compile time (i.e. before the program is run), this process is called ‘Early binding’ and this can be done in one of two ways:

11. Function overloading
12. Operator overloading

Function overloading

In function overloading you can have the same definition of a function is the same scope, this is exactly the same to constructor overloading.

So, for example, this class definition would be valid:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
          Interface i = new Interface();

          //Printing integer
              i.Print(1);

          //Printing float
              i.Print(2.0f);
```

```
            //Printing double
                i.Print(4d);

            //Printing long
                i.Print(4L);
            }
        }

        class Interface
        {
          string printStr = "Print!";

          public void Print(int i)
            {
             Console.WriteLine(i);
            }

          public void Print(double i)
            {
             Console.WriteLine(i);
            }

          public void Print(float i)
            {
             Console.WriteLine(i);
            }

          public void Print(long i)
            {
             Console.WriteLine(i);
            }
        }
}
```

As you can tell there are 4 definitions of “Print()” but all with different parameters.

Dynamic overloading

C# allows the creation of an abstract class; an abstract class can act as a template or a design for other classes to inherit and adapt for the

circumstance they are designed for. Derived classes then can take abstract functions and adapt their behaviour, however they’re some rules about abstract classes:

11. You cannot directly create an instance of an abstract class
12. You cannot declare an abstract method that is not housed in an abstract class
13. There is also a keyword known as ‘**sealed**’ and sealed classes cannot be inherited, abstract methods therefore cannot be defined as sealed

Below is an example that demonstrates the use of an abstract class:

```
using System;
namespace ConsoleApplication1
{
   abstract class Shape
    {
      public abstract int Volume();

      public int height { get; set; }
      public int width { get; set; }
    }

   //Creates a 2D Rectangle
   class Rectangle : Shape
    {
      public Rectangle(int w, int h)
        {
            height = h;
            width = w;
        }

      public override int Volume()
        {
         return height * width;
        }
    }

   //Creates a 3D Cylinder
   class Cylinder : Shape
    {
      public Cylinder(int w, int h)
        {
            height = h;
            width = w;
        }

      public override int Volume()
        {
         //(π * (width / 2)^3) * height
```

```
        return (int)(Math.PI * Math.Pow(width / 2, 3)) * height;
        }
    }

    class Program
    {
      static void Main(string[] args)
        {
        Rectangle rec = new Rectangle(10, 10);
        Console.WriteLine(rec.Volume());

        Cylinder cylin = new Cylinder(10, 10);
        Console.WriteLine(cylin.Volume());

        Console.ReadKey();
        }
    }
}
```

A few things to note in this, the area surrounded by the rectangle is the abstract class definition, where an abstract Volume() class is defined, this is then taken in the other methods and **overloaded,** notice the 'overload' keyword that is used to create a custom definition of Volume() where each class has a custom design for Volume that is then called in the Main method. An abstract class requires that the method that inherits it has to create an overloaded version. This means abstract classes are designed to be very strict plans for the shape and layout of a method.

There is also another type of dynamic overloading, it is known as a 'Virtual' this is very similar to abstract but is an optional version, where a definition is the base class can optionally be overridden, below is an example:

```
using System;
namespace ConsoleApplication1
{
   class Human
    {
      public virtual void SayHello()
         {
          Console.WriteLine("Hello!");
         }
    }

   class American : Human
    {
      //No override needed as SayHello is valid
    }

   class Columbian : Human
    {
      public override void SayHello()
         {
          Console.WriteLine("Hola!");
         }
    }

   class French : Human
    {
      public override void SayHello()
         {
          Console.WriteLine("Bonjour!");
         }
    }

   class Program
    {
      static void Main(string[] args)
         {
          American a = new American();
             a.SayHello();
```

```
            Columbian c = new Columbian();
                c.SayHello();

            French f = new French();
                f.SayHello();

            Console.ReadKey();
            }
        }
}
```

As you can see each class inherits from the Human class and some override the virtual class but some like the American class don't need to override it, this gives the option to changes classes if need be.

Interfaces

Interfaces are very structured, they are a blue print that every inherited method must follow, much like a contract. An interface is much like an abstract class, but how an abstract class should be inherited and possibly extended an interface should just be implemented. Interfaces allow the separation of function definitions from implementation and improves readability and design.

A interface is defined like so, each method or property that is to be created is put in a kind of list, an interface cannot contain any implementation.

```
interface ITransaction
{
      //Methods
      void Process();
}
```

Below is an example of using an Interface with the example used involving Vehicles:

```
interface IVehicle
{
        string Colour { get; }
        string EngineType { get; }
        int NumberOfWheels { get; }
        int Gas { get; }

        void StartEngine();
        void StopEngine();
}

class Car : IVehicle
{
        public string Colour { get; }
        public string EngineType { get; }
        public int NumberOfWheels { get; } = 4;
        public int Gas { get; }

        public void StartEngine()
          {
           Console.WriteLine("Started Engine!");
          }

        public void StopEngine()
          {
           Console.WriteLine("Stopped Engine!");
          }

        public Car(string colour, string engineType, int gas)
          {
              Colour = colour;
              EngineType = engineType;
              Gas = gas;
          }
}
```

As you can see, Car has to implement all of the items defined in the IVehicle

interface, this gives a kind of “plug and play” effect when creating new classes that relate to Vehicles because the design and layout is strictly enforced.

Exercise

Using the previous example implement a new Class called Bike using the IVehicles Interface, get a feel for how the interface works.

Solution

The implemented class should look like so:

```
class Bike : IVehicle
{
        public string Colour { get; }
        public string EngineType { get; }
        public int NumberOfWheels { get; } = 2;
        public int Gas { get; }

        public void StartEngine()
           {
            Console.WriteLine("Started Engine!");
           }

        public void StopEngine()
           {
            Console.WriteLine("Stopped Engine!");
           }

        public Bike(string colour, string engineType, int gas)
           {
               Colour = colour;
               EngineType = engineType;
               Gas = gas;
           }
}
```

Namespaces

Namespaces are used to separate definitions, it makes so names defined in

one namespace do not clash with another.

A namespace is defined like so:

```
namespace Namespace1
{

}
```

And to call another namespace's function you use the member operator (.), this is done like this:

```
Namespace1.Class c = new Namespace1.Class();
```

Note: Namespaces cannot directly contain functions or variable definitions.

Below is an example of using another namespace:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      public class SameName
         {
          public void Function()
              {
              Console.WriteLine("Main!");
              }
         }

      static void Main(string[] args)
         {
          SameName s1 = new SameName();
              s1.Function();

          Namespace1.SameName s2 = new Namespace1.SameName();
              s2.Function();

          Console.ReadKey();
         }
    }
```

```
}

namespace Namespace1
{
   public class SameName
    {
      public void Function()
         {
          Console.WriteLine("Namespace1!");
         }
    }
}
```

Output
>Main!
>Namespace1!

Note how there are two classes with a class with the "SameName" and how you specify which one with the Namespace and the member operator.

Namespaces in other files or this file can be automatically specified by the **using** keyword, you may have noticed the:

```
using System;
```

That has appeared on the top of some of the code snippets, this is basically telling the program to use the "System" namespace for items like:

```
Console.WriteLine("");
```

And

```
Console.ReadKey();
```

Using can also be used to chop off the preceding namespace specifier:

Without the using statement

```
using System;
namespace ConsoleApplication1
{
```

```
    class Program
     {
       static void Main(string[] args)
          {
           Namespace1.Class c = new Namespace1.Class();
              c.Function();

           Console.ReadKey();
          }
     }
}

namespace Namespace1
{
    public class Class
     {
       public void Function()
          {
           Console.WriteLine("Namespace1!");
          }
     }
}
```

With the using statement:b

```
using System;
using Namespace1;
namespace ConsoleApplication1
{
    class Program
     {
       static void Main(string[] args)
          {
           Class c = new Class();
              c.Function();

           Console.ReadKey();
          }
     }
```

```
}

namespace Namespace1
{
   public class Class
    {
      public void Function()
         {
          Console.WriteLine("Namespace1!");
         }
    }
}
```

Notice how the Class reference is now shorter and nicer to read.

End of chapter Quiz

- What are the two main ways of implementing Polymorphism?
- What is it called when they're more than one versions of a contractor or function?
- Defining a class as sealed achieves what?
- What can you not do with an abstract class?
- What keyword is used when implementing an abstract or virtual method?
- Namespaces are used for what purpose?
- Can you create an abstract method that isn't housed in an abstract class?
- What is the main difference between an Interface and an abstract class?
- What operator is used when accessing the items of a namespace?
- How can you tell the complier to include another namespace automatically?

Answers

- Static and Dynamic polymorphism.

- Overloading.
- Means the class cannot be inherited.
- Directly create an instance of it.
- Overload.
- To separate names of classes and methods.
- No, all abstract methods need to be in an abstract class.
- Interfaces do not allow implementation to be added and are a blueprint to design new classes.
- The member operator (full stop).
- By adding a ‘using’ statement at the top of the program.

Chapter 4

Intermediate functionality

Operator overloading

C# has support for altering the functionality of many of the built-in operators, such as '+' or the member operator '.' we have come across previously. Operator overloaders are special functions that include the keyword 'operator' followed by the operator sign, exactly like any other function overloaded operators have a return type and parameter list.

In this example, we're going to create a class called 'Grade' and overload the plus operator. An operator overloader is defined like so:

```
public static Grade operator+(Grade a, Grade b)
{
}
```

Note: The operator function needs to be static and public.

The example below implements this overloaded operator:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
          Grade a = new Grade(30);
              a.Print();

          Grade b = new Grade(40);
              b.Print();

          Grade c = a + b;
              c.Print();

          Console.ReadKey();
         }
```

```
    }

    class Grade
    {
      int mark;

      public Grade(int m)
        {
            mark = m;
        }

      public void Print()
        {
         Console.WriteLine($"The mark is: {mark}", mark);
        }

      public static Grade operator+(Grade a, Grade b)
        {
         Grade c = new Grade(0);
            c.mark = a.mark + b.mark;
         return c;
        }
    }
}
```

Output

```
>The mark is: 30
>The mark is: 40
>The mark is: 70
```

The operator in this instance just takes the mark of each grade instance and pluses the result. This is a very basic example and operator overloading can be incredibly useful.

Exercise

So for example if we wanted to compare two grades at the moment if we added:

```
if (a == b)
{
```

```
        Console.WriteLine("Equal grades!");
}
```

And set both grades mark to exactly the same, let's say 30. It should print out "Equal grades"

Overload the "==" operator to compare both marks and see if they're equal

Note:

- The operator overload function should return a bool.
- The "!=" Operator also needs to be implemented alongside (This will just do the complete opposite)

Solution

The entire program should look like the example, the new additions are surrounded by the box.

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
          Grade a = new Grade(30);
              a.Print();

          Grade b = new Grade(30);
              b.Print();

          //Test
          if (a == b)
              {
              Console.WriteLine("Equal grades!");
              }

          Console.ReadKey();
         }
    }
```

```
  class Grade
   {
     int mark;

     public Grade(int m)
       {
           mark = m;
       }

     public void Print()
       {
        Console.WriteLine($"The mark is: {mark}", mark);
       }

     public static Grade operator+(Grade a, Grade b)
       {
        Grade c = new Grade(0);
           c.mark = a.mark + b.mark;
        return c;
       }

     public static bool operator==(Grade a, Grade b)
       {
        return a.mark == b.mark;
       }
     public static bool operator !=(Grade a, Grade b)
       {
        return a.mark != b.mark;
       }
   }
}
```

Output

```
>The mark is: 30
>The mark is: 30
>Equal grades!
```

Regular expressions

Regular expressions are a way of matching patterns in inputted text, the .NET frame work has a built in regular expression support. This will be an overview of how to use basic expressions, there are a lot of combinations to learn and doing some research after this chapter is advised.

Each regular expression needs to be constructed, below is a list of each character, operator and function that allows you to create a regular expression:

Anchors & Extras

Character	Description
^	Is a reference to the start of the word
$	Refers to the end of a string
\w	Refers to a word consisting of a-z, A-Z, 0-9 and an underscore
+	Refers to 0 or more occurrences
|	Used to specify ‘or’
[nnn]	Used to specify a character class to check with, i.e. [abcd] will only ruturn matches on **aabc** and no **dcab**

So for example the regex:

```
^\w+$
```

Looks intimidating I know, but it simply matches any word it is given, using the System.Text.RegularExpression we can match a pattern. The code below demonstrates it:

```
using System;
using System.Text.RegularExpressions;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
        {
         while (true)
            {
             //Change expression here!
            string regExpression = @"^\w+$";

            Console.WriteLine("Please input text");
            string input = Console.ReadLine();

            if (Regex.IsMatch(input, regExpression))
                 {
               Console.WriteLine($"{input} is a match!", input);
                 }
            else
                 {
               Console.WriteLine("No match!");
                 }

            Console.WriteLine();
            }

        }
    }
}
```

Copy over this program to a Visual Studio project and run it, try different inputs out. Anything that includes **just** A-Z, a-z, 0-9 and an underscore will match, but spaces and punctuation for example will not return a match.

Note the use of the “@” sign, this is put before a string that want its value to be taken literally, i.e. turning off backslash to specify special characters.

Another more complex example is:

```
^([\w]+\.[\w]+)$
```

The sections in the square brackets specifies the character set, so in this case it's saying any word with (A-Z, a-z, 0-9 and '_') is valid, then it goes onto "\." which checks for a fullstop, and the [\w]+ is repeated again after the stop, so this regex looks for a pattern of a word separated by a full stop. Add this expression to the program used in the previous example and try it out.

This can now be adapted to this complex regex used for identifying email addresses, each section will be explained:

```
^((([\w]+\.[\w]+)|([\w]+))\@([\w]+\.)+([A-Za-z]{1,2}\.([A-Za-z]{1,2})|([A-Za-z]{1,3}))$
```

The regex will now identify email addresses.

- (([\w]+\.[\w]+)|[\w]+))
  - Is equal to any number of standard characters split between with a '.' followed by any number of standard characters
  - OR (|)
  - Any number of standard characters

- \@([\w]+\.)+
  - \@ will match just an "@" sign
  - Followed by any number of characters
  - Followed by a full stop
  - + means any number of the items in brackets

- ([A-Za-z]{1,2}\.([A-Za-z]{1,2})|([A-Za-z]{1,3}))
  - [A-Za-z] specifies that a word made up of just upper and lower-case letters, the "{1,2}" specifies the length of the word, a minimum of 1 and a max of 2 characters
  - \. separated by a '.'
  - Another "[A-Za-z]{1,2}"
  - OR (|)
  - "[A-Za-z]{1,3}" – Same as "[A-Za-z]{1,2}" but with a larger maximum length

Character Escapes

In regular expression, the backslash "\" signify that the character or group of character following it are special or to be taken literally:

Character	Description	Unicode Number (hex)
\t	Will match a tab character	0009
\r	Will match a carriage return (Note this is not the same as a newline (\n)	000D
\f	Matches a form feed (Better known as a page break	000C
\n	Matches a new line	000A
\e	Matches an escape	001B
\nnn	Use octal representation to specify a character (nnn is replaced by values)	Each letter has its own octal
\x nn	Uses a hexadecimal representation to specify a character (nn once again consists of two digits)	Each letter has it's own hexadecimal representation
\unnnn	Uses hexadecimal representation to display a Unicode character, each 'n' is a numeric value	Each Unicode letter has it's own hexadecimal representation
\W	Matches any non-character word	
\s	Matches any whitespace	
\S	Matches any non-whitespace	
\d	Matches any decimal	

	value	
\D	Any character but a decimal value	

Pre-processor directives

Pre-processors directives are designed as messages to the complier, they are processed before any compilation. They are signified by a hash symbol (#) and because of them not being statements do not require and end of line statement (;).

C# pre-processor directives offer nothing different to the ones in C++ or C, and they cannot be used to macros.

Directive	Explination
#define	It creates a symbol
#undef	Used to undefine a symbol
#if	It is used to test a symbol against a condition
#else	Used alongside #if
#elif	And else and an if in the same statement
#endif	Used to signify a close to an #if statement
#line	Modifies the compliers line number and an optional feature is to change file name outputs for files and warnings
#region	Useful for formatting
#endregion	Shows the end of an region

Above is a decently fleshed out table that includes all of the main directives. Below are examples of some useful directives:

<u>#define & #undef & if</u>

The example below shows if #define statement can be used with an #if statement:

```
#define VALUE
using System;

namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
#if (VALUE)
          Console.WriteLine("Defined");
#else
              Console.WriteLine("Not Defined");
#endif
          Console.ReadKey();
         }
    }
}
```

Output
>Defined

As you can tell this can be used to easily toggle code, the most common example is with use with debug code, or a verbose mode:

```
#define VERBOSE
using System;

namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
          Console.WriteLine("Start!");

          int a = 1 + 3;
#if (VERBOSE)
          Console.WriteLine("Addition complete");
#endif
          int b = 1 - 4;
#if (VERBOSE)
          Console.WriteLine("Subtraction complete");
#endif

          int c = 1 * 3;
#if (VERBOSE)
          Console.WriteLine("Multiplication complete");
#endif

          int d = 1 / 3;
#if (VERBOSE)
          Console.WriteLine("Division complete");
#endif

          Console.WriteLine("End!");
          Console.ReadKey();
         }
    }
}
```

The example below shows a very basic almost pointless program but it's supposed to demonstrate how #defined can be like toggles, copy over and run the program and compare what the program is like when you add or remove the #define at the top.

#if uses the normal conditionals as an actual if statement, the list is below:

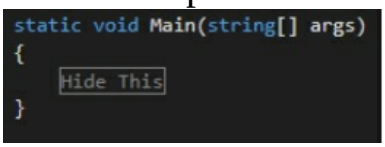
- ==
- || (OR)
- &&
- !=

#region
region offers no purpose to a program when executing but makes it easier to collapse and hide an area of code, it's used like this:

```
#region Hide This
Console.WriteLine("Print!");
#endregion
```

Output
>Print!

When collapsed it will look like this



Exception Handling

An exception is 'thrown' when an error occurs, it is a response a situation the program cannot continue from. Exceptions can be handled though and your program doesn't have to always explode, there're 4 keywords; **try, catch, finality** and **throw:**

- try – This is used to surround the code you would like to monitor for an exception, if an exception is found it transfers control to the catch section

- catch – This is where the program will end up if an exception is thrown within an accompanying try section.

- finally – This can accompany a try/catch but will run always regardless of an exception being thrown, a good example is

when opening a file to read, it always needs closing

- throw – This is used to manually throw an exception with your own custom text.

Below is an example of using a try/catch:

```
using System;

namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
         {
          try
              {
              //Diving by zero error!
              int zero = 0;
              int i = 1 / zero;
              }
          catch (Exception e)
              {
              Console.WriteLine(e.Message);
              }
          Console.ReadKey();
         }
    }
}
```

Output
> Attempted to divide by zero.

Note: The highlighted are signifies the parameter for the exception, the exception text can be obtained like below with the ".Message" property. But as you can see the program has printed out the error message, but not thrown a compilation error, the code within the try has not been completed, but has switched to the catch area.

Here is another example but this time a try/catch/finally is being used, this program opens a file (assume it exists) and the finally statement is there to 100% make sure the file is closed even if an exception is thrown.

```
using System;
using System.IO;
```

```
namespace ConsoleApplication1
{
   class Program
    {
      static void Main(string[] args)
        {

         StreamReader sr = null;
         try
            {
            sr = new StreamReader("file.txt");

            //Used to simulate an example
            throw new Exception("Error reading file!");
            }
         catch (Exception e)
            {
            Console.WriteLine(e.Message);
            }
         finally
            {
            //Closes the file
            if (sr != null)
                {
                    sr.Close();
                }
            }
         Console.ReadKey();
        }
    }
}
```

Note the use of the “throw new Exception(….” this is used to throw your own custom exceptions, this can be used in situations where you need to be informed of a critical error in your program.

End of chapter Quiz

- Which key characteristics does an operator overloading function need?
- What is the role of a regular expression?
- What is the namespace used for regular expresions?
- What does putting '@' before a string achieve?
- How does C#'s directives differ from C or C++?
- When will a finally statement run?
- What is the keyword 'throw' used for?
- What is the role of #region and #endregion?
- In regular expressions what does "^" signify?
- In regular expressions, what would this mean [abfe]+

Answers

- Public and static.
- Used to match patterns to strings.
- System.Text.RegularExpressions.
- It tells the complier to take the string literally and ignore backslash identifiers.
- There is no support for macros.
- Always, even if an exception has been thrown.
- It is used to create your own custom exceptions.
- They're used for formatting and improved readability.
- Signifies the start of the word.
- Any number of characters in a row that are a,b,f and e would be a match

Chapter 5

Program formatting

Attribute

An attribute is a tag used to convey information to the compiler about behaviours of elements of the program, an attribute is specified by a pair of square brackets ([]) and are placed above the element they are providing information about.

The .NET framework provides two types of attribute:

11. Predefined
12. Custom built

Predefined

There are three pre-defined attribute types:

- AttributeUsage
- Conditional
- Obsolete

Attribute Usage

Attribute usage describes how a custom attribute can be used and where it can be applied, the syntax is below:

```
[AttributeUsage(validOn, AllowMultiple, Inherited)]
```

Where:

- **validOn** specifies where the attribute can be set, it uses the enum "AttributeTargets" that can consist of:
    - AttributeTargets.Class
    - AttributeTargets.Constructor
    - AttributeTargets.Field
    - AttributeTargets.Method
    - AttributeTargets.Property,
- **AllowMultiple** specifies whether more than one version of the

attribute can be created and applied.

- **Inherited** specifies whether the attribute is inherited when other classes inherit the attribute the class is connected to

This is used later to make custom attributes.

Conditional

These conditional attributes are used to mark a method that depends on a certain pre-processor directive, this this example we will use a debugging example:

```
using System;
using System.Diagnostics;

namespace ConsoleApplication1
{
   class Program
    {
      public static void Main()
        {
            Debugging();
         Console.ReadKey();
        }

      [Conditional("DEBUG")]
      static void Debugging()
        {
         Console.WriteLine("This is example debugging code:");
         Console.WriteLine("Number of loops: " + 1);
         Console.WriteLine("Values found: " + 102);
        }
    }
}
```

This will only print out the example output if the program is set in debug mode, this gives a very nice clean way of abstracting code and unnecessary output. The DEBUG flag is automatically built into Visual Studio, so when the program is switched to release mode the debug mode will be false and the code will not run.

This can also be done with custom pre-processor directives like so:

```
#define CUSTOM
using System;
using System.Diagnostics;

namespace ConsoleApplication1
{
   class Program
    {
      public static void Main()
         {
             Custom_Method();
          Console.ReadKey();
         }

      [Conditional("CUSTOM")]
      static void Custom_Method()
         {
          Console.WriteLine("Custom output!");
         }
    }
}
```

Obsolete

This is used for legacy and compatibility reasons, if for example you has a class that used a function SaveFile() but was later updated and this function became obsolete a new function was created called Save() because now this function saved other items, SaveFile() would be marked as obsolete so other programs that use this functions would not have problems with compatibility.

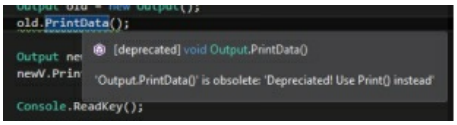
The example shows obsolete being used:

```
 using System;

namespace ConsoleApplication1
{
   class Program
    {
      public static void Main()
         {
```

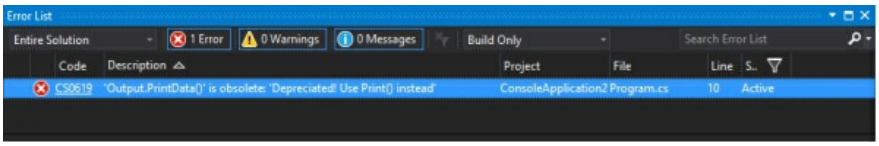
```
            Output old = new Output();
                old.PrintData();

            Output newV = new Output();
                newV.Print();

            Console.ReadKey();
           }
     }

    class Output
     {
       string example = "Example";

       [Obsolete("Depreciated! Use Print() instead", false)]
       public void PrintData()
          {
           Console.WriteLine(example);
          }

       public void Print()
          {
           Console.WriteLine(example);
          }
     }
}
```

Output
>Example
>Example

As you can see both method work, however the old.PrintData() will show a warning, it looks like this:



Note: This is because the highlighted ‘false’ parameter was used, this means just a warning is presented, if this is true an error is created and won’t allow the program to compile. This will appear in the error list like this:



Custom Attributes

The framework allows custom created attributes that can store any information the programmer deems relevant information, this is very similar to defining a class, however the class needs to inherit the System.Attribute class, with the AttributeUsage defined above. A basic example is below, the example is definition an attribute that just creates a description:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited = true)]
class Description : Attribute
{
        public string Text { get; }

        public Description(string text)
          {
              Text = text;
          }
}
```

As you can see this is very similar to creating a class, apart from the inheritance off Attribute and the AttributeUsage attribute above. This can be used like so:

```
[Description("Main method!")]
[Description("AllowMultiple == true, so you can have multiple Descriptions")]
class Program
{
        public static void Main()
          {

          }
}
```

This information can be obtained and used by the process of ‘reflection’, this will be explored in the next section.

Reflection

Reflection of objects is used for obtaining information at runtime, the System.Reflection gives access to these features.

Reflection can be used to:

- View information about attributes
- It allows late binding to methods and properties
- It allows creation of types at runtime

Metadata

Metadata is data about data, and in this case, it is data about attributes. To start you need to create an instance of Member info like so:

```
MemberInfo info = typeof(AttributeTest);
```

This can now be used to check information. The example below prints off the attributes attached.

Remember to include this at the top

```
using System.Reflection;
```

For the example, we are going to use the Description Attribute from earlier:

```
using System;
using System.Reflection;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main()
         {
          AttributeTest t = new AttributeTest();
              t.PrintAttributes();

          Console.ReadKey();
         }
    }

   [Description("Description 1!")]
   [Description("Description 2!")]
   class AttributeTest
    {
      MemberInfo info = typeof(AttributeTest);

      public void PrintAttributes()
         {
          //Returns an array of attributes
          object[] attributes = info.GetCustomAttributes(true);

          //Prints attributes
          foreach (Description attr in attributes)
              {
              Console.WriteLine(attr.Text);
              }
         }

    }

   [AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited
= true)]
   class Description : Attribute
```

```
    {
      public string Text { get; }

      public Description(string text)
        {
            Text = text;
        }
    }
}
```

Output
>Description 1!
>Description 2!

As you can see the program has printed out the description text from the attributes attached to it. The line:

```
…info.GetCustomAttributes(true);
```

Returns the list of attributes and the attributes are then printed in the following foreach loop.

Indexers

Indexers allow custom classes to be indexed much like you would with an array, each member is accessed with the array access operator ([])

The indexer is defined like so:

```
type this[int index]
{
        get
            {
            //returns values
            }
        set
            {
            //sets values
            }
}
```

Where you would call it like this:

```
Example[2];
```

With the exact syntax on an array.

So, for this example it will be a store inventory searcher, the indexer will take a name and search the catalogue:

```
using System;
using System.Collections.Generic;

namespace ConsoleApplication1
{
   class Program
    {
      static void Main()
         {
          Inventory main_floor = new Inventory();

          //Adds Items
          main_floor.AddItem(new Item("PS4", 499.50));
          main_floor.AddItem(new Item("RX_480", 200.50));

          //Uses indexer to grab price
          Console.WriteLine("Price: " + main_floor["RX_480"]);
          Console.ReadKey();
         }
    }

   class Inventory
    {
      List<Item> Items = new List<Item>();

      public void AddItem(Item item)
         {
             Items.Add(item);
         }

      public double this[string index]
         {
          get
              {
             foreach (Item item in Items)
                  {
                if (item.Name == index)
```

```
                    {
                return item.Price;
                    }
                }

            //Used to signify that there is not item
            return -1;
            }
        set
            {
            //You can also use this to set a value much like setting a value in
            //an attribute
            }

        }
    }

   //Simple class to hold item
   class Item
    {
      public Item(string name, double price)
        {
            Name = name;
            Price = price;
        }

      public string Name { get; }
      public double Price { get; }
    }
}
```

Both the highlighted areas signify where the indexer is defined and used. The program just keeps track of items and the indexer is used to take a string to find the price.

Exercise

Add an overloaded version of the indexer (This can be done) that takes the price as the index and returns the name. For simplicity purposes don't worry about two items having the same price, just assume all items have different prices

Solution

Your Inventory class should look like this: (The new indexer is highlighted)

```
class Inventory
{
      List<Item> Items = new List<Item>();

      public void AddItem(Item item)
        {
            Items.Add(item);
        }

      public double this[string index]
        {
         get
            {
            foreach (Item item in Items)
                {
               if (item.Name == index)
                    {
                  return item.Price;
                    }
                }

            return -1;
            }
        }
      public string this[double index]
        {
         get
            {
            foreach (Item item in Items)
                {
```

```
                if (item.Price == index)
                    {
                  return item.Name;
                    }
                 }

            return null;
             }
        }
}
```

Collections

Collection classes are specialized built in classes used for data storage and retrieval. Some of these classes have members that are defined as ‘object’, object is the base class for all data types in C#. This was the method of storing any data type before the concept of generic classes was implemented.

Below is a list and explanation of the main collections:

List<T>

List is the upgrade more modern version of ArrayList, it is a dynamically sized list that is used to store different values of the same type, it can be extended and reduced (where it resizes automatically). The List class is a generic class (Learned about in the next chapter) but what it means in this case is it can hold anything, even other lists (This is how you can make a 2D list).

A list is defined like so:

```
List<type> list = new List<type>();
```

Where for example a list of integers is defined like so:

```
List<int> list = new List<int>();
```

Values can be added to the list by using the .Add() function and can be removed by using .Remove()

You can obtain the length of the List<> or how many items it houses by accessing the .Count property.

Hashtable

Note: This is depreciated and should not be used, you should use a Dictionary<>, but it's important you know how this collection works.

A hashtable is table (as the name suggests) that stores key-value pairs, these pairs are determined by a hash function. A hash function takes an input of any length and returns a value of fixed length, this process can also no be undone. So in the hashtables case, a keypair that is to be added is hashed, now this hash will be equal to a position in the table. The key pair will then be stored in that location.

Note: the key specified must be unique

One of the biggest positivises of a hashtable is how quick it is to access or check if the table contains and item, this is because in most lineal structures like an array or list you need to incrementally search through a list to exhaustively check if there is an item, but in this case of a hashtable if a search target hashed to find where it would go and there is no value in that location, it's not there.

Dictionary<TKey, TValue>

Must like this List<T>, the Dictionary is a generic data type that is based off the Hashtable mentioned previously is uses a key value pair to save data.

A dictionary is defined like so:

```
Dictionary<KeyType, ValueType> dictionary = new Dictionary<KeyType, ValueType>();
```

Where "KeyType" and "ValueType" both define the data type that will be used to store each member of the pair.

Adding to the dictionary can be done like so:

```
dictionary.Add(1, "Value");
```

And you can obtain a value by searching for a key:

```
Console.WriteLine(dictionary[key]);
```

And the values can be incremented through using the foreach value, the type

that hold both values are called a KeyValuePair<TKey, TValue>. It's done like so:

```
foreach (KeyValuePair<int,string> item in dictionary)
{
        Console.WriteLine(item.Key);
        Console.WriteLine(item.Value);
}
```

Stack

This is known a First-in-Last-out (FILO) data structure where when a value Is placed in its 'pushed' and when a value is taken (Can only be from the top of the stack) it's popped.

A stack Is defined like so:

```
Stack s = new Stack();
```

Values are added buy pushing:

```
s.Push(2);
```

And taken of by popping. Note the use of the integer cast, this is because items are stored as objects.

```
int value = (int)s.Pop();
```

Queue

A Queue is very similar to a stack but instead is a First-in First-out (FIFO) Data structure. Items are added and taken away by using 'Enqueue()' and 'Dequeue()' respectively.

A queue is defined like so:

```
Queue q = new Queue();
```

And adding a removing from a queue is done like so:

```
//Add to the queue
q.Enqueue(1);

//Take from the queue
int i = (int)q.Dequeue();
```

Generics

Generics is a huge feature of C# that allows the programmer to create a single definition of a class that takes any data type, even custom-made classes. The usage of the generic type is denoted by “T”, this stems from its origins of originally being called “Templating” in C++. For example the List<> collection we have come across previously is a generic class,

You can create a generic class like so:

```
class Generic<T>
{

}
```

Notice the uses of <>, these are used to create generic data type, these also do not need to be define as ‘T’ and can be multiple definitions:

```
class Generic<Example, AnotherExample>
{

}
```

The above example is valid, but it’s advised for readability to start the definition with a ‘T’. Below is an example how to use the generic data types:

```
class Generic<TEx>
{
        public TEx value;

        public Generic(TEx _value)
          {
              value = _value;
          }

        public void Print()
          {
           Console.WriteLine($"Data is: {value}", value);
          }
}
```

This class definition can be used with any defined data type:

```
Generic<string> gStr = new Generic<string>("Hello");
gStr.Print();

Generic<int> gInt = new Generic<int>(2);
gInt.Print();
```

As you can tell from the code snippet above you define the type to use by definition it in the “<>”. In the example below it will show it working with a user defined type:

```
Generic<myClass> gMyClass = new Generic<myClass>(new myClass());
gMyClass.Print();
```

With an output of

```
> Data is: ConsoleApplication1.myClass
```

As you can tell generics allow a very dynamic way of definition a class that allows almost infinite amount of data types. Below is an actual functionally program that is used to swap two data types:

```
using System;
namespace ConsoleApplication1
{
   class Program
```

```
    {
      static void Main()
        {
         int a = 10;
         int b = 42;

         Console.WriteLine($"a is before swap: {a}", a);
         Console.WriteLine($"b is before swap: {b}", b);

         Swap<int> swap = new Swap<int>(ref a, ref b);

         Console.WriteLine($"a is after swap: {a}", a);
         Console.WriteLine($"b is after swap: {b}", b);

         Console.ReadKey();
        }
    }
   public class Swap<T>
    {
      public Swap(ref T Left, ref T Right)
        {
         T temp;
            temp = Left;
            Left = Right;
            Right = temp;
        }
    }
}
```

Another example could be to create a generic array, note the use of indexers to make accessing and changing values easier:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      static void Main()
         {
          Array<int> intArray = new Array<int>(1);
              intArray[0] = 1;

          Array<string> strArray = new Array<string>(1);
          strArray[0] = "Hello";

          Console.WriteLine(intArray[0]);
          Console.WriteLine(strArray[0]);
          Console.ReadKey();
         }
    }
   public class Array<T>
    {
      T[] array;
      public Array(int size)
         {
          array = new T[size];
         }

      //Indexer used to access and change values
      public T this[int index]
         {
          get
              {
             return array[index];
              }
          set
              {
             array[index] = value;
```

```
            }
        }

    }
}
```

It's cheating a little bit to have an array internally defined but you get the picture. The array is defined as an array of generics, this allows for a type to be defined later on.

End of chapter Quiz

1. What are the two types of attributes that the .NET Framework supports?
2. Where would you normally find the Obsolete built in attribute?
3. When creating a custom attribute, which class does it need to inherit?
4. How would you define 'metadata'?
5. What is the role of reflection in code?
6. What method can be used to get attribute info on a function?
7. What's the biggest reason to use an indexer?
8. When referring to a stack what does FIFO?
9. What brackets signify a generic class?
10. What variable is used to save the key and value of a dictionary?

Answers

1. Predefined and custom built.
2. On old deprecated method that can't be removed due to compatibility reasons but needs to be phased out.
3. System.Attribute.
4. Data about data.
5. View the underlying data of function or property.
6. GetCustomAttributes()
7. Improve syntax when accessing or changing values in a class.
8. First-in First-out.
9. < and >, inside the brackets the generic types are defined.
10. KeyValuePair<TKey, TValue>

Chapter 6

Processes and functionality

Delegates

Delegates are very similar to function pointers in C or C++ if you have come across this concept before. So, delegates are a reference type variable that holds references to a function.

A real-world use for delegates is using them for dynamic calls back to a function, the delegate is passed as a parameter and the function is dynamically called.

So for example the definition below can be used to reference any method that has two int parameters, it returns an integer and is marked as private.

```
public delegate int ExampleDeligate(int a, int b);
```

You then would create an instance of this delegate by using the 'new' keyword like so:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      public delegate int ExampleDeligate(int a, int b);
      static void Main()
         {
          ExampleDeligate addDel = new
ExampleDeligate(AddTwoNumbers);
         }

      public static int AddTwoNumbers(int a, int b)
         {
          return a + b;
         }
    }
}
```

As you can see the delegate is initialised and in the brackets is the name of function it will reference.

To use the delegate just treat it as it like any other function:

```
addDel(1, 2);
```

This gives a nice dynamic, useable way to pass functions as parameters. Below is an example using this to act as a calculator;

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      public delegate float Calculation(float a, float b);
      static void Main()
        {
         //All you have to do is pass in the function by name
             Calculater(10, 20, Add);
             Calculater(5, 267, Sub);
             Calculater(14, 2, Mul);
             Calculater(10, 10, Div);

         Console.ReadKey();
        }

      public static void Calculater(float a, float b, Calculation cal)
        {
         //Dynamic method call
         float value = cal(a, b);
         Console.WriteLine($"New value is: {value}", value);
        }


      //Basic functions
      public static float Add(float a, float b)
        {
         return a + b;
        }
```

```
        public static float Sub(float a, float b)
           {
            return a - b;
           }
        public static float Mul(float a, float b)
           {
            return a * b;
           }
        public static float Div(float a, float b)
           {
            return a / b;
           }
    }
}
```

Output
>New value is: 30
>New value is: -262
>New value is: 28
>New value is: 1

The highlighted area shows how you can pass in a delegate as a parameter, and all you do is pass in the function in the delegates parameter position, this gives a wonderfully easy dynamic way to reuse code.

Multicasting

Delegates also allow the concatenation of function calls onto each other, like this:

```
Calculation DEL = new Calculation(Add);
DEL += new Calculation(Sub);
DEL += new Calculation(Mul);
DEL += new Calculation(Div);
```

This means all the method will be called in one long chain, so for example say the parameters are 10 and 20 the answer would be equal to the last method added, in this case (10/20 = 0.5), in this example it runs over all the method but they don't affect each other so it doesn't change the answer, so careful consideration Is needed when choosing what method to add to a multicast delegate.

But the new example below it should multicasting when not returning values:

```
using System;
namespace ConsoleApplication1
{
   class Program
    {
      public delegate void Print();
      static void Main()
         {
          Print call = new Print(PrintHello);
          call += new Print(PrintToday);
          call += new Print(PrintCar);
```

```
            //Method call
                call();

            Console.ReadKey();
            }

        public static void PrintHello()
            {
            Console.WriteLine("Hello");
            }
        public static void PrintToday()
            {
            Console.WriteLine("Today");
            }
        public static void PrintCar()
            {
            Console.WriteLine("Car");
            }
    }
}
```

This shows you can string multiple function calls into one delegate call, this can also be passed as a parameter and used in another function.

Anonymous Methods

Previously we mentioned previously delegates are used to reference any methods with the same signature as the delegate.

Anonymous methods provide a technique to pass a code block as a delegate parameter. They're methods without a name, so there're just the body. You do not need to define the return type of the method, it's inferred (automatically determined) from the return statement in the methods body.

They are effectively delegates that aren't referencing an actual method, it's being created just for the delegate, so going back to our calculator example it would look like this:

```
using System;
namespace ConsoleApplication1
{
   //Delegate
   public delegate int Calculation(int a, int b);

   class Program
    {
      Calculation add = delegate (int a, int b)
        {
         return a + b;
        };
      Calculation sub = delegate (int a, int b)
        {
         return a - b;
        };
      Calculation mul = delegate (int a, int b)
        {
         return a * b;
        };
      Calculation div = delegate (int a, int b)
        {
         return a / b;
        };

      static void Main()
        {
         Program p = new Program();

            p.add(10, 20);

            p.div(2, 10);

         Console.WriteLine();
        }
```

```
    }
}
```

Where each command is defined as an anon method, and just called like a regular function.

The use of anonymous methods reduces the coding overhead of creating delegates because another method does not need defining prior to creating a delegate instance.
method seems wasteful.
So anonymous methods should be used when the overhead of creating a

Events

Events are user actions such as key pressed, clicks, mouse movements etc. This allows programs or code to execute that is dependent on an event triggering, this is known as event-driven programming.

Delegates are often used with events, events are associated with the event handler (trigger detection) by using the delegate. Events use the publisher-subscriber model:

- A publisher is a class that contains the event, this is as the name suggests used to publish the event.
- The subscriber class is a class that accepts the event. The delegate in the publisher class invokes the method of the subscriber class.

Declaring Event

To declare an event internally in a class, you first need a delegate type defined and attach it to the handler. The code below demonstrates this:

```
using System;
namespace ConsoleApplication1
{
   //Delegate
   public delegate string DelExample(string str);

   class EventExample
    {
      //Event decleration
      event DelExample MyEvent;

      public EventExample()
         {
          MyEvent += new DelExample(PrintWelcome);
         }

      public string PrintWelcome(string username)
         {
          return username + " Welcome!";
         }

      static void Main()
         {
          EventExample ex = new EventExample();

          //Calls the event handler
          Console.WriteLine(ex.MyEvent("User1"));
          Console.ReadKey();
         }
    }
}
```

Multithreading

A thread is a single process of execution, multithreading is the ability to run lots of small processes concurrently (At the same time), this stops programs from wasting CPU cycles waiting and can make programs much more efficient.

However, introducing multithreading into problems that are trivial or cannot be run concurrently can add an unnecessary overhead and can reduce the efficiency of a program.

So far we have written programs that execute in a single thread and the computer executes them just from start to finish in one straight line.

Threads

The tools required to create and work with threads resides in the System.Threading class. One of the simplest functions is being able to grab the current thread, this is done like so:

```
Thread t = Thread.CurrentThread;
```

Creating Threads

You can create a thread like so:

```
Thread s = new Thread(DoLotsOfWork);
```

With the function the thread will executing being contained in the brackets. The example below shows how this concurrent behaviour can work:

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
   class Program
    {
      public static void DoLotsOfWork()
        {
         for (int i = 0; i < 5; i++)
             {
             Thread t = Thread.CurrentThread;
             Console.WriteLine($"{t.Name}: i is equal to: {i}");
```

```
            }
        }

    static void Main()
        {
         for (int i = 0; i < 5; i++)
            {
            //Creates a thread
            Thread s = new Thread(DoLotsOfWork);

            //Assigns the thread a name
            s.Name = $"Thread {i}";

                s.Start();
            }

         Console.ReadKey();
        }
    }
}
```

Output

```
>Thread 2: i is equal to: 0
>Thread 2: i is equal to: 1
>Thread 2: i is equal to: 2
>Thread 2: i is equal to: 3
>Thread 2: i is equal to: 4
>Thread 0: i is equal to: 0
>Thread 1: i is equal to: 0
>Thread 0: i is equal to: 1
>Thread 1: i is equal to: 1
>Thread 1: i is equal to: 2
>Thread 1: i is equal to: 3
>Thread 1: i is equal to: 4
>Thread 0: i is equal to: 2
>Thread 0: i is equal to: 3
>Thread 0: i is equal to: 4
>Thread 3: i is equal to: 0
>Thread 3: i is equal to: 1
```

```
>Thread 3: i is equal to: 2
>Thread 3: i is equal to: 3
>Thread 3: i is equal to: 4
>Thread 4: i is equal to: 0
>Thread 4: i is equal to: 1
>Thread 4: i is equal to: 2
>Thread 4: i is equal to: 3
>Thread 4: i is equal to: 4
```

Note: this may come through in a different order for you.

This example above creates 5 threads that prints out 5 iterations of a for loop, as you can see it doesn't all come through in order, this is because it's occurring concurrently and each of these threads are working independently.

This lack of order can be a problem in some circumstances, this is where the "lock" keyword comes into its own, it's used to surround code that is deemed crucial and code that cannot have multiple threads using it at the same time, we'll adapt the example from before to include the lock:

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
   class Program
    {
      static object padLock = new object();
      public static void DoLotsOfWork()
        {
         lock (padLock)
         {
            for (int i = 0; i < 5; i++)
                 {

               Thread t = Thread.CurrentThread;
               Console.WriteLine($"{t.Name}: i is equal to: {i}");
                 }
         }
        }
```

```
        static void Main()
          {
           for (int i = 0; i < 5; i++)
               {
               //Creates a thread
               Thread s = new Thread(DoLotsOfWork);

               //Assigns the thread a name
               s.Name = $"Thread {i}";

                    s.Start();
               }

           Console.ReadKey();
          }
    }
}
```

Output

```
>Thread 0: i is equal to: 0
>Thread 0: i is equal to: 1
>Thread 0: i is equal to: 2
>Thread 0: i is equal to: 3
>Thread 0: i is equal to: 4
>Thread 2: i is equal to: 0
>Thread 2: i is equal to: 1
>Thread 2: i is equal to: 2
>Thread 2: i is equal to: 3
>Thread 2: i is equal to: 4
>Thread 1: i is equal to: 0
>Thread 1: i is equal to: 1
>Thread 1: i is equal to: 2
>Thread 1: i is equal to: 3
>Thread 1: i is equal to: 4
>Thread 3: i is equal to: 0
>Thread 3: i is equal to: 1
>Thread 3: i is equal to: 2
>Thread 3: i is equal to: 3
>Thread 3: i is equal to: 4
```

```
>Thread 4: i is equal to: 0
>Thread 4: i is equal to: 1
>Thread 4: i is equal to: 2
>Thread 4: i is equal to: 3
>Thread 4: i is equal to: 4
```

As you can see each iteration loop is coming through in order, i.e. from 0 – 4, may also be able to see that the threads are not necessarily in order, the lock works on a first come first serve basic and once a thread touches it, it becomes locked until the object is released. Please also not the use of an object alongside the lock, this is used as a flag for if the lock is available or locked.

This lock system is very useful when it comes to dealing with files, if two threads were to try and read or write to a file at the same time there will be errors or lines would not appear in order. Below is an example demoing this:

```
using System;
using System.IO;
using System.Threading;

namespace ConsoleApplication1
{
   class Program
    {
      public static void WriteToFile()
         {
          //Error here!
          StreamWriter sw = new StreamWriter("file.txt");
          for (int i = 0; i < 5; i++)
              {
             sw.WriteLine($"Test {i}");
              }
              sw.Close();
         }

      static void Main()
         {
          for (int i = 0; i < 5; i++)
              {
             //Creates a thread
             Thread s = new Thread(WriteToFile);

             //Assigns the thread a name
             s.Name = $"Thread {i}";

                  s.Start();
              }

          Console.ReadKey();
         }
    }
}
```

This will crash and throw a **System.IO.IOException** as two or more threads will try and open the file to write to it at the same time. This is a real-world

usage for the lock structure as adding a lock will solve this, below is the lock being introduced, this will solve the issue:

```
using System;
using System.IO;
using System.Threading;

namespace ConsoleApplication1
{
   class Program
    {
      static object padLock = new object();
      public static void WriteToFile()
        {
         lock (padLock)
             {
             StreamWriter sw = new StreamWriter("file.txt");
             for (int i = 0; i < 5; i++)
                  {
                sw.WriteLine($"Test {i}");
                  }
                  sw.Close();
             Console.WriteLine("File written to!");
             }
        }

      static void Main()
        {
         for (int i = 0; i < 5; i++)
             {
             //Creates a thread
             Thread s = new Thread(WriteToFile);

             //Assigns the thread a name
             s.Name = $"Thread {i}";

                  s.Start();
             }
```

```
            Console.ReadKey();
        }
    }
}
```

This code will now not crash, but it really cannot be considered multithreaded, it used multiple threads but effectively acts as a single threaded application, the lock statement should only be used when it's entirely necessary. Copy this code over and have a play with it and get familiar with how this code works.

LINQ

LINQ stands for Language Integrated Query and solves the problems with querying data and allows a tight readable compact syntax for returning queries from data.

We'll only be covering the LINQ queries for internal objects, but LINQ can be used to query data represented in many different ways. The diverse ways are:

- LINQ to objects
- LINQ to XML
- LINQ to datasets
- LINQ to SQL (DLINQ)
- LINQ to entities

There are two types of syntax for LINQ:

- Lambda (Method) Syntax:

```
var longWords = words.Where(w => w.Length > 10);
```

- Query (Compression) Syntax

```
var longWords = from w in words where w.Length > 10 select w;
```

We'll only go into the query syntax because I think it's the easiest to use but feel free to take a look at the other query for extra research.

Query Syntax

The query syntax is just a representation comparable to that of SQL, with a SELECT, FROM and WHERE represented in slightly different representation. To use LINQ queries, you need to include System.linq. Below is an example of using LINQ to search through some strings for values with less than 5 letters:

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
   class Program
    {
      static void Main()
         {
         string[] words = {"test","verylongword", "sw", "one"};

          var shortWords = from word in words
                           where word.Length < 5
                           select word;

          foreach (var word in shortWords)
              {
              Console.WriteLine(word);
              }
          Console.ReadKey();
         }
    }
}
```

Output

```
>test
>sw
>one
```

The LINQ statement is highlighted and will be broken down:

from word in words
This defines the range and where the range is from, this is effectively saying every word in the string array ‘words’

where word.Length < 5
Using the range value word, it grabs a property and compares it to a value, this is a conditional where if a value matches as ‘true’ will be added to the output.

select word;
If the previous conational was true, this is the value to take, so for example you could return the lengths of all words under 5 values by saying at the end of the query:

```
select word.Length;
```

Exercise

In the skeleton program below, add a LINQ statement to search through the array of values ‘numbers’ to search for even numbers. Then use a foreach loop to print the resulth

Code

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
   class Program
    {
      static void Main()
         {
         int[] numbers = {1, 4, 99, 34, 23, 12, 3, 2, 909, 1000, 13, 1, 2};

          //Add code goes here

          Console.ReadKey();

         }
    }
}
```

Solution

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
   class Program
    {
      static void Main()
         {
         int[] numbers = {1,4,99, 34, 23, 12, 3, 2, 909, 1000, 13, 1, 2};

          var evenNumbers = from value in numbers
                            where value % 2 == 0
```

```
                    select value;

            foreach (var value in evenNumbers)
                {
                Console.WriteLine(value);
                }
            Console.ReadKey();
            }
    }
}
```

The output should also be:

```
>4
>34
>12
>2
>1000
>2
```

So, remember LINQ offers a very nice tight controlled syntax for looking through data, it can also be used to look through many types of data including running SQL commands on a database to query data in tables that can be incredibly powerful for a programmer.

End of chapter Quiz

1. What do delegates allow a programmer to do?
2. What does multicasting allow you to do with delegates?
3. What is an anonymous method?
4. What is an event handler attached to?
5. What is a thread?
6. What is the structure called that prevents two threads from using the same block of code?
7. What does LINQ stand for?
8. What does LINQ allow you to do?
9. What are the two types of syntax for LINQ?
10. In what scenario would multithreading now help?

Answers

1. Save a reference to a function, this allows the programmer to pass function calls around as parameters.
2. You can squash many functions calls into a single delegate.
3. A function without the function definition and is just a delegate defined with a body.
4. A delegate that fires when the event is triggered.
5. A single independent process of execution.
6. Lock.
7. Language Integrated Query
8. Quickly query data for results, for example if you just wanted words with a length of 5 characters.
9. *Query and Lambda.*
10. *When the problem is trivial i.e. a simple problem.*

C++

Step-by-Step Guide to C++ Programming from Basics to Advanced

TABLE OF CONTENT

Chapter 1: Introduction and Installation

History of C++

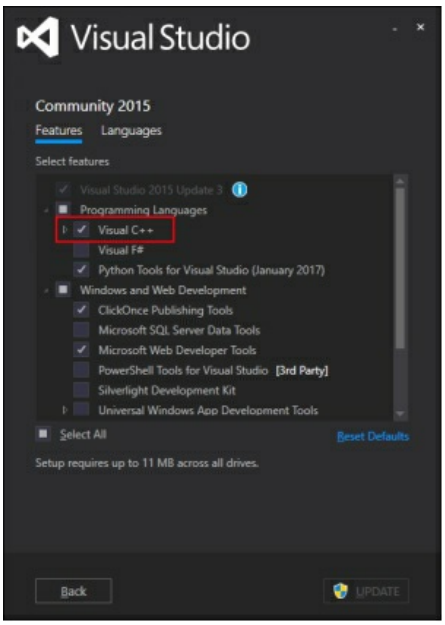
C++ is a general purpose programming language. It's aim as a programming language is to offer a platform to create efficient and bare-boned applications. C++ was released in 1983, and first started on in 1979 by Bjarne Stroustrup where it was known as "C with classes" where it was later in 1983 renamed to be C++.

The language is still maintained with updates every few years, where it is still one of the most popular languages in the world.

Running C++ on Windows

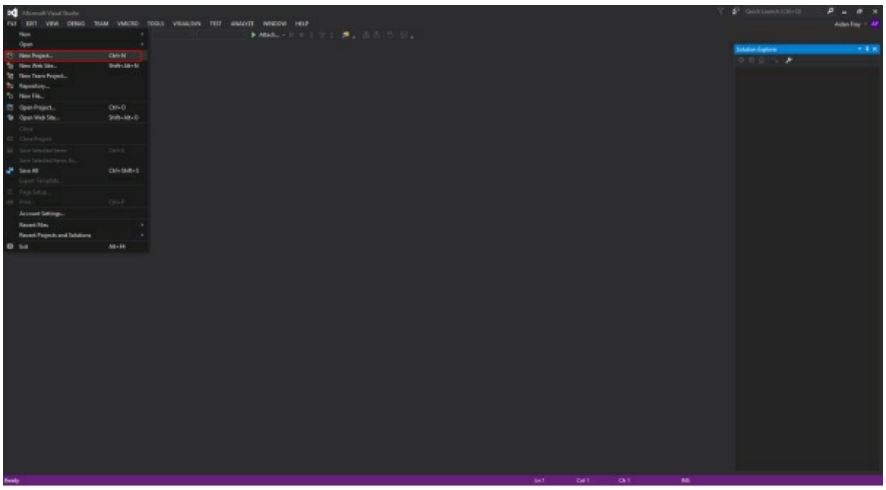
One of the best ways to run C++ on a Windows based computer is Visual Studio, this can be downloaded directly from Microsoft with the most recent version being Visual Studio 2017.

Download and run the installer, throughout the installer you'll be met with this menu, make sure C++ is selected.

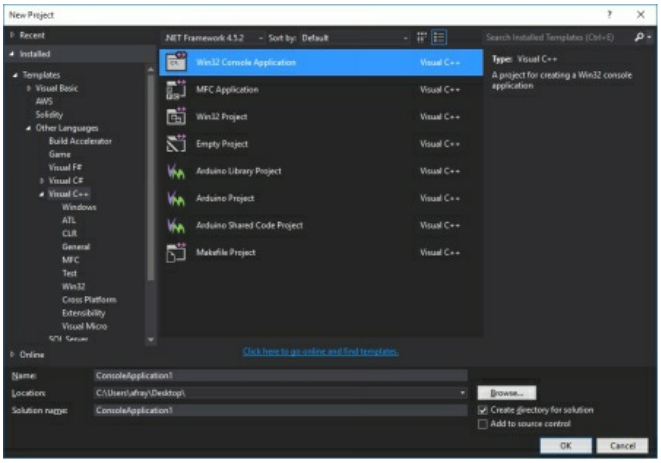


Complete the install and load up Visual Studio.

To create a project you need to then select FILE -> NEW PROJECT like so:



You'll then be met with this window:



Follow the menus through and you'll be ready to run the code.

Running C++ on Mac and Linux

Running on C++ on UNIX based environments can be achieved by the use of clang or "gcc", you can use it like so in the OS's terminal window:

```
gcc main.cpp -o main.out
```

The file can now be run like:

```
./main.out
```

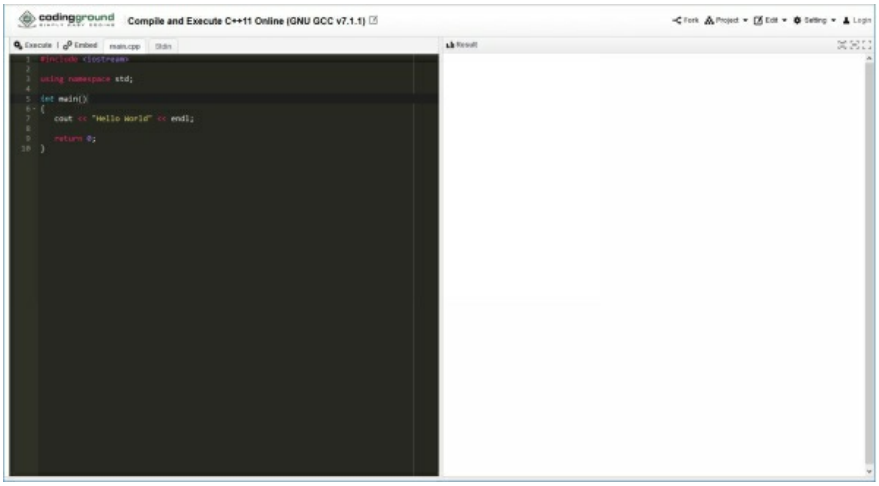
You can create a .cpp file in a notepad editor of your choice.

Running C++ Online

A easy way to compile C++ online is by using “TutorialPoints” C++ Online, like is below:

https://www.tutorialspoint.com/compile_cpp11_online.php

And should look something like this:



Prerequisite

A few things need to be explained before starting is basics:

The first is the basic program design:

```
#include "stdafx.h"

using namespace std;

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

The program above is the basic design provided, the code goes in the curly brackets of the main() section and each aspect will be described in the later

sections.

You'll also come across lines that start with "//", these are comments and their only purpose is to add descriptions to label and explain code, they are defined by a distinctive green colour:

```
//Comment
```

The next thing is output from a program, in examples in upcoming chapters you will come across lines like:

```
cout << "Print" << endl;
```

This for the time being is simply printing out data to the screen, the sections of it will be explained in later sections.

Chapter 2: Basics

Variables and types

The basis of a program is data, in the form of numbers and characters. The storing, manipulation and output of this data gives functionality. Variables hold this data, think of it as a box used to store a single item.

C++ is a heavily typed language where each variable must have a defined type. A type is an identifying keyword that defines what the variable can hold. The first type we will come across is the integer, this can hold real numbers (numbers without a decimal place), an integer is defined below:

```
int value = 6;
```

- **int** is the defined type keyword, we will learn about the difference possibilities later.

- "**value"** is the ID for the variable, this can be anything you want to call it, this is used to allow a variable to have a meaningful name, the variable could be defined as **"int pineapple = 3;"**, but it's good practise to make them relevant. However, there are a few exception to this as a variable cannot be a single digit i.e. '4' or cannot contain special characters (!"£$%^&*) etc.

- **"= 6;"** Is the assignment section, where the values of 3 is placed in the integer box for use later**. This also ends with a semi colon, this is used to signify the end of a line.**

This variable can now be used in valid areas of the program, like so:

```
int anotherValue = value;
```

The value of **"value"** defined earlier will now be placed in the **anotherValue** variable and they will now both have the value of 3. The value of **value**

doesn’t change as it is just being copied and placed into **anotherValue**.

String

The String is another crucial variable, this variable type is used to store a series of characters, an example could be the word “batman”, the word is composed of characters that are stored in the String variable. It’s an array of characters (More on arrays later), but effectively it’s the single characters of the word stored next to each other in memory. The ‘\n’ is a special character that stands for a newline.

```
#include "stdafx.h"
#include <string>

using namespace std;

int main()
{
	//String class
	string wordStr = "";

	//Character array
	char wordCharArray[] = "";

	return 0;
}
```

As you can see above there are two ways to save a string, the first way is by using a ‘wrapper’ that is something that hides functionality and adds extra on top, this makes it easier to move around a string (we’ll talk about passing around variables later) and the second way is to use a char array, string is the recommended way to store a string variables.

Note the use of:

```
#include <string>
```

This informs the program you want to use the “string” wrapper in the program.

Boolean

Booleans are used as expressions, their values can only be **true** or **false** and are used to signify certain states or flag certain events, they can also hold the result for a conditional expression (More on conditional expressions later). These will make more sense when we go thought conditional statements.

```
bool True = true;
bool False = false;
```

Float/Double

Floating point variables are decimal values created with float point precision math, the technical elements of how this works are outside of the scope of this tutorial but can easily be explained through resources on the internet, just search “floating point precision”. Floats allows for a higher level of accuracy of a value by providing decimal precision. You can specify a ‘float’ value by putting an ‘f’ and the end of the value.

```
float decimalValue = 3.0f;
```

Void

This data type is special and is used to specify no value is available. This sounds counter intuitive, but we’ll see where this is used later.

Notable keywords/terms

const – This keyword turns the variable read-only, meaning it's value cannot be changed after it is initialised. The keyword is used like so:

```
const float pi = 3.14f;
```

Global variable – This is a term describing a variable definition outside the main function. For example, the **int** friendCount is a global variable and the **int** currentMonth is not. Note the positions they're defined:

```
//Global
int friendCount = 0;

int main()
{
	//Not Global
	int currentMonth = 5;

	return 0;
}
```

This means the global variable can be used in **any** location in the program and can be dangerous if not used correctly. One correct way it to use it in conjunction with the **const** keyword above, this means functions can only reference the value and not change it.

Recap

- int Holds real numbers
- float / double Hold decimal numbers
- void Specifies there is not type
- boolean Holds either true or false

- string An array of characters making up a word or sentence
- global A variable that can be accessed anywhere
- const Means after a variable has been initialised its value cannot be changed

Conditionals

If statement

There are situations where you need something to happen if a certain condition is the case, this is the role of the If statement and where conditional statements come into the mix.

```
if (Condition)
{
	//Code will run here if Condition is true
}
//Code will jump here if Condition is false
```

In use:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	int numberOfBooks = 10;
	if (numberOfBooks > 0)
	{
	cout << "You have a book!" << endl;
	}
}
```

Output:

> You have a book!

These allow a programmer to control the flow of the program and choose situations to happen when a possibility is true, we'll go onto more examples later.

Else Statement

Else's are optional but these can be added to the end of an if-statement and will only run if the if-statement condition is false

```
if (Condition)
{
    //Code will run here if Condition is True
}
else
{
    //Code will run here if Condition is False
}
```

Else statements can also become an else/if statement where a new if statement is attached, this look like this:

```
if (Condition1)
{
    //Code will run here if Condition1 is True
}
if else(Condition2)
{
    //Code will run here if Condition1 is False and Condition2 is True
}
//If neither are true no code will run
```

You can also chain another else statement onto an else statement effectively creating an infinite chain. If any conditions along the chain are true, the ones below are not checked, I'll demonstrate this below:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	if (false)
	{
	cout << "Contition1!" << endl;
	 }
	else if (true) //This condition is true!
	{
	cout << "Contition2!" << endl;
	}
	else if (true) //Ignored, due to previous else statement being true
	{
	cout << "Contition3!" << endl;
	}
	else if (true) //Also ignored due to condition 2 being true
	{
	cout << "Contition4!" << endl;
	}

	return 0;
}
```

Output

```
> Condition 2
```

After the body of condition 2 is hit and the **cout** statement is executed, the

program does not go onto to check the other else statements.

Exercise

Adapt a program to check if a value is equal to another and print “EQUAL”, if they’re not print “NOT EQUAL”, the skeleton code is below:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	int value1 = 10;
	int value2 = 10;

	//CODE GOES HERE
}
```

Solution

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	int value1 = 10;
	int value2 = 10;

	if (value1 == value2)
	{
	cout << "EQUAL";
	}
```

```
        else
        {
        cout << "NOT EQUAL";
        }

        return 0;
}
```

Change the values of value1 and value2 and see how the program changes.

Using multiple conditions

You can use more than one condition in a single statement, there're two ways of doing this **AND** signified by **&&** and **OR** signified by || (Double vertical bar).

AND checks if both conditions are true before triggering the body of the statement

```
if (Condition1 && Condition2)
{
    //Code will run here if Condition1 AND Condition2 are True
}
```

OR checks if one of the conditions are true before triggering the body of the statement

```
if (Condition1 || Condition2)
{
    //Code will run here if Condition1 OR Condition2 are True (Works if both are true)
}
```

Recap

15. If statement Deals with conditional statements, code within it's body will run if the condition is true
16. else statement Used as an extension to an if statement that is only checked if the if statement is false
17. && Used to string two conditions together and will only return true if both are true
18. || (Double Bar) Used to string two conditions together and will only return true if one of the conditions are true

Switch-Case

Switch cases are used to test a variable for equality with a constant expression without the need for multiple if-statements. One use for this structure is check users input string. Below is the basic structure of the switch case:

```
//Switch-Case
switch (expression)
{
//Case statement
case constant-expression:
        break; //Break isn't needed

//Any number of case statements
// |
// |
//\ /
// .

default:
        break;
}
```

The switch starts off with an ‘expression’, this is the variable that is to be compared to the ‘constant-expressions’, these constants are the literal values of the variable such as ‘1’ or ‘Z’. Breaks are optional but without them the code will ‘flow-down’ into other statements. There is an example below using a switch-case statement, the user inputs a character if the char is N or Y, ‘No’ and ‘Yes’ is output respectively but there’s a default case that applies for all situations, this needs to be at the end.

```
char character;

//Reads in user input (Explained in more detail later)
scanf("%s", &character);

        //Switch-Case
switch (character)
{
         case 'N':
        printf("NO\n");
        break;
        case 'Y':
        printf("YES\n");
        break;
        default:
        printf("Do not understand!\n");
        break;
}
```

If there is no **break** statement a ‘flow-down’ will occur, below is an example just like above but without the break statements and we’ll what the output is like:

```
char character;

//Reads in user input (Explained in more detail later)
char character;
cin >> character;

//Switch-Case
switch (character)
{
```

```
        case 'N':
        printf("NO\n");
        case 'Y':
        printf("YES\n");
        default:
        printf("Do not understand!\n");
}
```

If ‘N’ is the user input the Output will be:

```
>NO
>YES
>Do not understand!
```

This is because when one case statement is triggered it will continue down until a break statement is found to stop running the case body code.

Iteration

Iteration means looping, and looping quickly gives programs the ability to perform lots of similar operations very quickly, there're two types of iteration: 'for loops' and 'while loops/do-while loop'

For Loop

The for loop is given an end value and loops up until that value, while keeping track of what loop number it's currently on, here is an example below:

```
for (int x = 0; x > 10; x++)
{
        cout << "Loop!" << endl;
}
```

```
for(int x = 0; x > 10; x++)
```

Each part of the for-loop has a role

Red

This is the *Declaration* section to define the loop counter variable, this defines the start point of the counter

Green

This section is called the *Conditional* and it contains a conditional statement that is checked at the end of the loop to see if the if-statement should continue looping, so in this case the loop should continue looping if **x > 10,** if this condition becomes false the loop will not continue.

Blue

The blue statement is the *Increment* section where the loop counter is incremented (increased in value), the **x++** is shorthand for **x = x + 1** . This

can also be **x--**, if there was a case requiring the counter to decrease.

14. Declaration section defines the loop counter
15. Conditional section continues the loop if true
16. Increment section is where the loop counter is incremented

Conditional Loops

Conditional loops work like the for loop but don't have a loop counter and will only loop while a condition is True. This means you can create an infinite loop, like so:

```
while (true)
{
        cout << "This will never stop looping!" << endl;
}
```

Note:

An infinite loop is normally constructed using a naked for-loop:

```
for (;;)
{

}
```

This loop will never end and your program will get stuck within the loop.

The example below shows if the condition is false the program will never reach the code within the brackets

```
while (false)
{
        cout << "This will never loop!" << endl;
}
```

To use this loop effectively you can use it with a conditional statement (like the if statement) or you can use it with a bool variable, examples are below

```
int count = 0;
while (count > 10)
{

	//Remember this, it’s the same as “count = count + 1;”
	count++;
}
```

As it says above you can also use a while loop directly with a Boolean variable:

```
int count = 0;
bool keepLooping = true; //Bool is true (1) is true
while (keepLooping)
{
        //Remember this, it's the same as "count = count + 1;"
        count++;

        if (count == 3)
        {
        keepLooping = 0; //keepLooping is now false, and the loop will stop
        }
}
```

Note:
The section in the brackets of the while loop is checking if that condition is true, you can also write it like so:

```
while (!stopLooping) {}
```

This is effectively saying, keep looping while stopLooping is "not true", the "!" is symbol means "not".

Do-While

A Do-While loop is almost exactly the same as While loop but with one small difference, it checks if the condition is true after executing the code in the body of the loop, a while loop checks the if the condition is true before executing the body. The code snippet below shows this difference:

```
while (false)
{
        cout << "While Loop" << endl;
}

do
{
        cout << "Do Loop" << endl;
} while (false)
```

Output:

> Do Loop

Because even though the Boolean is false, the Do loop executed a single time because the check was at the end of the body.

Using a Do-Loop

A real-world example of a do-loop cool be checking for user input, it prints out and asks for input if all is okay there is no need for looping if not it will loop. An example looking for the user to enter a ‘a’ is below

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	//Will loop if this is false
	bool correct = true;
	do
	{
	cout << "Please enter the letter 'a':";

	//Takes in user input
	char a;
	cin >> a;

	//Checks if answer is correct
	if (a != 'a')
	{
	//Incorrect
	correct = false;

	cout << "Incorrect" << endl;
	}
	else
```

```
        {
        //Correct
        correct = true;
        }
        } while (!correct);

        //If the user has completed the task
        cout << "Correct" << endl;
}
```

Loop control keywords

Sometimes there are situations where you want to prematurely stop the entire loop or a single iteration (loop), this is where loop control keywords come into use.

You have either a **break** or **continue** keyword

break – Will stop the entire loop, this can be useful if an answer has been found and the rest of the planned iterations would be pointless.

```
for (int x = 0; x < 5; x++)
{
        if (x == 3)
        {
        break;
        }

        //The technicalities of this statement will be explained later
        cout << "Loop value: " << x << endl;
}
```

Output:

```
> Loop value: 0
> Loop value: 1
> Loop value: 2
```

Now if the code is changed to not include the break:

```
for (int x = 0; x < 5; x++)
{
        //The technicals of this statement will be explained later
        cout << "Loop value: " << x << endl;

}
```

Output:

```
> Loop value: 0
> Loop value: 1
> Loop value: 2
> Loop value: 3
> Loop value: 4
```

Continue – If we use the code from above but replace it for a **continue** the code will look like so:

```
for (int x = 0; x < 5; x++)
{
        if (x == 3)
        {
        continue;
        }

        //The technicals of this statement will be explained later
        cout << "Loop value: " << x << endl;
}
```

The output it like so:

Output:

> Loop value: 0

> Loop value: 1

> Loop value: 2

> Loop value: 4

This shows that when ***x = 3*** the **continue** is executed and the loop is skipped and so is the **cout** statement is also skipped so there is no “Loop value: 3”

Nested loops

You can also place loops within loops to perform specific roles, in the example we are using for-loops but this can also be done with the while/do loop.

The example is printing out a 2D grid, the nested for-loop gives another dimension:

```
//Prints a 5x5 grid
for (int y = 0; y < 5; y++)
{
	for (int x = 0; x < 5; x++)
	{
	//Prints an element of the row
	cout << "X ";
	}

	//Moves down a row
	cout << endl;
}
```

Output

```
> X X X X X
> X X X X X
> X X X X X
> X X X X X
> X X X X X
```

Functions

Functions are the building blocks of a program, they allow the reuse of code, the ability to keep it readable and stops the programmer repeating code. Repeating code is heavily advised against because bugs will be repeated multiple times and changes to code also need repeating. Functions give a centralised controlled area that deals with the distinct roles of the program.

A method has two elements, **parameters**:(the items passed into the function) and the **return type** (the variable being returned) these are both optional and you can have a function with neither.

A function must be defined above its call, like so;

```
//Function call
void PrintSmile()
{
	cout << ":)" << endl;
}

int main()
{
	//Method call
	PrintSmile();

	return 0;
}
```

Function Parameters

Sometimes it might be useful to pass data into a program, there are two types of parameter passing, by **reference** and by **value.** Passing by reference is what it sounds like, it passes a direct reference to a variable not a copy, so

any changes to that passed variable effect it back in the calling function. Passing by value is the passing of a copy of that variable, so any changes to that variable do not effect that passed variable.

16. Passing by reference means changes to the parameter effects the passed variable
17. Passing by value means changes to parameter does not affect the passed variable

Passing by value

```
void Add(int num1, int num2)
{
	//Adds values together
	int newValue = num1 + num2;

	//Prints result
	printf("The result is: %d", newValue);
}

int main()
{
	//Method call
	Add(10, 4);

	return 0;
}
```

Output:

> The result is: 14

Below is another example but this time a string is used a parameter:

```
void PrintStr(char word[])
{
        cout << word << endl;
}
```

The method call looks like this:

```
PrintStr("Flying Squirrel");
```

This is an incredibly useful feature that allows us to make general purpose code and change its function output by what is put in as a parameter.

Passing by reference

Passing by reference is done with the use of pointers (Will be further explained later) but a pointer is a memory address, so any changes done affect the variable being passed in. A reference parameter is shown by the "&" and is done like so:

```
void Change(int& ParaValue)
{
	ParaValue = 20;
}

int main()
{
	int value = 10;

	Change(value);

	cout << value << endl;

	return 0;
}
```

Output

>20

As you can see the change method alters the value of the integer 'value' and effects the value back in the main method.

Returning values

Returning allows us to return data from a method, this lets us do computation within a function and get the function to automatically return the result. Let's take the one of the previous examples and adapt it so it returns the result

instead of printing it:

```
int Add(int num1, int num2)
{
	//Adds values together
	int newValue = num1 + num2;

	//Returned keyword
	return newValue;
}

int main()
{
	//Method call
	int storeResult = Add(10, 4);
	return 0;
}
```

The areas that changed have been highlighted. When a value is to be returned, the “return” keyword is used, after this line has run it returns to the **line where the method was called** so any code under the return **will not run.** The returned result is then stored in ‘storeResult’ to be used later. Returning can happen with any variable type. I’ll show you an example below that checks if the number is an even value (Using the modulus operator talked about above that finds the remainder of a division):

```
 bool EvenNumber(int value)
{
	if (value % 2 == 0)
	{
	//Return true
```

```
        return 1;
        }
        return 0;
}

int main()
{
        if (EvenNumber(2))
        {
        cout << "Even number!" << endl;
         }

        if (EvenNumber(5))
        {
        cout << "Odd number!" << endl;
         }

        return 0;
}
```

Output:

>Even number!

This program uses the return variable form the method EvenNumber() as a conditional for the if-statement and if it's true it will print "Even number!". As you can see from the output the first one prints but the second does not.

Arrays

Arrays have been mentioned previously, it is a data structure that holds a set number of variables next to each other in memory. The array will be given a *type*, for example 'int'. Arrays are used to quickly define lots of variables and

keep relevant variables together. An array is defined below:

A static size, with the size in the square brackets:

```
int lotsOfNumber[20];
```

Or you can define values at the definition, **Note**: a size does not need to be defined because it's automatically determined by the number of values you specify:

```
int lotsOfNumbers[] = {1,3,4};
```

21. **Arrays start at 0**, so the first index has an identifying value of 0, the second is 1 and so on. This means when accessing values, you need to remember it is always one less than the number of values it contains

You can access an index like so:

```
int var = lotsOfNumbers[0];
```

This will grab the first index of the array and place it in 'var'.

Arrays are very useful to access the tightly related data very quickly, you can use a for-loop to loop through the indexes and use them according. An example is below:

```
int lotsOfNumbers[] = { 1, 3, 4, 10};
for (int x = 0; x < 4; x++)
{
        cout << x << endl;
}
```

Output:

```
> 1
```

```
> 3
> 4
> 10
```

In C++ you cannot get the length directly and need to work it out, this can be done simply using the **sizeof()** keyword, this returns the size of the elements in the brackets, so for example on a 64bit machine a **int** should be represented using 4bytes so **sizeof** will return 4. The length of an array can be worked out as so:

```
int arrayLength = sizeof(lotsOfNumbers) / sizeof(lotsOfNumbers[0]);
```

This takes the entire size of the array, and divides it by the first element and the division gives how many indices the array holds.

Multi-dimensional arrays

You can also define a second dimension in the array or even a third, this gives more flexibility when working with arrays. For example, a 2D array could be used to store coordinates or positions of a grid. A 2D array is defined as so:

```
int arrayOfNumbers[][2] = { {1,1}, {1,1}};
```

You access an element by putting the values in the square brackets for the x and y coordinate

```
int element = arrayOfNumbers[X][Y]
```

The obvious difference is that the second dimension **needs** a value, this cannot be automatically resolved when creating an array and that the initialization requires nested curly brackets. Below is the 3D example:

```
int arrayOfNumbers[][][2] = {{{1,1},{1,1}},{{1,1},{1,1}}};
```

But at this stage it is starting to lose readability and it's much better practise to lay it out like so:

```
int arrayOfNumbers[][][2] =
{
        {{1,1},{1,1}},
        {{1,1},{1,1}}
};
```

Passing Arrays in functions

As we saw when we learned about functions above, passing variables in as parameters can be very useful and so can passing in lots of variables stored as an array. However, we learnt above how to determine the size of an array above but this **does not work** with an array passed as a parameter, so we must pass in a variable that represents the number of elements that array is holding. There is a special variable type used to hold count variables, it's called **size_t** and it is an unsigned integer value (Meaning it cannot become negative) used to hold values for a count.

An example of an array being passed as a parameter is below:

```
void Print_SingleDimenArray(size_t length, int ageArray[])
{
	//Length is used to dynamically determine the for loop length
	for (int i = 0; i < length; i++)
	{
	printf("%d\n",ageArray[i]);
	}
}

int main()
{
	//An array of ages
	int ages[] = { 32, 11, 12, 1, 8, 5, 10 };

	//Size is determined as shown previously
	size_t ageLen = sizeof(ages) / sizeof(ages[0]);

	//Method call
	Print_SingleDimenArray(ageLen, ages);
```

```
}
```

Output

```
>32
>11
>12
>1
>8
>5
>10
```

The array has been passed and used to print values. The length of the array is crucial to the program as it allows the for loop to work for arrays of varying length.

22. Any changes to an array in any method will change the array variable in the calling method. This is what was mentioned earlier as **passing by reference,** the whole array isn't copied and passed by reference with a little trickery occurring.

User I/O

There are situations and programming problems that involve the use of user interaction this is where user input and output comes in, the library that deals with this is known as "iostream" and is included into the project like:

```
#include <iostream>
```

Printing/Output

Sometimes the user needs a prompt or information needs to be displayed, this is where console printing comes into use, the key word **cout** is used to print out data to the screen, this is then used with the **stream insertion operator**

that is written as “<<” this is used to signify what is to be printed. And example is below:

```
cout << "Hello!" << endl;
```

The output would just be the word “Hello!” to the screen. The **endl** is just used to print a carriage return (“**\n**”)

You can also create an output involving a variable or another string, multiple stream insertion operators are used:

```
int x = 10;
cout << "The value is: " << x << endl;
```

Output

```
> The value is: 10
```

Reading/Input

Taking in user input can also be useful, the operator used is known as the **stream extraction operator** “>>” and is used in conjunction with **cin**, it works like so:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	//Reads in name
	char name[10];
	cin >> name;

	//Prints name
	cout << name;
}
```

End of chapter Quiz

This section is designed to keep you on your toes about the previous content, there will be 10 questions that you can answer to test your knowledge, below in the neighbouring section will be the answers.

21. Name the data type used to hold a series of characters?’
22. What is the “%” sign used for?
23. What is this sign checking for “==”?
24. If a “&&” is between two conditional statements in a singular if, what does it mean?
25. What does “++” mean at the end of variable mean?

26. What is this operator called: “>>”?
27. Does an array have a built in property to find its size?
28. What is this loop called, and what does it do?

```
for(;;)
{

}
```

29. What two values does a Boolean variable hold?
30. Where is the type ‘void’ used?

Answers

13. String.
14. Gets the modular of two numbers.
15. If the two values either side are equal.
16. That the statement will only be true if **both** of the conditionals are true.
17. It will increment the value of that variable.
18. The stream extraction operator.
19. No, the only way to find it is to use the **sizeof()** operator to find the total size of the array and each elements size, then divide both values to find the size.
20. It is called a naked for loop, and will just loop forever.
21. True or False.
22. Used when signifying a function does not return a value.

Chapter 3: Class Structure and OOP

Classes are the building blocks of a program in C++, they allow easy cookie cutter way of creating places that store and represent data. A class in C++ has two files, the header file (.h) and the implementation file (.cpp).

23. Header file should just contain definitions of classes and variables.
24. The implementation file as the name suggests contains the actual functionality of the functions and variables.

A header file looks like this when empty:

```
class Example
{
public:
	Example();
	~Example();
};
```

There are some key bits about this, the **public** section defines how the methods can be accessed, don't worry too much about this yet as this will be explained further in the Encapsulation section. And the Example() and ~Example() functions are the Constructor and Destructor respectively, and the constructor runs when the class is created, and the destructor runs when the class is destroyed, this can either be done manually by calling the method or the compiler will do this automatically.

The .cpp file then looks like this:

```
#include "Example.h"

Example::Example()
{
}
```

```
Example::~Example()
{
}
```

Each of these are references to the Example's classes. You can implement functions or variables that are not defined in the header file, however this is considered a bad practice.

So for example lets used a fleshed out version of a class to show how it can be used. An example is below:

```
class Shape
{
public:
        Shape(double len, double wid);
         ~Shape();

        double getArea();

private:
        double length;
        double width;
        double area;
};
```

```
#include "Shape.h"
#include <iostream>
using namespace std;

Shape::Shape(double len, double wid)
{
```

```
        length = len;
        width = wid;

        area = length * width;

        cout << "Shape made!" << endl;
}

Shape::~Shape()
{
}

double Shape::getArea()
{
        return area;
}
```

And an instance of shape can be made like so:

```
Shape s = Shape(10, 10);
```

Note: the header file needs including at the top:

```
#include "Shape.h"
```

This gives the programmer a really easy way to create lots of containers for useful data.

Inheritance

Inheritance is one of the most important concepts in object orientated programming, it's the idea of taking a class, the base class and giving other classes it's characteristics like functions and variables. This provides a perfect opportunity to reuse code.

With our shape example above, we can now create another class called “Square” but we inherit characteristics from the base class “Shape”. For displaying purposes I’m going to put all code into a single .cpp file from now on. The Square class is below:

```
#include "stdafx.h"
#include <iostream>

using namespace std;


class Shape
{
public:
	double length;
	double width;
	double area;

	Shape(double l, double w)
	{
	length = l;
	width = w;
	area = l * w;
	}
	~Shape()
	{

	}
	double getArea()
	{
	return area;
```

```
        }
};

class Square : public Shape
{
public:
        Square(double l, double w) : Shape(l, w)
        {
        }
};

int main()
{
        Square s = Square(10, 10);
        cout << s.getArea() << endl;

}
```

Ouput

> 100

Square now has everything that Shape has defined as Public and Protected (These will be explained in the next section) but as mentioned above it's a great way to repeat already created code.

Exercise

Add a class called "Rectangle" and let it inherit from the shape class. Don't forget to include a constructor for the rectangle class.

Solution

Something like this:

```
class Rectangle : public Shape
{
public:
        Rectangle(double l, double w) : Shape(l, w)
        {

        }
};
```

Encapsulation

Encapsulation is the idea of limiting access to sensitive variables or methods contained within a class, there're three different access modifiers:

	Containing Class	Derived Class	Containing Assembly	Outside Assembly
Public				
Protected				
Private				

Above is a grid comparing where you can and cannot access a variable or method marked with either modifier.

14. Containing class is the class that the object was defined.
15. Derived class is any class that inherits another
16. Containing Assembly is anywhere inside the current .exe or .dll file
17. Outside Assembly is any other area that exists

This means you can control the access to variables. But what if a variable is marked as private, what if it needs to be accessed externally? You used

“getters” and “setters”, methods that are specifically designed to retrieve and alter variables. These are a wonderful practise to keep code safe and secure, so let’s alter our shape class to make it more secure.
Before we could do this:

```
Shape s = Shape(10,10);
s.length = -1;
```

This is undesirable as it invalid information, you cannot have a negative length, this is where a setter comes into use, the function can contain code to validate (check) the value that is attempting to be placed in the length variable, something like this:

```
void setLength(double value)
{
	if (value > 0)
	{
	length = value;
	}
	else
	{
	cout << "Error: Value is negative!" << endl;
	}
}
```

This function performs the necessary checks on the value to prevent invalid information getting into variables in the class. Below is an improved version of the class:

```
class Shape
{
private:
        double length;
        double width;
        double area;

public:
        Shape(double l, double w)
        {
        length = l;
        width = w;
        area = l * w;
        }
        ~Shape()
        {

        }

        double getArea()
        {
        return area;
        }
        double getLength()
        {
        return length;
```

```
        }
        double getWidth()
        {
        return width;
        }
}
```

As you can see all the variables have been marked as Private and getters have been added, this is the way classes should be structure.

Polymorphism

The word polymorphism means having many forms and polymorphism in the context of OOP means having one method that can have many roles.

We can use the example of the class shape and overload a method that determines the area if for example we wanted to make a Triangle class, this is the shape class but with a virtual getArea() method:

```
class Shape
{
protected:
        double length;
        double width;
        double area;

public:
        Shape(double l, double w)
        {
        length = l;
        width = w;
```

```
        }
        ~Shape()
        {

        }

        virtual double getArea()
        {
        return length * width;
        }
};
```

Virtual methods give the option be overridden an changed by a class that inherits it, it is also an optional change.

```
class Triangle : public Shape
{
public:
	Triangle(double l, double w) : Shape(l, w)
	{

	}


	double getArea() override
	{
	return 0.5 * (length * width);
	}
};
```

Above is the implementation of the Triangle class, highlighted is the “override” keyword used to change the functionality of the virtual function.

```
int main()
{
	Shape s = Shape(10, 10);
	cout << "The shapes area is: " << s.getArea() << endl;

	Triangle t = Triangle(10, 10);
	cout << "The triangles area is: " << t.getArea() << endl;
}
```

Ouput

```
>The shapes area is: 100
```

```
>The shapes area is: 50
```

As you can see from the example above each getArea() function is implemented differently and returns a different result.

This polymorphic design still gives the ease of creating lots of new data types without the need for repeating code but with a level of customising involved.

Exercise

Add a new class called Circle, it will need a different constrictor that only takes height and the getArea method needs overloading.

Solution

```
class Circle : public Shape
{
public:
	Circle(double h) : Shape(h, 0)
	{

	}

	double getArea() override
	{
	return 3.14 * ((length / 2) * (length / 2));
	}
};
```

As you can see a constructor is provided that only includes the height and the getArea() returns an relatively imprecise area for the circle.

Polymorphism if done correctly can give an extremely clean and readable code base with methods and classes adapting behaviour where necessary.

Interfaces

An interface describes the behaviour of capabilities of C++ class without committing to implementation. Interfaces are implemented using abstract classes, and are just the plan for the class.

Abstracts classes are made by adding a pure virtual functions to a class, a pure virtual function is a function with no body and is specified by placing a "=0" next to the function like so:

```
virtual double getVar() = 0;
```

This means the role the function is just to be inherited and adapted. Failing to override a pure virtual function in an inherited class results in an error, so It's considered a strict blueprint to design classes.

An error seen from not implementing a pure virtual function:

Below is an example involving animals:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

class Animal
{
public:
	//Pure virtual function
	virtual void MakeSound() = 0;
};
```

```
class Dog : public Animal
{
public:
	void MakeSound()
	{
	cout << "Bark!" << endl;
	}
};

class Cat : public Animal
{
public:
	void MakeSound()
	{
	cout << "Meow!" << endl;
	}
};

class Lion : public Animal
{
public:
	void MakeSound()
	{
	cout << "Raw!" << endl;
	}
};

int main()
{
	Dog d;
```

```
        d.MakeSound();

        Cat c;
        c.MakeSound();

        Lion l;
        l.MakeSound();
}
```

Output

```
>Bark!
>Meow!
>Raw!
```

As you can tell it’s a very rudimental pointless example, however it’s supposed to show how a base class can hold basic information and class that inherits it is supposed to add information on top, in this example the program states that anything that inherits animals needs to have a sound, this can be adapted upon to make complex designs for classes and easily fill in relevant implementation.

Another example below will use the classic shape example and have a base class called “Shape” that will have an abstract function “getArea()” that will need implementing for each shape:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

class Shape
{
public:
	//Pure virtual function
	virtual int Area() = 0;

	Shape(double h, double w)
	{
	heigth = h;
	width = w;
	}
protected:
	double heigth;
	double width;
};

class Circle : public Shape
{
public:
	Circle(double h, double w) : Shape(h, w) {}

	int Area()
```

```
        {
        return 3.14 * ((heigth / 2) * (heigth / 2));
        }
};

class Rectangle : public Shape
{
public:
        Rectangle(double h, double w) : Shape(h, w) {}

        int Area()
        {
        return heigth * width;
        }
};

int main()
{
        Rectangle r = Rectangle(10, 20);
        cout << r.Area() << endl;

        Circle c = Circle(5, 5);
        cout << c.Area() << endl;

}
```

Another simple example, but take your time and get used to it, abstract class architecture allows easy addition of new function and classes even if the design has been made due to the easy blueprint-esk design.

End of chapter Quiz

- What are the two files used to make a class?
- What symbol comes before the destructor definition?
- What role should a header file take?
- What does protected mean for a variable that is being inherited?
- What is the role of polymorphism?
- What keyword is used to change a virtual method?
- How do you signify a pure virtual function?
- Where can a function or variable marked as private be accessed?
- What is role of the constructor?
- What should the .cpp file contain?

Answers

- Header (.h) and C++ file (.cpp).
- The symbol "~" I used to show a destructor.
- Just definitions of functions and variables.
- It can only be accessed outside the definition class by a class that inherits it.
- To have a single base definition but lots of different implementations.
- Override is used to change a virtual method
- With a normal function definition that is equal to zero.
- Only in the class where it was defined.
- Is a special function without a return type that is called when an instance of a class is created.
- Implication of functions and variables defined in the header file.

Chapter 4: Improved Techniques

Structures

Structs originate from C, and come from the world of pre-class programming. However, they're still useful and can be use in places where classes are deemed as overkill.

A good example of using a structure being used can be for record keeping, for example you needed to keep track of football players in a team you would need to store:

- Name
- Kit Number
- Wage
- Strongest foot
- Etc

If there is no need to add functionality onto this information a structure is perfect for this situation. A structure storing this information is defined like this:

```
struct FootballPlayer
{
        string Name;
        int KitNumber;
        double Wage;
        string StrongestFoot;
};
```

This structure contains all the values that are relevant, this will provide a relevant container for similar information.

A new instance of the structure is defined like so:

```
FootballPlayer player1;
player1.Name = "Messi";
player1.KitNumber = 10;
player1.Wage = 1000000;
player1.StrongestFoot = "Both";
```

This code will create a FootballPlayer and assign values to all its member variables.

Real world usage of structures tend to be “simple” usage where they need to just store trivial data like the example above, when there needs to be functionality and encapsulation use a class. This is known a POD (Plain Old Data) encapsulation.

Note: The default access for structure variables is **public** while classes default to **private.**

Enums

Enums is short for Enumerated Type as has the role of a boolean but with unlimited user defined types, this can be used a flags for a situation like “HOME_SCREEN” or “SETTING_SCREEN”.

An enum is defined like so:

```
enum CurrentScreen
{
        HOME,
        SETTINGS,
        CONTACTS,
        CALCULATOR
```

```
};
```

With each word in the brackets being a state the enum can be set as. A enum instance is created like so:

```
int main()
{
	CurrentScreen phone1;

	phone1 = HOME;
}
```

This code above makes a new enum and assigns it as “HOME”. You can now use the enum to check what states it is, the example below will use a switch-case:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

enum CurrentScreen
{
	HOME,
	SETTINGS,
	CONTACTS,
	CALCULATOR
};

int main()
{
	CurrentScreen phone1;

	//Change me
	phone1 = HOME;

	switch (phone1)
	{
	case HOME:
	cout << "You're on the home page" << endl;

	break;
	case SETTINGS:
	cout << "You're on the settings page" << endl;
```

```
        break;
        case CONTACTS:
        cout << "You're on the contacts page" << endl;

        break;
        case CALCULATOR:
        cout << "You're on the calculator page" << endl;

        break;
        default:
        break;
        }

}
```

Output

>You’re on the home page

This code shows you can use conditional structures along with an enum to easily create a nice branched structure. Mess around with the commented “Change me” enum definition and see how the output changes.

Unions

Unions are a data structures designed to provide efficient memory usages, they can contain as many member variables as it needs however it can **only** store data for a **single variable** at a time. So the entire structure only takes up memory for the biggest member variable.

These have to be used with caution because setting value of another member variable will remove data stored in another, so it is incredibly easy to lose important data.

A union is defined like so:

```
union Example
{
        int i;
        int x;
        int y;
};
```

And an instance and value is created like so:

```
Example e;
e.i = 4;
```

To show how the size of the union works the program below works it through:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

union Example
{
	int i;
	int x;
	int y;
};

int main()
{
	Example e;
	e.i = 4;

	cout << "Your int size is: " << sizeof(int) << " bytes" << endl;
	cout << "The union has 3 variables that should be a total of:  " <<
sizeof(int) * 3 << " bytes"<< endl;
	cout << "However, the union is " << sizeof(e) << " bytes" << endl;
}
```

Depending on your CPU architecture your output will be:

Ouput

```
>Your int size is: 4 bytes
>The union has 3 variables that should be a total of:  12 bytes
>However, the union is 4 bytes
```

As you can tell the union is only the size of a single integer, these structures can be used to efficiently use memory is there’re situations where variables are idle and only one is needed.

Below is code that shows what happens to all the values when one is changed:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

union Example
{
	int i;
	int x;
	int y;
};

void Print_Union(Example e)
{
	cout << "i: " << e.i << endl;
	cout << "x: " << e.x << endl;
	cout <<	"y: " << e.y << endl;
}

int main()
{
	Example e;

	cout << "1" << endl;
	e.i = 4;
```

```
        Print_Union(e);

        cout << "2" << endl;
        e.y = 3;
        Print_Union(e);

        cout << "3" << endl;
        e.x = 1;
        Print_Union(e);
}
```

Ouput

```
> 1
> i: 4
> x: 4
> y: 4
> 2
> i: 3
> x: 3
> y: 3
> 3
> i: 1
> x: 1
> y: 1
```

As you can tell when you change one value, they all equal the same value, this is because all **values share the same memory location** and therefore when one changes they all change.

Variable argument lists

Variable argument lists allow for the possibly of endless parameters. Before we have come across functions with a static set amount of parameters (variables you can pass into a method) like:

```
int Add(int i, int y)
{

}
```

Has two parameters integer “i” and integer “y”.

This is known as a **Variadic Function** or a function with unlimited **arity**

(parameter number, so “Add” above has an arity of 2)

The variables are stored in a variable known as the **va_list** this works as any other variable but it just holds the list of variables passed. A variadic function is defined like so:

```
void Add(int numberOfVariables, ...)
{
}
```

Where “numberOfVariables” is the number of extra parameters passed, note this does not include this variable. This is required because the va_list does not know the total number of variables and does not keep track of where it is in the list.

Using the va_ist requires the use of a few functions:

- **va_start**(va_list, numberOfVariables)
    - Grabs the variables passed in and assigns the number specified by the second parameter to the first parameter
- **va_arg**(va_list, type)
    - Grabs the next variable from the specified list in the first parameter and creates a variable specified by “type”
- **va_end**(va_list)
    - Tidies up the list specified

We can now use these functions to create a working variadic function:

```
#include "stdafx.h"
#include <cstdarg>
#include <iostream>

using namespace std;

void Add(int numberOfVariables, ...)
{
	//The variable list
	va_list arg;

	//Grabs the list
	va_start(arg, numberOfVariables);

	int total = 0;
	for (int i = 0; i < numberOfVariables; i++)
	{
	//Grabs the next variable
	total += va_arg(arg, int);
	}

	//Clean up
	va_end(arg);
	cout << "Total is: " << total << endl;
}

int main()
{
	//4 Arguments
	Add(4,
	1, 2, 3, 4);
```

```
        //10 Arguments
        Add(10,
        6, 9, 5, 11, 22, 1, 3, 56, 90, 63);
}
```

Ouput

```
>Total is: 10
>Total is: 266
```

Each section of the variable list process is labelled, this function adds all parameters it's given. This gives a very dynamic way to create one function that can handle an almost infinite list of parameters.

Exercise

Create a function that multiplies all the parameters given and returns a value, print out that value. Note: this #include is required:

```
#include <cstdarg>
```

Solution

Something like this:

```
#include "stdafx.h"
#include <cstdarg>
#include <iostream>

using namespace std;

int Multi(int numberOfVariables, ...)
{
        va_list arg;
        va_start(arg, numberOfVariables);
```

```
        int total = 1;
        for (int i = 0; i < numberOfVariables; i++)
        {
        //Grabs the next variable
        total *= va_arg(arg, int);
        }
        va_end(arg);
        return total;
}

int main()
{
        cout << "The total is: " << Multi(3, 56, 90, 63) << endl;
}
```

Namespaces

Namespaces are used to prevent variables and functions with the same name getting mixed up, for example the function Run() can be applicable in many situations so there needs to be a way to distinguish between different Run() functions, this is where namespaces can be used.

They are defined like so with all functions and variables in the curly brackets being part of that namespace:

```
namespace NamespaceName
{

}
```

The example below shows how they are used:

Two namespaces are defined like so with the same function name:

```
namespace Proccesing
{
	void Run()
	{
	cout << "Processing!" << endl;
	}
}

namespace Initialisation
{
	void Run()
	{
	cout << "Initialisation started!" << endl;
	}
}
```

This gives the programmer the opportunity to pick which Run() they can execute by just specifying the namespace like so:

```
Proccesing::Run();
```

and

```
Initialisation::Run();
```

and with them both run together:

```
int main()
{
	Initialisation::Run();
	Proccesing::Run();
```

```
}
```

Output

```
> Initialisation started!
> Processing!
```

Namespaces therefore gives a way to neatly segment code, and improves readability by providing a way to almost label blocks of code.

End of chapter Quiz

- What is the recommended situation you should use a struct?
- What is enum short for?
- How the size of a union dictated?
- What is a key factor about a union?
- What is a Variadic function?
- What does a va_list hold?
- Why do you need to specified the number of parameters in a variable function?
- Why does va_end() need to be run?
- What is the base role of a namespace?
- What syntax is used when calling a function inside a namespace, say you're calling the function Add() inside the namespace Math?

<u>Answers</u>

- For plain old data (POD).
- Enumerated type.
- It's the size of its largest member variable
- All the member variables share the same memory location so only on variable can hold a value at once
- A function with a possibly unlimited number of parameters
- A list of passed in variables to a function
- There is no way to tell the size of a va_list, so you need to specify the number of passed parameters
- It performs the clean-up of the variable list when it has been used
- To segment and label variables and functions in a meaningful way.
- Math::Add()

Chapter 5: Advanced Features

File I/O

You will eventually come across a programming situation where you need data to exist between the program running and not running, this is where file saving comes into use.

File saving is achieved by using functions in the **fstream** library, that is included like this:

```
#include <fstream>
```

You also need “iostream” that we’ve come across before for “cout” and “cin”

```
#include <iostream>
```

There are three main objects that are used when saving and reading to file:

- ofstream
    - This deals with the output stream, and is used to create and write data to a file
- ifstream
    - This is the opposite of ofstream and is used to read data from files
- fstream
    - This is a function that has the characteristic of both ofstream and ifstream.

Opening a file

To open a file we need to use the “open()” function that is a member of all the main objects, it works like so:

```
fstream f;
```

```
f.open("file.txt", ios::in);
```

Where:

- “file.txt” Is the filename
- “ios::in” is the flag for how the file is to be open, below is a table explaining all the options.

Name	Description
ios::app	Opens the file in append mode, all input is to be added to the end of the file
ios::ate	Opens a file for input into a file, and moves the read/write control to the end of the file
ios::in	Opens a file for reading
ios::out	Opens a file for writing
Ios::trunc	If the file exists, the data inside is deleted (truncated)

You can also have two of these modes by **OR**ing them together, this can be useful if you want to open the file to write to it and you want to get rid of already existing data:

```
f.open("file.txt", ios::out || ios::trunc);
```

Closing a file

Closing a file is as simple as you’d image. To achieve this all you need to call is the “close()” function:

```
f.close();
```

This flushes the stream and releases all the allocated memory to the file.

Writing to a file

Writing to a file is achieved by the use of the use of the insertion operator (<<), this has been met before in the case of “cout” and all that need to be done is data needs to be streams to a fsteam or ofstream instead of cout.

This is done like so:

```
f << "Writing data!" << endl;
```

Reading data

Reading data is the same as writing data, just flipped. You use the stream extraction operator (>>) much like we’ve done before with **cin** but instead use it with **fstream** or **ifstream**.

It’s also done like so:

```
char data[100];
f >> data;
```

Note: Reading like this will read all data up until a space or newline

Working example

Above is all you need to know to read a write to a file, below is working code example with reading and writing:

```
#include "stdafx.h"
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
	//Opens a file for writing
	fstream write;
	write.open("file.txt", ios::out);

	//Writes data to file
	write << "Example Data" << endl;
	write << "More example data" << endl;
	write.close();



	//Opens a file for reading
	fstream read;
	read.open("file.txt", ios::in);

	//"!read.eof()" = Read while not end of file
	//.eof() is a bool to flag if the end of the file has been reached
	char data[100];
	while (!read.eof())
	{
```

```
        //Reads in
        read >> data;

        //Prints
        cout << data << endl;
        }

        read.close();
}
```

Output

```
>Example
>Data
>More
>example
>data
```

As you can tell, each read command is stopping when it reaches a space or a newline (as mentioned above), in this case it just complicates the read section because now we have all the sentences separated. One way we can fix this is by using “getline()”, We’ll adapt the above example to read whole lines, changes will be highlighted:

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
	//Opens a file for writing
	fstream write;
	write.open("file.txt", ios::out);

	//Writes data to file
	write << "Example Data" << endl;
	write << "More example data" << endl;
	write.close();


	//Opens a file for reading
	fstream read;
	read.open("file.txt", ios::in);

	//"!read.eof()" = Read while not end of file
	//.eof() is a bool to flag if the end of the file has been reached
	string data;
	while (!read.eof())
	{
	//Reads in
	getline(read, data);
```

```
        //Prints
        cout << data << endl;
        }

        read.close();
}
```

Output

>Example Data

>More example data

The <string> library has been included as to access the getline() method, this method uses the “read” stream as the first parameter and the second parameter being the place for the data to be stored. As you can tell “data” has also been adapted to a string, this just allows a cleaner way of reading in because we don’t need to specify length.

Recursion

A recursive function is a function that contains within its definition a call to itself, this therefore creates a loop and if you’re not a carful an infinite one.

Recursion can be very useful as it creates iteration with a small code base, below is a recursive example:

```
void Recursive(int count)
{
        if (!(count <= 0))
        {
        //Prints current value of count
        cout << count << endl;
```

```
        //Makes recursive call
        Recursive(count - 1);
        }
}
```

All this function does is count down from the initial value that's it's called with, so:

```
Recursive(5);
```

Will output:

```
>5
>4
>3
>2
>1
```

The key bit about the function above is the if statement, this makes sure that recursive call stops, this is known as the stopping condition as stops the program from crashing due to overflowing function calls.

A more useful example that you'll see everywhere if you're researching recursion is a factorial example, for anyone that needs refresh a factorial is the product of all the positive integers from 1 to a given number, so the factorial of 4 (written like 4!) is:

```
4! = 4 * 3 * 2 * 1 = 24
```

And the recursive code looks like this:

```
int Factorial(int value)
{
```

```
        if (value > 0)
        {
        return value * Factorial(value - 1);
        }
        else
        {
        return 1;
        }
}
```

As you can see there is a function call in a return statement, you'll use this a lot in recursion as it means the function will fall into sort of like a recursive whole, hit the bottom (in this case "return 1") and use this information at the bottom to fill in the gaps on the way up and determine an answers back at the top. This all gives a different way to deal with iteration in a clean concise way.

Exercise

This is a hard exercise and it aims to try and trigger a good understanding of recursion, so take your time and do some background research prior to starting. In this exercise I would like you to create a recursive function that calculates Fibonacci's sequence with all the values below 1000, printing them as you go.

Start it off with a call like so:

```
Fibonacci(0, 1);
```

Note: You'll need two parameters, previous and next.

Solution

Something similar to this:

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int Fibonacci(int previous, int next)
{
	//Print
	cout << previous << endl;

	if (next < 1000)
	{
	int newNext = previous + next;

	return Fibonacci(next, newNext);
	}
}

int main()
{
	Fibonacci(0, 1);
}
```

Output

```
> 0
>1
>1
>2
>3
>5
>8
>13
>21
>34
>55
>89
>144
>233
>377
>610
>987
```

Remember there are always different ways of doing things, so if you got the same output it’s still correct!

Pre-processors

Are just instructions for the compiler to process before any code is complied, they all begin with a (#) and do not end with a semi colon (;) as they are not statements.

#define

Define is used to create a definition of a symbol or word, the most common

example is using #define to specify if the program is in debug mode, this allows you to only run debug code when the debug flag is defined. You can also use them to define constant no-changing values, an example is below:

```
#include "stdafx.h"
#include <iostream>

#define NUMBER 10

using namespace std;

int main()
{
	cout << NUMBER << endl;
}
```

Output

```
>10
```

This is used as kind of an alias system, not variable is defined here but you can put very important values in these define statements and create more readable code.

You can also use #define statement to create macro-functions, it is done like so:

```
#define BIGGEST(a,b) ((a < b) ? b : a)
```

This is effectively saying if b is greater than (<) a return (?) b as the value else (:) return a.

And is called like any other function:

```
cout << BIGGEST(10, 2) << endl;
```

One of the most useful real world uses of this would be to create a better syntax for a for loop within the #define, this is just for syntax purposes:

```
#define For(index, max) for(index = 0; index < max; index++)
```

All this does is just define a for loop within the #define body, you just put the list and the value you want to index and you then have a tidy syntax.

I looks like this when used:

```
int i;
For(i, 10)
{
	cout << i << endl;
}
```

Much nicer than the equivalent:

```
for (int i = 0; i < 10; i)
{
	cout << i << endl;
}
```

Please note these do the **exact** same thing, it's just a different syntax.

Conditional Compilation

This is used in conjunction with the #define, there is **#ifdef** (If defined) and **#ifudef** (If undefined) that both check for #define statements. It can be used like:

```
#ifdef VERBOSE
	cout << "Print!" << endl;
#endif
```

It surrounds code and say for example there is not a #define above the code will be ignored.

```
#define VERBOSE;
```

#ifudef is just the opposite and will ignore code if there is a #define.

The # and ## operators

The # operator replaces a macro with that given string, and example is below:

```
#define CREATE_STRING(x) #x;
```

Then when it's called it will be replace the entire call with "x", so:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

#define CREATE_STRING(x) #x

int main()
{
	cout << CREATE_STRING(STRING EXAMPLE) << endl;
}
```

Output

```
>STRING EXAMPLE
```

This is because:

```
CREATE_STRING(STRING EXAMPLE)
```

Calls:

```
#define CREATE_STRING(x) #x
```

Where:

```
#define CREATE_STRING(STRING EXAMPLE) #STRING EXAMPLE
```

And

```
#STRING EXAMPLE
```

Is equal to:

```
"STRING EXAMPLE"
```

Operator

The double hash operator concatenates two items into one, this is better explained with an example:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

#define MAKE_ONE(a, b) a ## b

int main()
{
	int numone = 100;

	cout << MAKE_ONE(num, one) << endl;
}
```

Output

```
>100
```

Why has this happened? Let’s walk it through:

The double hash takes two items and concats them so:

```
MAKE_ONE(num, one)
```

Resolves to:

```
#define MAKE_ONE(num, one) num ## one
```

And;

```
num ## one
```

is equal to:

```
numone
```

So:

```
cout << MAKE_ONE(num, one) << endl;
```

Equals

```
cout << numone << endl;
```

And the variable above is called and the output is 100.

Dynamic memory and pointers

To understand pointers you need to understand how variables are stored. Variables are allocated a memory location, and the size is dependent of the size of the variable. A pointer is a variable that holds the address of that location, so it allows direct access to a variable.

A pointer is defined like so:

```
int* p;
```

You then need to assign the pointer to point at an actual memory location, to do this we need to use the reference operator (&). The code below assigns a variable for the pointer to point at:

```
#include "stdafx.h"
```

```
#include <iostream>

using namespace std;

int main()
{
	int i = 10;
	int* p = &i;
}
```

This defines a pointer to an integer, lets print out the value just after initialisation and see what it looks like, by adding the code below to the bottom of the body of the main method:

```
cout << p << endl;
```

Output

> 008FF768

This is the address that holds the integer “I”

Note: This value will be different every time

This allows you to pass values by reference, if you were to pass the pointer as a parameter any changes to the pointer will result in a change to the base variable, the example below demonstrates this:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

void Change(int* pointer)
{
```

```
        *pointer = 200;
}

int main()
{
        int i = 10;
        int* p = &i;

        cout << "Before: " << i << endl;

        Change(p);

        cout << "After: " << i << endl;
}
```

Output

>Before: 10

>After: 200

As you can tell the value of “I” has changed but without the direct changing to the variable.

Pointers can point to any data type, you can even have pointers to objects.

New and delete operators

The **new** operator allows manual allocation of memory for a data type, see the example below:

```
double* p = new double;
```

As you can tell the **new** keyword allocates memory and returns a pointer to that memory, in the case of the example above it’s a pointer to a double, this is manually allocated memory and the compiler will not manually release the data and if this isn’t manually rectified there can be what is called a **memory**

leak where memory is taken up unnecessarily.

This is where the **delete** operator comes into use, this is the opposite to the **new** keyword and will deallocate the memory for the variable.

Let's put the operators into work:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	//Creates pointer
	double* p = NULL;

	//This checks to see if memory has been allocated
	//It just checks to make sure "p" is pointing to something
	if (!(p = new double))
	{
	cout << "ERROR: Memory problems" << endl; //Will only print if there is a problem
	}

	//Changes value etc
	*p = 100;
	cout << "P is: " << *p << endl;

	//Deallocates memory
	delete p;
}
```

As you can see there is a check for the allocation of memory for p, this is a recommended practise because it keeps tabs on any memory problems, if you were to just do this:

```
double* p;
p = new double;
```

If the second line was to fail p could be pointing to any memory location and would cause a crash if the value of p was changed.

Dynamic memory for arrays

As you can do with variables, you can also do with arrays with the:

13. **new[]** to allocate a chunk of memory
14. **delete[]** to deallocate chunks of memory

So for example if we wanted to make an array of pointers to 5 objects:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

class MyClass
{
public:
	MyClass()
	{
	cout << "Made!" << endl;
	}
	~MyClass()
	{
	cout << "Deleted!" << endl;
	}

};

int main()
{
	//Create
	MyClass* classes = new MyClass[5];
```

```
        //Delete
        delete[] classes;
}
```

Output

```
>Made!
>Made!
>Made!
>Made!
>Made!
>Deleted!
>Deleted!
>Deleted!
>Deleted!
>Deleted!
```

As you can see the first line in the main() creates 5 MyClasses, the constructors fire and print “Made!”, the next link then goes onto free the memory the destructors fire and print “Deleted!”, this can be used to create a lot of pointers with a very small amount of code.

Pointer arithmetic

You can add a value onto a pointer to move it along to the next memory location, let’s say we have a pointer to an integer, and on a 64bit system an int should be 4bytes. So:

```
int* p = new int;
p = p + 1;
```

The code above will move to the next 4 bytes, by “+1” you say move to the next variable sized gap, so for example if we had a “long long” that should be

8 bytes adding one will move 8 bytes along. You can also do this in the opposite direction and take away one, this does the exact same just in the other direction.

The example below demonstrates this:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
	//A size check
	cout << "int is: " << sizeof(int) << " bytes" << endl;

	//Creates memory
	int* next = new int[10];

	for (int i = 0; i < 10; i++)
	{
	//Moves along 4 bytes
	cout << next + i << endl;
	}

	//Deletes memory
	delete[] next;
}
```

Output

```
>int is: 4 bytes
>00DDDB50
>00DDDB54
>00DDDB58
>00DDDB5C
>00DDDB60
```

```
>00DDDB64
>00DDDB68
>00DDDB6C
>00DDDB70
>00DDDB74
```

As you can see, memory is allocation and the pointer is moved along by 1 each time. As you can see from the output the hexadecimal addresses are 4 different from each other because the memory addresses are adjacent.

End of chapter Quiz

- What does the flag “ios::app” mean?
- Why is it important to close a file?
- Which operator is used to writing data to a file?
- What does .eof() signify?
- What does every recursive function need?
- What does #ifudef mean?
- What does the “##” operator achieve?
- When should “delete []” be used?
- What does the “new” operator return?
- What does adding 1 to a integer pointer?

Answers

- It will open the file in append mode.
- It releases resources attached to the file.
- Insertion operator (<<).
- Is a method that returns true when the end of the file is reached.

- A stopping condition to prevent endless looping.
- Will include the code it encapsulates if the alias is undefined.
- It concatenates and replace two values.
- When you want to delete a block of user assign memory.
- A pointer to a newly created memory location
- *If the integer is 4 bytes it will move along 4 bytes.*