# NEA support guide for C# - Programming techniques

# Introduction

This guide is designed to support candidates during the NEA Tasks and may be included as part of the resource bank that they have access to.

Disclaimer: Please note that this is not a complete guide to C# and only explores some of the ways to use C# to express the techniques in the specification.

Once downloaded from the website, this document becomes *uncontrolled*.

# GCSE Programing techniques

### Contents

[The use of variables 4](#_Toc483558892)

[Keywords for C# data types 5](#_Toc483558893)

[Variable naming 6](#_Toc483558894)

[Constants 6](#_Toc483558895)

[Operators 7](#_Toc483558896)

[Relational/Equality operators 7](#_Toc483558897)

[Arithmetic operators 7](#_Toc483558898)

[Using DIV in C# 9](#_Toc483558899)

[Logical operators 10](#_Toc483558900)

[Precedence 11](#_Toc483558901)

[Inputs 11](#_Toc483558902)

[Entering strings/text 11](#_Toc483558903)

[Entering numbers 11](#_Toc483558904)

[Outputs and assignments 13](#_Toc483558905)

[Sequence 14](#_Toc483558906)

[Selection 14](#_Toc483558907)

[IF or CASE Select? 15](#_Toc483558908)

[Nesting IF statements 15](#_Toc483558909)

[Logic errors in IF statements 16](#_Toc483558910)

[Iteration 17](#_Toc483558911)

[FOR Loops/Count Controlled 17](#_Toc483558912)

[Counting forwards 18](#_Toc483558913)

[Counting backwards 19](#_Toc483558914)

[Condition controlled loops 19](#_Toc483558915)

[WHILE Loops 19](#_Toc483558916)

[DO WHILE loops 21](#_Toc483558917)

[Infinite loops 21](#_Toc483558918)

[Using break and continue key words 23](#_Toc483558919)

[The use of the break in loops 23](#_Toc483558920)

[The use of the break in SWITCH / Case Select 23](#_Toc483558921)

[The use of the continue keyword 23](#_Toc483558922)

[Examples of break and continue in loops 24](#_Toc483558923)

[The use of basic string manipulation 25](#_Toc483558924)

[Placeholders 26](#_Toc483558925)

[‘Contains’ within a string 27](#_Toc483558926)

[ASCII values 27](#_Toc483558927)

[Substrings 28](#_Toc483558928)

[Strings cutting from the left 28](#_Toc483558929)

[String cutting from the right 28](#_Toc483558930)

[File handling 29](#_Toc483558931)

[StreamReader 29](#_Toc483558932)

[The use of close 30](#_Toc483558933)

[Reading a file line by line 31](#_Toc483558934)

[Reading a whole file at once 32](#_Toc483558935)

[Write to a file (overwrite) 32](#_Toc483558936)

[Write to a file (append) 33](#_Toc483558937)

[The use of arrays 34](#_Toc483558938)

[Declaring an array 34](#_Toc483558939)

[Initialising an array with data items 34](#_Toc483558940)

[Assigning data items to an array after initialisation 35](#_Toc483558941)

[Sorting arrays 36](#_Toc483558942)

[Reverse sorting arrays 37](#_Toc483558943)

[Printing arrays 38](#_Toc483558944)

[Using a FOR loop 38](#_Toc483558945)

[Using a FOR EACH loop 39](#_Toc483558946)

[The use of records to store data 40](#_Toc483558947)

[Printing a record structure 41](#_Toc483558948)

[How to use sub programs 42](#_Toc483558949)

[Creating a static function 43](#_Toc483558950)

[Creating a non-static function 43](#_Toc483558951)

[Functions with parameters 43](#_Toc483558952)

[Single Parameter functions 43](#_Toc483558953)

[Multiple parameter functions 43](#_Toc483558954)

[Using functions within a program 44](#_Toc483558955)

[Using returned values 44](#_Toc483558956)

[Where should I use functions 45](#_Toc483558957)

[Casting 46](#_Toc483558958)

[Random numbers 47](#_Toc483558959)

[Recursion 50](#_Toc483558960)

[Scope 51](#_Toc483558961)

[Local variables 51](#_Toc483558962)

[Avoiding scope issues 52](#_Toc483558963)

[Global variables 52](#_Toc483558964)

[The use of SQL to search for data 53](#_Toc483558965)

# The use of variables

**Pseudocode**

|  |  |
| --- | --- |
| x=3 name= ”Bob” | Variables and constants are assigned using the = operator. |
| global userid = 123 | Variables in the main program can be made global with the keyword global. |

**C# Syntax**

|  |  |
| --- | --- |
| int parrotAge;  string parrotStatus = "Alive"; | C# requires you to declare the variable type at the same time as the variable name.  However it does not require you to assign a value to the variable straight away. |
| int parrotAge;  string parrotStatus = "Alive";  parrotAge = 12;  System.Console.WriteLine("The parrot is currently " + parrotAge + " and is " + parrotStatus); | Once assigned you can use the variable with other values or variables as shown.  Note that if you leave the variable ‘null’ when you declare it, you must assign a value to the variable first before being able to carry out further operations on it. Screen shot |
| parrotAge = parrotAge + 1;  Console.WriteLine(parrotAge); | A variable can be overwritten with a new value at any time. |
| Screen shot | You cannot assign data to a variable of different types. Each variable will only hold the data type defined.  As you can see – an error is shown in the IDE as “two” is a string, and we are trying to assign it to an ‘int’. |

# Keywords for C# data types

|  |  |
| --- | --- |
| **Data Type** | **Range** |
| **byte** | 0 .. 255 |
| **sbyte** | -128 .. 127 |
| **short** | -32,768 .. 32,767 |
| **ushort** | 0 .. 65,535 |
| **int** | -2,147,483,648 .. 2,147,483,647 |
| **uint** | 0 .. 4,294,967,295 |
| **long** | -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807 |
| **ulong** | 0 .. 18,446,744,073,709,551,615 |
| **float** | -3.402823e38 .. 3.402823e38 |
| **double** | -1.79769313486232e308 .. 1.79769313486232e308 |
| **decimal** | -79228162514264337593543950335 .. 79228162514264337593543950335 |
| **char** | A Unicode character. |
| **string** | A string of Unicode characters. |
| **bool** | True or False. |

## Variable naming

There are some basic rules with variable names in C#:

* the first letter of a variable must be either a letter, underscore (\_) or the @ symbol.
* after this, they can be any combination of letters, underscores or characters
* they can only be one word
* they can only use letters, numbers and underscores (\_)
* hyphens are not allowed (-)
* spaces are not allowed
* they can’t begin with a number
* special characters are not allowed such as $ or ‘.

Remember:

* variable names are case sensitive, SPAM and spam are different variables
* it is convention to use a lower case letter at the start of a variable name
* you can use camelCase or not\_camel\_case
* a good variable name describes the data it contains
* the variable type always comes before the variable name in C#.

# Constants

**Pseudocode**

|  |  |
| --- | --- |
| const vat = 20 | Variables in the main program can be made constant with the keyword const. |

**C#**

|  |  |
| --- | --- |
| const int ParrotAge = 0; | The keyword for declaring a constant is **const**. It is used BEFORE the data type and declaration of the variable.  If you try to use the keyword **const** and do not declare a value to the variable, then it will throw an error! As you can see – there is a warning line at the semi colon.  **Remember a constant cannot have a new value assigned to it during run time.** |

# Operators

## Relational/Equality operators

|  |  |
| --- | --- |
| == | Equal to |
| != | Not equal to |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |

When using Logical Operators, the answer to a comparison is always TRUE or FALSE (i.e. a Boolean result. Have a look at what happens with the following code, and results:

|  |  |
| --- | --- |
| int valueA = 23;  int valueB = 15;  Console.WriteLine(valueA == valueB);  Console.WriteLine(valueA != valueB);  Console.WriteLine(valueA < valueB);  Console.WriteLine(valueA <= valueB);  Console.WriteLine(valueA > valueB);  Console.WriteLine(valueA >= valueB); | Screen shot |

You can also save these results as a variable.

|  |  |
| --- | --- |
| int valueA = 23;  int valueB = 15;  bool myResult = false;  myResult = valueA != valueB;  Console.WriteLine("My Result = " + myResult); | Screenshot |

## Arithmetic operators

|  |  |
| --- | --- |
| + | Addition e.g. x=6+5 gives 11 |
| - | Subtraction e.g. x=6-5 gives 1 |
| \* | Multiplication e.g. x=12\*2 gives 24 |
| / | Division e.g. x=12/2 gives 6  **NB** Using integers will result in DIV being applied |
| % | Modulus e.g. 12MOD5 gives 2 |
| Math.Pow(A, b); | Exponentiation e.g. Math.Pow(3, 4) gives 81  **Note:** both numbers need to be **double** for this to wor**k.** |

**Examples of Arithmetic operators:**

|  |  |
| --- | --- |
| int valueA = 23;  int valueB = 15;  Console.WriteLine(valueA + valueB);  Console.WriteLine(valueA - valueB);  Console.WriteLine(valueA \* valueB);  Console.WriteLine(valueA / valueB);  Console.WriteLine(valueA % valueB); | Screenshot |
| double result, number1, number2;  number1 = 2;  number2 = 2;  result = Math.Pow(number1, number2);  Console.WriteLine(number1 + " ^ " + number2 + " = " + result); | Screenshot |

### Using DIV in C#

C# does not have a DIV specific reserved word to use. DIV already exists when you do integer division.

Entering A / B = C, where A and B are set as integers will give you a DIV. See this example:

|  |
| --- |
| int ParrotAge\_TotalDays = 745;  int ParrotAgeYears = 0;  int ParrotAgeDays = 0;  ParrotAgeYears = ParrotAge\_TotalDays / 365;  ParrotAgeDays = ParrotAge\_TotalDays % 365;  Console.WriteLine("The parrot is " + ParrotAgeYears + " years and " + ParrotAgeDays + " days old!"); |
| Screenshot |

## Logical operators

* AND: ==
* OR: ||

Logical Operators chain relational operators together, where you may need two or more things to be either TRUE or FALSE to continue.

|  |
| --- |
| string caster = "witch";  string victim = "peasant";  bool later = true;  string spellCast = "She turned me into a newt";  string victimStatus = "I got better!";  if ((caster == "witch") && (victim == "peasant"))  {  Console.WriteLine(spellCast);  } |
| Screen shot- Both conditions here evaluate to true  - Therefore the IF statement condition evaluates to true  - Thus the IF statement is executed |

|  |  |
| --- | --- |
| string caster = "witch";  string victim = "peasant";  bool later = true;  string spellCast = "She turned me into a newt";  string victimStatus = "I got better!";  if ((caster == "witch") && (victim == "dog"))  {  Console.WriteLine(spellCast);  } | Here the IF statement will not execute because the victim is not a dog.  BOTH conditions must be true for the IF statement to execute. |
| string caster = "witch";  string victim = "peasant";  bool later = true;  bool lie = true;  string spellCast = "She turned me into a newt";  string vicitmLies = "We believe you are lying!";  string victimStatus = "I got better!";  if ((caster != "witch") || (lie == true))  {  Console.WriteLine(vicitmLies);  } | Screen shot  - At least one condition is true  - Therefore the IF statement condition evaluates to true  - Thus the IF statement is executed |

## Precedence

C# uses standard orders or precedence. When using long calculations, remember that how you use brackets can change the results significantly!

|  |  |
| --- | --- |
| int valueA = 23;  int valueB = 15;  int valueC = 15;  int valueD = 4;  int resultA, resultB;  resultA = valueA + valueB / valueC - valueD;  resultB = (valueA + valueB) / (valueC - valueD);  Console.WriteLine("Result A = " + resultA);  Console.WriteLine("Result B = " + resultB); | Screenshot  - Note that adding brackets has changed the result.  - Remember BODMAS (or BIDMAS!) |

# Inputs

**Pseudocode**

|  |  |
| --- | --- |
| Variable=input(prompt to user)  Name=input(“What is your name”) | Here we declare a variable and assign the input to it. We also pompt the user as to what to input. |

**C#**

Everything a user enters at the console is treated as text. C# will not convert say a number you enter into an integer, or a double. You need to tell it to do that. Therefore there are two ways you can enter data into C#

### Entering strings/text

|  |  |
| --- | --- |
| string myName = "";  Console.Write("Please enter your name: ");  myName = Console.ReadLine();  Console.WriteLine("Your Name: " + myName); | Screen shot |

### Entering numbers

Numbers need to be Parsed on entry. You can parse to any numerical data type. Remember to try and keep the variable data type the same as the data type you are parsing to! If you do not, the compiler **may** not let you compile your code!

|  |  |
| --- | --- |
| int valueA = 23;  Console.Write("Please Enter an Integer: ");  valueA = int.Parse(Console.ReadLine());  Console.WriteLine("Your Value: " + valueA); | Screen shot |

However, as long as the entry you parse is of lower accuracy than the variable, then it will allow you to parse, and then store the number entered.

**Parsing a float to be stored as an int will NOT work:**

|  |  |
| --- | --- |
| int valueA = 23;  Console.Write("Please Enter an Integer: ");  valueA = float.Parse(Console.ReadLine());  Console.WriteLine("Your Value: " + valueA); | Screenshot  - As you can see from the screenshot above, the compiler has flagged an error with the Parse attempt! |

**Parsing an int to be stored as a float WOULD work:**

|  |  |
| --- | --- |
| float valueA = 23;  Console.Write("Please Enter an Integer: ");  valueA = int.Parse(Console.ReadLine());  Console.WriteLine("Your Value: " + valueA); | Screenshot |

# Outputs and assignments

**Pseudocode**

|  |  |
| --- | --- |
| print (string)  print (variable) | Outputs the argument (string or variable) to the screen. |

**C#**

|  |  |
| --- | --- |
| C# uses a library to input and output to console. It should be imported already, when you create a new project within your IDE.  Without the highlighted text (right) the System Libraries would not be imported, and therefore you could not use the Console.WriteLine() method. | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.IO; |
| Console.WriteLine("How do you know she's a witch?"); | |
| Here the Console.WriteLine method takes the string and outputs to the screen. | |
| string myAnswer = "She turned me into a newt!";  Console.WriteLine("How do you know she's a witch?");  Console.WriteLine(myAnswer); | |
| Screenshot | |

# Sequence

**Pseudocode**

|  |  |
| --- | --- |
| x=3  y=2  x=x+y  print (x)  5 | x is assigned the value of 3, y is assigned the value of 2. x is then re-assigned to be the value of 3 plus 2 which evaluates to 5 and is printed to the screen.  It should be noted that that value of x changes in sequence, line by line as it is interpreted, at the start of line 3 (x=x+y) x still has a value of 3 but once that line is run it then changes to be x+y or 5. |

**C#**

|  |  |
| --- | --- |
| int x, y;  x = 3;  y = 2;  x = x + y;  Console.WriteLine(x); | Screenshot |

# Selection

**Pseudocode**

|  |  |
| --- | --- |
| if entry==“a” then  print(“You selected A”)  elseif entry==“b” then  print(“You selected B”)  else  print(“Unrecognised selection”)  endif | Selection will be carried out with if/else and switch/case. In this example, the pseudocode is checking the input and returning a message based upon the specific input given, the else block is used as a catch for any unexpected input which allows the code to degrade gracefully.  The switch/case method works in the same way. |

|  |  |
| --- | --- |
| switch entry:  case “A”:  print(“You selected A”)  case “B”:  print(“You selected B”)  default:  print(“Unrecognised selection”)  endswitch | The CASE SELECT is slightly different in that you always have to have the ‘default’ case.  This effectively creates a ‘Catch’ for when invalid entry is recognized. In an IF statement you would have to specifically create an ELSE construct to cope for this. |

### IF or CASE Select?

IF statements use relational operators to make decisions. Is a variable equal to that variable, is it greater than etc.

Case Selects only look for exact matches, via the ‘case’ line. Therefore you have to make a decision as to what choice you are making, as to which may be most appropriate.

**C#**

|  |  |
| --- | --- |
| int duckWeight = 15;  int personWeight = 13;  if (duckWeight >= personWeight)  {  Console.WriteLine("Clearly not a witch!");  }  else  {  Console.WriteLine("She's a Witch!");  } | Screenshot  Here the weight of the person is not greater than that of the duck, and therefore therefore the IF statement executes, and the ELSE is skipped. |
| int duckWeight = 15;  int personWeight = 34;  if (duckWeight >= personWeight)  {  Console.WriteLine("Clearly not a witch!");  }  else  {  Console.WriteLine("She's a Witch!");  } | Screen shot  Here the weight of the person is greater than the duck. This means that the IF statement is FALSE and does not run.  Therefore the ELSE statement executes by default. |

### ELSE IF statements

|  |  |
| --- | --- |
| int duckWeight = 15;  int personWeight = 15;  if (duckWeight > personWeight)  {  Console.WriteLine("Clearly not a witch!");  }  else if (personWeight > duckWeight)  {  Console.WriteLine("She's a Witch!");  }  else  {  Console.WriteLine("They weigh the same!");  } | Screen shot  Note the subtle change in logic in the first IF statement with the removal of the ‘=’.  Now the first two tests are for greater than. As the weights are the same, it checks both IF statements, which return false, and therefore executes the ELSE statement. |

### Logic errors in IF statements

|  |  |
| --- | --- |
| int duckWeight = 15;  if (duckWeight > 5)  {  Console.WriteLine("The duck is a small duck!");  }  else if (duckWeight > 10)  {  Console.WriteLine("The duck is a normal size!");  }  else if (duckWeight >= 15)  {  Console.WriteLine("The duck is a large duck!");  } | When using logic statements you need to be careful about the order you carry out your IF statement conditions.  Here, you can see that the duckWeight is 15. By looking at the IF statement results, we can see that the duck SHOULD be reported as being a small duck. (yellow highlight)  However, the first IF condition checks for duckWeight being greater than 5. As 15 > 5 = True, it will use the FIRST IF statement, and then skip the rest. |

### Nesting IF statements

It is possible to **nest** IF statements.

|  |  |
| --- | --- |
| if (*Case A is true*)  {  #This code will execute  if (*Case B is true*)  {  #This nested IF code will execute  }  }  else  {  Console.WriteLine("They weigh the same!");  } | Note the change in logic.  If Case A is false, the ELSE statement will execute.  If Case A is true then any code in that IF statement will run.  This will check the NESTED if statement.  Case B will ONLY be checked if Case A is true.  Case B IF statement code will ONLY execute if **both** Case A and Case B are true. |

# Iteration

## FOR Loops/Count Controlled

**Pseudocode**

|  |  |
| --- | --- |
| for i-0 to 7  print (“Hello”)  next i | Will print “Hello” 8 times (0-7 inclusive). Note that the count starts at 0. |

**C#**

Count controlled loops will need a variable to be incremented or decremented to define when the loop ends.

The structure of the FOR Loop is:

for (count ; controll; counter)

The integer you use to ‘control’ iterations with can either be created inside the loop itself, or use a variable that already exists within the program.

The counter operation always executes at the end of each iteration.

|  |  |  |
| --- | --- | --- |
| **Control created** | **Benefits** | **Drawbacks** |
| In the loop | * + - Easy to see where the loop terminates | * + - Loop will always have fixed iterations |
| Using pre-existing variable | * + - Allows you to link the loop to other variables in your program     - Allows varying number of iterations to take place | * + - Can lead to run time issues |

### Counting forwards

|  |  |
| --- | --- |
| for (int count = 1; i <= 5; i++)  {  Console.WriteLine(i);  } | Here is an example of the FOR loop in action.  Our count is defined within the loop itself. We always know this loop will start a 1 and then iterate until the count = 5. When the count ‘i’ reaches 6, the loop will exit as the condition becomes false.  Screen shot |
| int countTo = 0;  Console.Write("What number do you want to count to?: ");  countTo = int.Parse(Console.ReadLine());  for (int i = 1; i <= countTo; i++)  {  Console.WriteLine(i);  } | Here we are using a variable within the condition to vary the number of iterations.  This means the loop has more functionality:  Run 1:  Screen shot  Run 2:  Screenshot  However, what would happen if the user entered ‘0’ for a number to count to? |

### Counting backwards

|  |  |
| --- | --- |
| int countTo = 0;  Console.Write("What number do you want to count from?: ");  countFrom = int.Parse(Console.ReadLine());  for (int i = countFrom; i >= 0; i--)  {  Console.WriteLine(i);  } | You can also decrease your counter to reach a certain value.  Screesnshot |

## Condition controlled loops

**Pseudocode**

|  |  |
| --- | --- |
| while answer!=”computer”  answer=input(“What is the password?”)  endwhile  do  answer=input(“What is the password?”)  until answer == “computer” | The while loop will keep looping while its condition is True. |

### WHILE Loops

Where we do not know when the end of the loop may occur, we can use a condition controlled loop. The syntax is essentially: while (condition = TRUE) – run some code.

We can replicate a FOR loop we used earlier with a condition controlled loop.

|  |  |  |
| --- | --- | --- |
| int countTo = 0;  int startValue = 0;  Console.Write("What number do you want to count to?: ");  countTo = int.Parse(Console.ReadLine());  while (startValue <= countTo)  {  Console.WriteLine(startValue);  startValue++;  } | As you can see, we need some extra things to similar a FOR loop with a condition control.   1. We need a value to compare against so that we can get a BOOLEAN result 2. We need to increase the ‘counter’ still by using the ‘++’ operator. | |
| Screenshot | However, as you can see – the results look identical. The question is: What is more efficient? | |
| string myName = "";  const string storedName = "Ceredig";  while (myName != storedName)  {  Console.Write("Guess is my name? ");  myName = Console.ReadLine();  }  Console.WriteLine("This is my name!"); | | Screen shot  Here we repeat the user to enter data until it is correct/matches other data, and then allow the program to continue. |

## DO WHILE loops

DO WHILE loops are fairly similar to WHILE loops, in so far as they execute code until a certain condition is met. However, a DO WHILE loop **ALWAYS** runs the first iteration of the loop before checking the condition.

|  |  |  |
| --- | --- | --- |
| bool quit = true;  do  {  Console.Write("Please enter a menu choice: ");  }  while (quit == false); | | Here we see that the ‘quit’ Boolean data type (highlighted yellow) is **the same** in each case.  Our check condition is whether or not the variable ‘quit’ is equal to false. If quit is false, then the loop will carry on.  Because quit is set to **true**, quit == false would equate to **false** and therefore the loop would exit.  With a **WHILE** loop, because we check the condition first, the loop would not run…  However, with a DO WHILE loop, because we check the condition AFTER the ‘do’ part, we would get output from the program. |
| bool quit = true;  while (quit == false)  {  Console.Write("Please enter a menu choice: ");  } | |
| **DO WHILE RESULT:** | Screen shot | |
| **WHILE LOOP RESULT:** | Screen shot | |

### Infinite loops

It is often easy to create infinite loops without meaning to! These errors will compile, and are Run Time errors (i.e. only apparent when the program runs). Most often these are down to logic errors.

|  |  |
| --- | --- |
| int numberOfParrots = 1;  while (numberOfParrots <= 1)  {  Console.WriteLine("This parrot is dead!");  numberOfParrots--;  } | |
| Here the user is trying to get the program to print out the number of dead parrots. However, a LOGIC error means that this will loop infinitely! | Screen shot |

# Using break and continue key words

## The use of the break in loops

There is a keyword, break, which when used, will automatically jump out of a code construct that you are using. The use of break in programming can make the flow of control **very hard to follow** and is considered bad practise by many coders. Where possible, it is usually better to try and manage the exit from LOOPS, IF statements, etc. through other means (e.g. meeting conditions) rather than the use of break.

## The use of the break in SWITCH / Case Select

We **must** use the ‘break’ key word when using SWITCH (Case Select) statements. This is part of the formal syntax within C#.

## The use of the continue keyword

There is a keyword, continue, which when used, will automatically return to the top of the construct, and trigger the next iteration – e.g. any counters will update, and a new iteration starts. Again this may lead to confusion in the flow of control between constructs. However, it can lead to efficiencies in programming, by avoiding execution of code.

### Examples of break and continue in loops

|  |  |
| --- | --- |
| int number = 13;  int stop = 9;  for (int i =0; i <= number; i++)  {  if (i % 2 == 0)  {  continue;  }  else if (i == stop)  {  break;  }  else  {  Console.WriteLine("This parrot is number " + i);  } | |
| Here we use the two key words for two purposes.  continueis used to return to the start of the FOR loop if a number is even. In this case, if continuewas not used, the program would check the rest of the IF statement before looping and print out “This parrot is number X”.  breakis used to exit the FOR loop once we reach 9 (and parrot 9 will not be printed). | Screen shot  **NB:** We could **avoid** using break here by replacing it with: number = 13; |

# The use of basic string manipulation

**Pseudocode**

|  |  |
| --- | --- |
| stringname.length | This gets the length of a string. |
| stringname.subString(startingPosition, numberOfCharacters) | This gets a substring but the string will start at the 0th character. |
| stringname.upper  stringname.lower | This converts the case of the string to either upper or lower case. |
| ASC(character)  CHR(asciinumber) | This converts to and from ASCII. |
| someText="Computer Science"  print(someText.length)  print(someText.substring(3,3))  16  put | Here length of the variable is printed along with the 3 characters 3 character in for 3 characters. |

**C#:**

|  |  |
| --- | --- |
| string messageA = "This part of the ";  string messageB = "message is broken in ";  string messageC = "to three separate strings!"; | Here we have a message stored as 3 separate strings. |
| string messageA = "This part of the ";  Console.WriteLine(messageA.Length); | We use String.Length to count the individual characters (including spaces!) within the string.  Screen shot |
| string messageA = "This part of the ";  Console.WriteLine(messageA);  Console.WriteLine(messageA.ToUpper()); | We can use the .ToUppper method to change the case of a string.  Screen shot |

|  |  |
| --- | --- |
| string upperString = "THIS IS ALL IN CAPITALS";  Console.WriteLine(upperString);  Console.WriteLine(upperString.ToLower()); | We can use the .ToLower method to change the case of a string.  Screen shot |
| string messageA = "This part of the ";  string messageB = "message is broken in ";  string messageC = "to three separate strings!";  string completeMessage = null;  completeMessage += messageA;  completeMessage += messageB;  completeMessage += messageC;  Console.WriteLine(completeMessage); | You can append strings together, either into a separate variable.  Screen shot  += appends one variable to another keeping the original value that was there. |
| string messageA = "This part of the ";  string messageB = "message is broken in ";  string messageC = "to three separate strings!";  Console.WriteLine(messageA + messageB + messageC); | You could concatenate directly when printing to screen. This will not store the final result however.  Screen shot |

### Placeholders

|  |  |
| --- | --- |
| You can use references for values within strings, to make formatting easier | |
| int duckWeight = 34;  int suspectWeight = 35;  Console.WriteLine("The duck weighs " + duckWeight + ". However the suspect weighs " + suspectWeight + "!");  Console.WriteLine("The duck weighs {0}. However the suspect weighs {1}!", duckWeight, suspectWeight); | |
| Screen shot | You can see that both lines of code produce the same output, but arguably, using the second method is easier to do, and clearer to follow. |

### ‘Contains’ within a string

|  |  |
| --- | --- |
| string messageA = "This part of the ";  bool result;  result = messageA.Contains("This");  Console.WriteLine(result);  result = messageA.Contains("this");  Console.WriteLine(result); | You can also check to see if strings contain certain values/strings.  This example checks for the word “This” in the string “This part of the “ variable.  Note than comparisons are case sensitive.  Screen shot |

### ASCII values

|  |  |
| --- | --- |
| string messageA = "This part of the ";  foreach (char c in messageA)  {  int asciiValue = c;  Console.WriteLine(asciiValue);  } | You can return ASCII values of characters.  Screen shot |

|  |  |
| --- | --- |
| int value;  Console.Write("Enter your value: ");  value = int.Parse(Console.ReadLine());  char character = (char)value;  Console.WriteLine(character); | To find the character an integer refers to, you can use the following.  This gives the result:  Screen shot |

## Substrings

### Strings cutting from the left

|  |  |
| --- | --- |
| string messageA = "This part of the ";  string subMessage = null;  subMessage = messageA.Substring(0, 6); | This cuts from the left, starting at index 0, and then taking 6 characters, and saving it as the new string.  Screen shot |

### String cutting from the right

|  |  |
| --- | --- |
| string messageA = "";  string subMessage = null;  Console.Write("Please enter a string: ");  messageA = Console.ReadLine();  subMessage = messageA.Substring(messageA.Length-6,6);  Console.WriteLine(subMessage); | Cutting from the right requires us to know how long the string is. If you know this already, this is easier. If not, then you need to work out the string length first using the String.Length method.  Then you can count back the number of characters you want to cut! Screen shot |

# File handling

File handling requires two classes to be used. StreamReader and StreamWriter. Each class needs to be instantiated during execution.

## StreamReader

**Pseudocode**

|  |  |
| --- | --- |
| myFile = **openRead**(“sample.txt”)  x = myFile.**readLine()**  myFile.**close()** | To open a file to read from openRead is used and readLine to return a line of text from the file. |

**C#**

|  |  |
| --- | --- |
| string textFileLine = "";    StreamReader myFileReader = new StreamReader("C:/TextFile/MyTextFile.txt");  textFileLine = myFileReader.ReadLine();  Console.WriteLine(textFileLine);  myNewReader.Close(); | |
| The above code creates the link to read from the file using the StreamReader. Once this is created, we can then read from the file.  This example reads the first line from the file and then save it as the variable ‘textFileLine’. It then prints the variable to screen, as show on the right. | Screen shot |

## The use of close

If we do not close the file after a StreamReader or StreamWriter then the link to the file stays open, and the computer ‘remembers’ where it was up to with that file.

|  |
| --- |
| string textFileLine = "";  StreamReader myFileReader = new StreamReader("C:/TextFile/MyTextFile.txt");  textFileLine = myFileReader.ReadLine();  Console.WriteLine(textFileLine);  // Here is some other code we do in the mean time.  Console.WriteLine("My other code has just run and now I want the first line of the file again!");  textFileLine = myFileReader.ReadLine();  Console.WriteLine(textFileLine); |
| Here we read the first line, and the code goes off and does something else. If we then came back and wanted to read the first line of the file again later, we would find that it actually now reads the SECOND line of the file.  Screen shot |

To tell C# that we have finished using the StreamReader, we use the **‘.close()**’ method, which **terminates** the StreamReader that is currently in use. This means that we cannot use it again and need to create a new one.

**It is better to try and read all of the file that you will need in one “operation” and then close the file. You can save the file into an array or a string for later, and use that variable within the program. This is much more effective!**

## Reading a file line by line

**Pseudocode**

|  |  |
| --- | --- |
| myFile = openRead(“sample.txt”)  while NOT myFile.endOfFile()  print(myFile.readLine())  endwhile  myFile.close() | readLine is used to return a line of text from the file. endOfFile()is used to determine the end of the file. The example will print out the contents of sample.txt |

**C#**

|  |  |
| --- | --- |
| string textFileLine = "";  bool fileEmpty = false;    StreamReader myNewReader = new StreamReader("C:/TextFile/MyTextFile.txt");  while(fileEmpty == false)  {  textFileLine = myNewReader.ReadLine();  if (textFileLine != null)  {  Console.WriteLine(textFileLine);  }  else  {  fileEmpty = true;  }  }  myNewReader.Close(); | |
| Here we set up two variables. One holds the value that will be read from the file. The other is a Boolean to flag when file is empty.  We use a WHILE loop here, as we do not always know how many lines there may be in the file. If we do – we can use a FOR loop.  We then read a line from the file, and use the IF statement to check what happens. If the line is NOT NULL (i.e. contains text) it will print that line out. Otherwise, it changes the flag to ‘true’, indicating the file is empty and then the WHILE loop will not iterate again. | Screen shot  **Note: if you have a file that has empty lines in between text, this method will not work for you!** |

## Reading a whole file at once

|  |  |
| --- | --- |
| textFileLine = File.ReadAllText("C:/TextFile/MyTextFile.txt");  Console.WriteLine(textFileLine); | |
| This reads ALL of the file content into a single variable.  Note that here it does not matter if there are line breaks in between lines of text. | Screen shot |

## Write to a file (overwrite)

**Pseudocode**

|  |  |
| --- | --- |
| myFile = **openWrite**(“sample.txt”)  myFile.**writeLine**(“Hello World”)  myFile.close() | To open a file to write to, openWrite is used and writeLine to add a line of text to the file. In the example, Hello world is made the contents of sample.txt (any previous contents are overwritten). |

**C#**

|  |  |
| --- | --- |
| StreamWriter myFileWriter = new StreamWriter("C:/TextFile/MyTextFile.txt");  myFileWriter.WriteLine("My new line of text!");    myFileWriter.Close(); | |
| By default, C# will OVERWRITE to a file, if you use the above code.  The file I have used is the SAME file that had the Monty Python quotes in. Note now, that the content has been **overwritten**. | Screen shot |

## Write to a file (append)

**C#**

|  |  |
| --- | --- |
| StreamWriter myFileWriter = new StreamWriter("C:/TextFile/MyTextFile.txt", true);  myFileWriter.WriteLine("My new line of text!");  myFileWriter.Close(); | |
| The use of the keyword true as an option within the StreamWriter instantiation allows us to now ADD extra lines to a file that already exists.  As we can see on the right, we already have text within the text file, and we have now added “My new line of text!” to the end of the file. | Screen shot |

# The use of arrays

Arrays can be single, or multi-dimensional. However, in C# arrays are static, in so far as once you have said that an array is say 5 elements in size, you cannot add a 6th element.

**Pseudocode**

|  |  |
| --- | --- |
| array names[5]  names[0]="Ahmad"  names[1]="Ben"  names[2]="Catherine"  names[3]="Dana"  names[4]="Elijah"  print(names[3])  array board[8,8]  board[0,0]="rook” | Arrays will be 0 based and declared with the keyword *array*.  Example of a 2D array |

## Declaring an array

**C#**

|  |  |
| --- | --- |
| int[] peopleAges = new int [5];  string[,] myMapGrid = new string[5, 5]; | Here we have created an empty 1D and a 2D array in C#. The syntax is not quite the same as creating a basic data type variable. |

## Initialising an array with data items

|  |
| --- |
| int[] peopleAges = new int[] {5,4,3,2,1};  int[] peopleAges2 = { 5, 4, 3, 2, 1 };  string[,] myMapGrid = new string[,] {{"A1", "A2", "A3"}, {"B1", "B2", "B3"}, {"C1", "C2", "C3"}};  string[,] myMapGrid2 = { { "A1", "A2", "A3" }, { "B1", "B2", "B3" }, { "C1", "C2", "C3" } }; |
| Above shows how we can both initialise a variable to be of type array and **also** add data items in when initialised. There are two ways of doing this – one slightly longer than the other. Both work! The first way is perhaps viewed as more “formal” as it clearly shows the type of array being declared again in the assignment. |

## Assigning data items to an array after initialisation

|  |  |
| --- | --- |
| int[] peopleAges = new int[5];  peopleAges[0] = 31;  peopleAges[1] = 24;  peopleAges[2] = 12;  peopleAges[3] = 68;  peopleAges[4] = 44; | After we have initialised the array, we can add data items, of that data type, into the array.  **All of the data items within an array must be of the same type.**  Here we have added 5 ages to our array.  *Remember that an array index starts at [0]!* |
| string[,] namesAndAges = new string[3, 2];  namesAndAges[0, 0] = "Jill";  namesAndAges[0, 1] = "23";  namesAndAges[1, 0] = "Rashpal";  namesAndAges[1, 1] = "43";  namesAndAges[2, 0] = "Leoung";  namesAndAges[2, 1] = "55"; | With a 2D array, we need to make sure that:   1. All the data items are of the same type – therefore ages here are stored as **strings** 2. We remember the dimensions of the array when assigning data! |
| **Trying to assign a variable outside of the array boundary**  string[,] namesAndAges = new string[3, 2];  namesAndAges[0, 0] = "Jill";  namesAndAges[0, 3] = "23"; | The code on the left shows us that we have tried to assign ‘23’ to [0,3] in the array.  The array’s bounds are [3,2] and therefore [0,3] does not exist. It generates the following **run time** error shown below.  **These errors do not show in the IDE**, and will simply result in the program crashing once executed as shown below. |
| **Array out of bounds run time error:**  Screen shot | |

## Sorting arrays

|  |  |
| --- | --- |
| Sorting lists is usually useful and you can do this by using the Array.Sort() method. However, once sorted, the array will remain in a sorted state | |
| int[] peopleAges = {34,65,12,43,65,76,85,34,11};  Console.WriteLine("Unsorted array items:");  foreach (int age in peopleAges)  {  Console.WriteLine(age);  }  //Sort the array using Array.Sort()  Array.Sort(peopleAges);  Console.WriteLine("Sorted array items");    foreach (int age in peopleAges)  {  Console.WriteLine(age);  } | Screen shot |
| You can also use the sort method on Strings as shown below | |
| string[] peopleNames = {"Eric", "Jessica", "Abdul", "Chia", "Chen", "Douglas"};  //Sort the array using Array.Sort()  Array.Sort(peopleNames);  Console.WriteLine("Sorted array items");  foreach (string name in peopleNames)  {  Console.WriteLine(name);  } | Screen shot |

## Reverse sorting arrays

|  |  |
| --- | --- |
| Sorting arrays in reverse order can be achieved by a two step process.   1. Sort in order 2. Reverse the array | |
| int[] peopleAges = {34,65,12,43,65,76,85,34,11};  //Sort the array using Array.Sort()  Array.Sort(peopleAges);  //Reverse the order using Array.Reverse()  Array.Reverse(peopleAges);  Console.WriteLine("Reverse sorted array items");  foreach (int age in peopleAges)  {  Console.WriteLine(age);  } | Screen shot |
| **The same works for strings in reverse.** | |
| string[] peopleNames = {"Eric", "Jessica", "Abdul", "Chia", "Chen", "Douglas"};  //Sort the array using Array.Sort()  Array.Sort(peopleNames);  //Reverse the order using Array.Reverse()  Array.Reverse(peopleNames);  Console.WriteLine("Reverse sorted array items");  foreach (string name in peopleNames)  {  Console.WriteLine(name);  } | Screen shot |

## Printing arrays

You can print from arrays in two different ways.

1. Using a FOR loop
   1. Used if you already know before code execution, how long each array is
   2. You want to cycle through an array in a specific order (e.g. a 2D array, or in reverse)
2. Using a FOR EACH loop
   1. Easier syntax
   2. Will only print from “top to bottom”
   3. Will do 2D arrays, but only in a specific order

### Using a FOR loop

|  |
| --- |
| **1-dimensional Array**  string[] mySweetList = {"Refreshers", "Smarties", "Ferrero Rocher", "Starburst" };  foreach (string sweet in mySweetList)  {  Console.WriteLine("You own: {0}", sweet);  } |
| Screen shot |
| **2-dimensional Array**    string[,] mySweetList = {  { "Refreshers", "Smarties", "Ferrero Rocher", "Starburst" },  { "Gold bar", "Rocky Road", "Fruit Pastilles", "Moam" }  };  foreach (string sweet in mySweetList)  {  Console.WriteLine("You own: {0}", sweet);  } |
| Screen shot |

## Using a FOR EACH loop

|  |
| --- |
| **1-dimensional Array**  string[] mySweetList = {"Refreshers", "Smarties", "Ferrero Rocher", "Starburst" };  for (int i = 0; i < 4; i++)  {  Console.WriteLine("You own: {0}", mySweetList[i]);  } |
| Screen shot |
| **2-dimensional Array**  string[,] mySweetList = {  { "Refreshers","Smarties","Ferrero Rocher","Starburst" },  { "Gold bar","Rocky Road","Fruit Pastilles","Moam" }  };  for (int z = 0; z < 4; z++)  {  for (int i = 0; i < 2; i++)  {  Console.WriteLine("You own: {0}", mySweetList[i,z]);  {  } |
| Screen shot |

# The use of records to store data

**Pseudocode**

|  |  |
| --- | --- |
| array myClass[5,4]  myClass [0,0] = "Name"  myClass [0,1] = "Class"  myClass [0,2] = "Age"  myClass [0,3] = "FavColour" | Records can be written as 2D array.  The first ‘row’ would contain the field names. |
| myClass [1,0] = “Sunita”  myClass [1,1] = “3B”  myClass [1,2] = “15”  myClass [1,3] = “Green” | We can then use each ‘row’ underneath to contain a record. |

**This is a start to storing the following records as Pseudocode:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Class** | **Age** | **FavColour** |
| Sunita | 3B | 15 | Green |
| Gerry | 3B | 15 | Blue |
| Rashford | 3A | 16 | Yellow |
| Bill | 3D | 14 | Mauve |

**C#**

We already know how to set up a 2D array, and to add data to an array location.

|  |
| --- |
| string[,] myClass = new string[5,4];  for (int y = 0; y < 4; y++)  {  for (int x = 0; x < 4; x++)  {  Console.WriteLine("Enter value for [{0},{1}]: ", y, x);  myClass[x, y] = Console.ReadLine();  }  } |

|  |  |
| --- | --- |
| As you can see, the FOR loop navigates the array and allows data entry for each data location within the array. | Screen shot |

## Printing a record structure

We can use a FOR loop to print these out in to a format that “looks like” a table through use of inserting TAB characters within the output statements.

|  |  |
| --- | --- |
| string[,] myClass = {  {"Name", "Class", "Age", "fColour"},  {"Sunita", "3B", "15", "Green"},  {"Gerry", "3B", "15", "Blue"},  {"Gordon", "3A", "16", "Yellow"},  {"Bill", "3D", "14", "Mauve"},  };  //Use a nested FOR loop to cycle through the array  for (int y = 0; y < 4; y++)  {  for (int x = 0; x < 4; x++)  {  //Print out each data item in the array using the "\t"  //to add a TAB in between each item  Console.Write(myClass[y,x] + "\t");  }  //We had to use the Console.Write() to avoid adding in 'returns' after each  //line, therefore we need to add a line break in the outer loop.  Console.WriteLine();  } | |
| As you can see, this gives us a nicely formatted table-styled print out. However, this does depend on using data items that are **under** a certain length, otherwise the tab formatting does not always work as nicely as we want it to! | Screen shot |

# How to use sub programs

Sub Programs are called **functions** within C#. **Functions** can be used to separate commonly used code into a separate code block. This can help us create easy to follow and efficient programs.

**Methods** are special functions that are used within Object Orientated Programming.

There are **two types of function.**

1. Does not return a value

*Beware! In the pseudocode guide this is named procedure*

1. Does return a value

*Beware! In the pseudocode guide this is named function*

In C# each function use has a slightly different syntax.

We can also make functions more complex by passing variables into them. When we define a function, we show the **parameters** that will be used. When we run the function, the variables passed into the function are called **arguments**.

**Methods** are created **outside** of the static void Main(string[] args) – i.e. the main method.

static void Main(string[] args)

{

//This is the main body of the program

}

//This function is created outside of the Main

static void printHello()

{

Console.WriteLine("Hello!");

}

**Beware!** when creating functions in C#, as it is easy to forget the { }, delete one by accident etc. It is good practice to keep things nicely aligned so that you can see where functions start and end!

**Pseudocode**

|  |  |
| --- | --- |
| function triple(number)  return number\*3  endfunction  y=triple(7)  procedure greeting(name)  print(“hello”+name)  endprocedure  greeting(“Hamish”) | Here we define a function which uses the **parameter** (number).  When we call the function ‘7’ becomes the **argument** for that function.  The calculation is then performed and the function is ended.  Here we can use the argument “Hamish” for the procedure called ‘greeting ‘. This procedure will print a string including the argument. |

## Creating a static function

Void functions **do not return a value.**

|  |  |
| --- | --- |
| static void printHello()  {  Console.WriteLine("Hello!");  } | Here we have defined a function that does not return a value by using the keywords static void.  The name of the function is highlighted in green – here it is printHello .  The two brackets ‘()’ show that no parameters are needed to run this function. |

## Creating a non-void function

Non-void functions need to say what **variable type** they return.

|  |  |
| --- | --- |
| static int myAge()  {  return 35;  }  static string myMessage()  {  return "She turned me into a newt!";  } | Here we have defined a function that does return a value.  The first function returns an integer, the second returns a string.  Note that the keyword void has changed to the type of variable you want to return, e.g. here we use int or string to show the function will return a variable of that type. |

## Functions with parameters

### Single Parameter functions

|  |  |
| --- | --- |
| static double calcVAT (double price)  {  return price \* 1.2;  } | Notice now we have included the data type and identifier that we are passing to the function.  Here we use a double to use in the method for calculating a VAT total. |

### Multiple parameter functions

|  |
| --- |
| static double calcVAT(double price, double vatRate)  {  return price \* vatRate;  } |
| Here we declare that the function needs has two parameters. Both of these parameters are double. We then return the calculations as a double. |

## Using functions within a program

Functions are great for sectioning out code, as discussed. We have seen that some functions may return values. However, unless we **use** the values a function returns, we lose that data.

### Using returned values

|  |  |
| --- | --- |
| //Variables  int number;    //Get the user number  Console.WriteLine("What number shall I square? ");  number = int.Parse(Console.ReadLine());  //Use the function to double the number  squareNumber(number);  //Print the answer  Console.WriteLine(number);  //Function to multiply a number by itself  static int squareNumber (int number)  {  return number\*number;  } | Here we have created a function to square a number (number \* number).  However these is an issue with the programs output as it stands, which is shown here:  Screen shot  The issue lies with the red highlighting. We send the argument to the function – which is the number the user typed in. The function correctly works out number\*number. However, **we do not store the value that the function returns.** |
| **Original**  //Use the function to double the number  squareNumber(number);  **New**  //Use the function to double the number  number = squareNumber(number); | With the modified code, we can now assign the returned value to a variable (here we just overwrite the value of number).  It is important to remember to do this, otherwise you may lose data, or store incorrect data etc. |

|  |  |
| --- | --- |
| //Variables  int number;    //Get the user number  Console.WriteLine("What number shall I square? ");  number = int.Parse(Console.ReadLine());  //Use the function to double the number  number = squareNumber(number);  //Print the answer  Console.WriteLine(number);  //Function to multiply a number by itself  static int squareNumber (int number)  {  return number\*number;  } | |
| Screen shot | This is the result of the updated program code, and as you can see, it now correctly squares the number, stores it as ‘number’ and then prints it to the screen. |

### Where should I use functions

Functions are for repeated code blocks that may be used many times. Common examples could be:

* Calculations
* Validation
* Conversions
* Recursion

# Casting

As everything from the keyboard in C# is viewed as a string by the compiler, we need to **cast** our input into the correct data type. Some languages do this automatically (such as Python)! Sadly, in C# we have to do this manually.

**Pseudocode**

|  |  |
| --- | --- |
| str(3) returns "3"  int("3") returns 3  float("3.14") returns 3.14 | Variables can be typecast using the int str and float functions. |

**C#**

|  |  |
| --- | --- |
| //Variables  string userEntry = "54";  int castNumber;  double dblNumber;  string backToString;  //Various casting options  castNumber = int.Parse(userEntry);  dblNumber = double.Parse(userEntry);  backToString = Convert.ToString(dblNumber);  //Printing the output  Console.WriteLine(userEntry);  Console.WriteLine(castNumber);  Console.WriteLine(dblNumber);  Console.WriteLine(backToString); | Here we have option to cast from strings to numerical data types and from numerical data types back to a string.  The output is shown below:  Screen shot  As we can see it makes no different to the printing to screen – but it **does** allow us to now carry out calculations on the numeric data types such as multiplication and division. |

# Random numbers

It is possible to generate a random number in C# for use within a program.

|  |  |
| --- | --- |
| Random randomGenerator = new Random();  int myRandomNumber = randomGenerator.Next(0, 100);  Console.WriteLine(myRandomNumber); | As the radom generator is method within a pre-defined class, we have to create an instance of this method first.  Next we can assign a random number to a variable, using the name of the instance we have created (here it is randomGenerator), the .Next() call and then a range (shown in the brackets.  The last line prints this variable to screen. |

We can use the same random generator for any random number, and simply change the range of the arguments we pass through to the Random method we use.

|  |  |
| --- | --- |
| int myRandomNumber;  Random randomGenerator = new Random();  myRandomNumber = randomGenerator.Next(-40, 0);  Console.WriteLine(myRandomNumber);  myRandomNumber = randomGenerator.Next(0, 50);  Console.WriteLine(myRandomNumber);  myRandomNumber = randomGenerator.Next(345, 554);  Console.WriteLine(myRandomNumber); | Here we generate 3 random numbers using the same instantiation randomGenerator.  The three numbers we generate are between:   * -40 to 0 * 0 to 50 * 345 to 554   Screen shot |

**Combinations of techniques**

Following is an example that uses the majority of techniques within the syntax guide. It combines a DO-WHILE, SWTICH, Random, bool, string, int, functions without parameters, functions with parameters and IF-ELSE statements.

|  |
| --- |
| int witchWeight = 0, duckWeight = 0;  bool judgement = false;  string menuchoice = null;  discussion();  do  {  Console.WriteLine("Witch assessment menu: ");  Console.WriteLine("1 - weigh a witch ");  Console.WriteLine("2 - weigh a duck ");  Console.WriteLine("3 - pass judgement on the witch");  Console.Write("Please enter a menu choice: ");  menuchoice = Console.ReadLine();  switch (menuchoice)  {  case "1":  witchWeight = generateWeight();  Console.WriteLine("Your witch weighs: {0}", witchWeight);  break;  case "2":  duckWeight = generateWeight();  Console.WriteLine("Your duck weighs: {0}", duckWeight);  break;  case "3":  if (witchWeight == 0)  {  Console.WriteLine("It appears that you haven't weighed your witch yet!");  }  else if (duckWeight == 0)  {  Console.WriteLine("It appears that you haven't weighed your duck yet!");  }  else  {  judgement = weightCheck(witchWeight, duckWeight);  }  break;  default:  break;  }  } while (judgement == false);  }      static void discussion()  {  Console.WriteLine("So, logically, If she weighs the same as a duck...");  Console.WriteLine("She's made of wood!");  Console.WriteLine("And therefore?");  Console.WriteLine("A witch!");  }  static bool weightCheck(int witch, int duck)  {  Console.WriteLine("We have compared the witch and the duck!");  if (witch >= duck)  {  Console.WriteLine("A witch! Burn the witch!");  return true;  }  else  {  Console.WriteLine("Not a witch... try another one!");  return false;  }  static int generateWeight()  {  int newWeight;  Random generator = new Random();  newWeight = generator.Next(1, 100);    return newWeight;  } |
| Screen shotHere is a sample of some of the output from the program.  As you can see, we now have a great program the weighs a suspected witch, weighs a duck and then compares which is heavier.  If the duck is heavier, then the witch is condemned.  If the duck is lighter, the witch survives!  If you do not weigh either the witch or the duck, it does not allow you to progress to ‘passing judgement’ and tells you as such. |

# Recursion

Recursion is when a function calls itself during its execution:

|  |  |
| --- | --- |
| static void Main(string[] args)  {  countdown(10);  }  static void countdown(int tMinus)  {  if (tMinus ==0)  {  Console.WriteLine("\*\*\* Blast Off!! \*\*\*");  }  else  {  Console.WriteLine(tMinus);  countdown(tMinus-1);  }  } | |
| Screen shot | The flow of control starts with the Main method. We have one line of code – countdown(10). This calls the function countdown, using the argument of 10 (i.e. we want the countdown to start at 10).  The function ‘countdown’ will print a message if the variable is equal to 0 (in this case ‘blast off’).  However, if it isn’t equal to 0 then it prints the variable tMinus, and then call itself using the variable-1.  Effectively this means that you would have 10 of the functions running, all ‘nested’ inside each other. |

# Scope

Variables have scope. Scope is the ‘visibility’ of a variable to other parts of the program. Scope is a very powerful thing, and can be used effectively to manage memory and stop mistaken access to variables.

## Local variables

**Example**

|  |  |
| --- | --- |
| static void Main(string[] args)  {  string name;  Console.Write("Enter your name: ");  name = Console.ReadLine();    printName();  }  static void printName ()  {  Console.WriteLine("Your name is {0}", name);  } | The code on the left will not compile.  We have a variable, ‘name’ as a string, which allows the user to enter a string into the program.  We want to use this variable later on in a function to print out a message. However, the **scope** of the variable is **local** to that function. Therefore only code that uses the ‘name’ variable within the Main method will be able to ‘see’ the variable.  The bottom yellow line will not work, as name cannot be ‘seen’ by this method. |

**Example 2:**

|  |  |
| --- | --- |
| static void Main(string[] args)  {  getName();  Console.WriteLine("Your name is ” + myName);  }  static void getName ()  {  string myName = Console.ReadLine();  } | The code on the left will also not compile.  The variable ‘myName’ is created within the function ‘getName’. However, it can firstly **only** be seen within this function.  Secondly, after the function runs, the variable is **destroyed** and ceases to exist. Therefore it cannot be used within the Main method. |

## Avoiding scope issues

The best way to avoid scope issues is to declare any variables you will need throughout your program within the main method. You can then use them within functions you create as **arguments** that you pass into the function (see the section on functions). However, you may then also need to **return** the results of the function to overwrite these variables with a new value.

The advantage of doing it this way is that you have to physically pass and return variables; this means you can keep a close watch on what is going on!

Another way could be to use global variables.

### Global variables

Global variables in C# are more complex to set up. They can also be viewed as “slack” lack of programming. If you set every variable as global – you can get access to them anywhere within the program. Whilst this may seem a good idea, you can potentially end up overwriting variables by mistake.

|  |  |
| --- | --- |
| public static class MyGlobals  {  public const double vatRate = 1.2;  public static double itemCost;  }  class Program  {  static void Main(string[] args)  {  Console.Write("What is the cost of the item without VAT: ");  MyGlobals.itemCost = double.Parse(Console.ReadLine());  Console.WriteLine("The true cost of your item is: {0}", MyGlobals.vatRate\*MyGlobals.itemCost);  }  } | |
| In the code above, we have created a public class (highlighted in green). This contains our ‘global variables’. Notice that each variable has extra key words before its declaration:   * public const * public static   The keyword public shows us this is a globally accessible variable.  When trying to access the variables, note we now have to use the MyGlobals. before the name of the variable we want to use. | Screen shot |

# The use of SQL to search for data

**Pseudocode**

|  |  |
| --- | --- |
| SELECT (including nested SELECTs)  FROM (including use of \* wildcard)  WHERE  LIKE (with % used as a wildcard)  AND  OR |  |

**SQL**

This example assumes there is a database created called ‘Customers’ with columns called:

* CustomerID
* CustomerName
* ContactName
* Address
* City
* Country.

|  |  |
| --- | --- |
| SELECT \* FROM Customers | This selects everything (\*) from the Customers database. |
| SELECT *ContactName*,*Address* FROM *Customers* WHERE *ContactName = ‘Mr Jones’*; | This selects the ContactName and Address columns from the Customers table and then specifically looks for a “Mr Jones” in the ContactName field. |
| SELECT *ContactName*,*Address* FROM *Customers* WHERE *CustomerID = 3*; | This selects the ContactName and Address columns from the Customers table where the CustomerID field equals 3. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SELECT *ContactName*,*Address* FROM *Customer*  WHERE FirstName LIKE J\*; | This selects the ContactName and Address columns from the Customers table and then looks for a something LIKE J\* in the FirstName field. This is a more open search and will return any value that is like the pattern provided.  Here It would find all of the results that started with the letter ‘J’  You can also use these operators:   |  |  | | --- | --- | | = | Equal | | <> | Not equal (!= sometimes) | | > | Greater than | | < | Less than | | >= | Greater than or equal | | <= | Less than or equal | | BETWEEN | Between an inclusive range | | LIKE | Searcher for a pattern | | IN | Specify multiple values | |
| SELECT \* FROM Customers  WHERE Country = ‘England’  AND (Type = ‘Witch’ OR Type = ‘Duck); | You can also use Boolean operators (AND, OR) to refine a search and these can be combined using brackets. |

**OCR Resources**: *the small print*OCR’s resources are provided to support the teaching of OCR specifications, but in no way constitute an endorsed teaching method that is required by the Board, and the decision to use them lies with the individual teacher. Whilst every effort is made to ensure the accuracy of the content, OCR cannot be held responsible for any errors or omissions within these resources.   
© OCR 2017 - This resource may be freely copied and distributed, as long as the OCR logo and this message remain intact and OCR is acknowledged as the originator of this work.

OCR acknowledges the use of the following content: n/a

Please get in touch if you want to discuss the accessibility of resources we offer to support delivery of our qualifications: [resources.feedback@ocr.org.uk](mailto:resources.feedback@ocr.org.uk)

We’d like to know your view on the resources we produce. By clicking on ‘[Like](mailto:resources.feedback@ocr.org.uk?subject=I%20liked%20GCSE%20(9-1)%20Computer%20Science%20Programing%20techniques)’ or ‘[Dislike](mailto:resources.feedback@ocr.org.uk?subject=I%20disliked%20GCSE%20(9-1)%20Computer%20Science%20Programing%20techniques)’ you can help us to ensure that our resources work for you. When the email template pops up please add additional comments if you wish and then just click ‘Send’. Thank you.

If you do not currently offer this OCR qualification but would like to do so, please complete the Expression of Interest Form which can be found here: [www.ocr.org.uk/expression-of-interest](http://www.ocr.org.uk/expression-of-interest)

Looking for a resource? There is now a quick and easy search tool to help find free resources for your qualification:   
www.ocr.org.uk/i-want-to/find-resources/