Name: myng chan

Introduction to Programming

with C#

Bring this booklet to every lesson

[](https://www.bing.com/images/search?view=detailV2&ccid=xteMWvBx&id=A8D6446370385F3C0644294C98245BBAC58F1F71&thid=OIP.xteMWvBxCLXWBJRk4ryyxQHaHa&mediaurl=https%3a%2f%2fih0.redbubble.net%2fimage.416412087.0587%2fap%2c550x550%2c12x12%2c1%2ctransparent%2ct.png&exph=550&expw=550&q=c%23+logo&simid=607999452000620038&selectedIndex=2)

© 2020 Hills Road Sixth Form College**Course Outline**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Chapter** | **Objectives** | **Class Activities** | **Homework (bold=submit for marking)** | **Extension** |
| **Introduction**  **0. Hello World** | Learn how to use the **Visual Studio** (Integrated Development Environment)**,** to create, save, compile and run a minimal program. | Basic familiarisation with the IDE: Hello World  Analyse and comment BoxVolume program | Testing and Errors in next Chapter;  Correct Circle code  Self-Test: Data Types |  |
| **1.Temperature Converter Variables and Data Types** | Learn how to declare **variables** and the importance of data types. | Temperature conversion algorithm and program. Also implement corrected circle program. | Read: next Ch.;  Self-Test: Division operators and error types |  |
| **2.Card Game- Arithmetic, Error Handling and Selection** | This problem will give you practice in using a range of **arithmetic operations.**  Learn about different types of error and the basics of exception handling. | Develop algorithm. Complete and extend program. | Read: next Ch.  Self-Test: rewrite nested if | Exception Handling. Add loops to card game program |
| **3. Rock, Paper, Scissors –Making Decisions** | Learn how to use **if… then... else…** statements to control the flow of execution in a program. | Devise algorithm and complete rock, paper, scissors program. | Read: next Ch.  Self-Test: write simple loops | Design and write a leap year program |
| **4.Guessing Game – Repetition (2/3 lessons)** | Learn how to use **for, while and do-while** loops to control the flow of execution in a program.  Learn **which is the best type** of loop to use to solve a particular problem. | If miniquiz  Design and implement simple guessing game.  Extra exercises on p. 32 to reinforce loop constructs | Do: complete table on p.32 (type of loop)  Write pseudocode for two problems | 3n+1 algorithm  Roman Numerals |
| **5. Guessing Game Assignment**  **(2 lessons)** | Learn Linear and Binary Search algorithms | Devise algorithm for binary search.  Guessing Game | **Write up Ass. 1: Guessing Game** |  |
| **6.Encryption Introducing Strings**  **Caesar Shift** | Learn how to use **strings** to store text and how to manipulate individual characters within the strings.  Learn how to use **nested loops.**  Learn how to test a program using **hand-tracing and functional testing.** | Test and improve Caesar Shift program | Read: next Ch.  Self-Test: strings, scope, loops |  |
| **Short test on programming and algorithms** | To check that you can hand-trace an algorithm and write a short program based on supplied pseudocode. |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **7.Base Converter – Introducing functions** | Reinforce understanding of binary and decimal number bases.  Revisit algorithms to convert between number bases.  Learn how to use **functions** to split programs up into sections.  Understand **why** the use of functions makes programming easier.  Learn how a Hierarchy Chart can show the structure of a program. | Convert binary to decimal algorithm into C# function and add to program. | Read: next Ch.  Self-Test: arrays | Redesign Guessing game program to use functions. |
| **8.BubbleSort – Introducing arrays** | Experience the **bubblesort** algorithm!  Learn how **arrays** can be used to store multiple items of data  Gain more practice in using functions. | Demonstrate bubblesort using students.  Refactor Bubblesort program to use functions. | Read: Next Ch.  Self-Test: functions, arguments | Extend card game program. Write Battleships program. |
| **9.Selection Sort – reusing code with arguments** | Learn how to use **arguments** to supply data to functions  Appreciate **why** arguments make functions much more flexible and re-usable. | Refactor Bubblesort to include parameterised functions. | Read: next Ch.  Self-Test: functions | Fibonacci function |
| **10. Introducing functions** | Learn how to use **functions** as an alternative to functions. | “warmup tasks”: improve dectobin, add bintodec | **Ass. 2: Palindrome Tester** | Morse code  Fraction addition |
| **11. Assignment 2: Palindrome Tester** |  | Design palindrome tester algorithm. Start work on palindrome program. |  |  |
| **12. Assignment 3: Anagrams** | Consolidate programming work done so far | Debug Selection Sort function. Start work on Anagrams task. | **Ass. 3: Anagrams** |  |
| **13.Text Files**  **and reusing code** | To learn how to use text files to store data permanently.  Learn how to use a **unit** to re-use code. | Slide show on text files. Text File analyser, anagrams, Text file search |  |  |
| **14 Assignment 4: File Encryption** |  | Start Encryption assignment | **Ass. 4: Text File encryption** | Vigenere  One-time pad |
| **Practice Exam** | This exam will include theory and practical questions. |  |  |  |
| **15. Records and Binary Files – Address Book** | Learn about record structures and their uses  Learn about binary files, how they differ from text files and gain practical experience in using them | Slide show  Develop ideas for address book program.  Modify or replace existing program (2 lessons). |  |  |

# Contents

[Contents 4](#_Toc45095418)

[Introduction to C# and the course 6](#_Toc45095419)

[1.1 Homework: Advance Reading and Self-Tests 6](#_Toc45095420)

[2 Hello World – Introduction to the IDE 7](#_Toc45095421)

[2.1 Objectives 7](#_Toc45095422)

[2.2 Visual Studio 7](#_Toc45095423)

[2.3 Create simple application using Visual Studio 7](#_Toc45095424)

[2.4 Enhancing the Hello World application 8](#_Toc45095425)

[2.5 Extra Task – analysing a program 9](#_Toc45095426)

[2.6 Homework 10](#_Toc45095427)

[2.7 Self Test: complete and bring to the next lesson 11](#_Toc45095428)

[3 Temperature Converter–Variables and Data Types 12](#_Toc45095429)

[3.1 Objectives 12](#_Toc45095430)

[3.2 Advance Reading: Constants, variables and Data Types 12](#_Toc45095431)

[3.3 Lesson Task 13](#_Toc45095432)

[3.4 Homework 14](#_Toc45095433)

[4 Card Game –Arithmetic, Error Handling and Selection 15](#_Toc45095434)

[4.1 Objectives 15](#_Toc45095435)

[4.2 Advance Reading: integer division and remainders – the div (/) and mod (%) operators 15](#_Toc45095436)

[4.3 Extra tasks: 19](#_Toc45095437)

[4.4 Homework 19](#_Toc45095438)

[Rock, paper, scissors – Making Decisions 20](#_Toc45095439)

[4.5 Objectives 20](#_Toc45095440)

[4.6 Advance Reading and Examples 20](#_Toc45095441)

[4.7 Lesson Task – Rock, Paper, Scissors 25](#_Toc45095442)

[4.8 Extra Task – Leap Year 26](#_Toc45095443)

[4.9 Homework 27](#_Toc45095444)

[4.10 Self-Test: bring your answers to the next lesson 27](#_Toc45095445)

[5 Guessing Game - Repetition (looping) 28](#_Toc45095446)

[5.1 Objectives 28](#_Toc45095447)

[5.2 Advance Reading and Examples 28](#_Toc45095448)

[5.3 Lesson Task – Guessing Game 33](#_Toc45095449)

[5.4 Extra Exercises (lots, because loops are very important) 34](#_Toc45095450)

[5.5 Homework 35](#_Toc45095451)

[6 Assignment 1 - Guessing Game Mark 2 38](#_Toc45095452)

[6.1 Linear and Binary Searches 38](#_Toc45095453)

[7 Encryption – Introducing Strings 39](#_Toc45095454)

[7.1 Objectives 39](#_Toc45095455)

[7.2 Strings 40](#_Toc45095456)

[7.3 Testing the Caesar Shift Program 41](#_Toc45095457)

[7.4 Homework 43](#_Toc45095458)

[7.5 Self-Test: bring your answers to the next lesson 43](#_Toc45095459)

[8 Base Converter – Introducing Functions 45](#_Toc45095460)

[8.1 Objectives 45](#_Toc45095461)

[8.2 The binary to decimal algorithm 46](#_Toc45095462)

[8.3 Hierarchy Charts 46](#_Toc45095463)

[8.4 Tasks for Lesson 47](#_Toc45095464)

[8.5 Homework 47](#_Toc45095465)

[9 Bubblesort – Introducing Arrays 48](#_Toc45095466)

[9.1 Objectives 48](#_Toc45095467)

[9.2 Arrays 49](#_Toc45095468)

[9.3 Tasks 51](#_Toc45095469)

[9.4 Extra Exercises 51](#_Toc45095470)

[9.5 Homework 52](#_Toc45095471)

[10 Selection Sort – Re-using Code with Arguments 53](#_Toc45095472)

[10.1 Objectives 53](#_Toc45095473)

[10.2 Scratch recap 53](#_Toc45095474)

[10.3 Arguments 54](#_Toc45095475)

[10.4 The Program 55](#_Toc45095476)

[10.5 Structured Programming 56](#_Toc45095477)

[10.6 Lesson Task 56](#_Toc45095478)

[10.7 Homework 58](#_Toc45095479)

[11 Using Functions to return a value 59](#_Toc45095480)

[11.1 Objectives 59](#_Toc45095481)

[11.2 Advance Reading: Functions Example: modified base converter 59](#_Toc45095482)

[11.3 Lesson Tasks 61](#_Toc45095483)

[11.4 Extra Exercises: 61](#_Toc45095484)

[12 Assignment 2 – Palindrome Tester 62](#_Toc45095485)

[12.1 The Problem 62](#_Toc45095486)

[12.2 The Algorithm 62](#_Toc45095487)

[12.3 Task 62](#_Toc45095488)

[13 Assignment 3 - Anagrams 63](#_Toc45095489)

[13.1 Objectives 63](#_Toc45095490)

[13.2 Task 63](#_Toc45095491)

[14 Text Files and Re-using code 65](#_Toc45095492)

[14.1 Objectives 65](#_Toc45095493)

[14.2 Example: Using Text Files in C# 65](#_Toc45095494)

[14.3 Re-using code – using namespaces and classes to create a library of functions 66](#_Toc45095495)

[14.4 Tasks using text files: 68](#_Toc45095496)

[15 Assignment 4 - File Encryption 69](#_Toc45095497)

[15.1 Extensions – alternative encryption algorithms: 69](#_Toc45095498)

[16 Records and Binary Files- Address Book 70](#_Toc45095499)

[16.1 Objectives: 70](#_Toc45095500)

[16.2 Structures and Records 70](#_Toc45095501)

[16.3 Binary Files 70](#_Toc45095502)

[16.4 Assignment 71](#_Toc45095503)

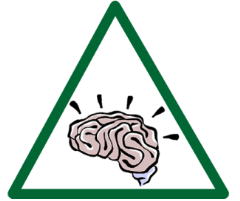
[Appendix 1 – Useful Library Functions 72](#_Toc45095504)

[Appendix 2 – C# Coding conventions 74](#_Toc45095505)

# Introduction to C# and the course

Throughout the course we shall be using the C# programming language (pronounced C sharp) . C# is a popular programming language developed by Microsoft that can be used to create programs for tablets, phones, web servers, and desktops. The Syntax of C# is similar to Java, C++, and JavaScript and it is supported by the Visual Studio programming environment.

We shall start by learning the basic syntax of C#; how to use variables and how to write simple programs that use the major features of algorithms: sequence, branching and looping. Then we shall go on to explore some more interesting algorithms, including sorting, encryption and base conversion. Many of the tasks set will involve completing and modifying programs, rather than creating your own from nothing.

Thinking Triangles

When you see this symbol in the booklet, you will work in your thinking triangles. Usually this will be to develop an algorithm. Be prepared to report back on your ideas and explain your reasoning.

C:\TEMP\Cache\Content.IE5\GAW7HHIT\MC900282180[1].wmfProgramming Pairs

When you see this symbol in the booklet, you will work in “programming pairs”. One person (“the driver”) will act as the typist, the other (”the navigator”) will tell them what to type. You will regularly swap roles and work with different partners.

## C:\TEMP\Temporary Internet Files\Content.IE5\YTVM5DIR\MC900078622[1].WMFHomework: Advance Reading and Self-Tests

In the early part of this course, your homework will often be to read the next lesson’s topic **in advance**. This is to make the best possible use of lesson time, which is to do practical programming work with support from your teacher.

To test your understanding you are given some fairly straightforward questions to answer. **Complete this in your booklet and bring it to the next lesson.**

Assignments

Later in the course most of your homework tasks will be to complete and document programming tasks. You should be able to do most of the practical work in the lesson and focus on the documentation at home.

# Hello World **– Introduction to the IDE**

## Objectives

Learn how to use the **Visual Studio IDE** (Integrated Development Environment)**,** to create, save, compile and run a minimal program.

## Visual Studio

Visual Studio can be rather daunting to begin with as it is designed for professional software developers. Your teacher will show you how to get started but use the next couple of pages if you need a reminder.

You should install Visual Studio on a computer at home, so that you can write code at home. Do not rely on only writing code in your lessons – this will not give you enough practice to reach the standard of programming required for A Level. College computers in the computing resource area have Visual Studio installed and you can often use the computer science classrooms for programming practice during lunchtime.

You can download Visual Studio from [www.visualstudio.com/downloads/](https://www.visualstudio.com/downloads/) . You want the Community Edition, there are downloads for both Windows and Mac. (Linux users can download a different product – Visual Studio Code).

## Create simple application using Visual Studio

Begin by creating a simple "Hello World" console application. Follow these steps:

1. Start Visual Studio Community (introduced in the last chapter), and select **File** -> **New** -> **Project**. From the project dialog, select the Console App (.NET framework). This is the most basic application type on a Windows system, but it's great for learning the language. Once you click Ok, Visual Studio creates a new project for you, including a file called Program.cs. This is where all the fun is, and it should look something like this.
2. Visual Studio uses the template to create your project. The C# Console Application template automatically creates some “boilerplate” code for you, which should look like this. We’ll look at this later on, but for now all we need to know is where we enter your own code.

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace ConsoleApp1  
{  
    class Program  
    {  
 static void Main(string[] args)  
 {

YOUR CODE GOES HERE  
 }  
   }  
}

1. Enter the following code into the correct place in the program

Console.WriteLine("Hello, world!");  
Console.ReadLine();

This code calls the built-in Console.WriteLine function to display the "Hello World!" in the console window. The Console.ReadLine() function waits for user entry, here it just makes the program wait so that the Console (the black window) stays on the screen.

By selecting the green arrow on the toolbar, or hitting F5, you can run the program. When you do, the console window is visible for only a brief time interval before it closes.

## Enhancing the Hello World application

Enhance your application to prompt the user for their name and display it along with the date and time.

To modify and test the program, do the following:

Enter the following C# code in the code window immediately after the opening bracket that follows the static void Main(string[] args) line and before the first closing bracket:

Console.WriteLine("\nWhat is your name? ");

var name = Console.ReadLine();

Console.WriteLine(" Hello " + name);

Console.WriteLine(" Press any key to exit... ");

Console.ReadLine();

This code displays "What is your name?" in the console window and waits until the user enters some characters, followed by the Enter key. It stores these characters as a string into a variable called name and displays the contents of name in the console window. Finally it displays a message explaining how to exit the program.

Enter this code and run the program. (Note that it you copied and pasted the code rather than typing it, you may get the wrong type of quote mark (“) inserted. This will give you an error and will need correcting.)

Output using different syntax:

string name = "Mark";

var date = DateTime.Now;

// String concatenation:

Console.WriteLine(" Hello " + name);

// Using composite formatting:

Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);

// Using string interpolation:

Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");

You've created and run your first C# application.

## Extra Task – analysing a program

Here is a simple program which shows you how to **output** to the screen, take **input** from the keyboard and use **variables**.

using System;

namespace PBoxVolume

{

// standard library

class PBoxVolume

{

// program header

static void Main(string[] args)

{

// variable declarations

int length, height, width, volume;

// input and assignment statements

Console.WriteLine("Enter length: ");

length = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter width: ");

width = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter height: ");

height = Convert.ToInt32 (Console.ReadLine());

volume = length \* width \* height;

// output statement

Console.WriteLine($"The volume of the box is " + volume);

Console.ReadLine();

}

Is there any difference between Convert.ToInt32 or Int32.Parse?

}

}

Your task is to:

* study the code

1. work out what it does: calculates the volume given 3 values.

* add **comments** to the code, identifying the following features:
  + program header
  + statement to include a standard library
  + variable declarations
  + output statement
  + input statement
  + assignment statement

**Comments** are used to give information about the action of the program. This is essential as it is important that others (including examiners and more importantly yourself!) can follow your intentions. In C# comments can be enclosed in like this /\* this is a comment \*/ or if they are on a single line they can be preceded by a double slash //

*/\* a multi-line comment looks*

*like*

*this \*/*

*//this is a single-line comment*

## Homework

**Do:**

1. Microsoft “numbers in C#” tutorial <https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/intro-to-csharp/numbers-in-csharp> tick

* The “Advance Reading” Section of the next chapter in this booklet “Variables and Data Types”
* Computing without Computers Chapter 2 "The Language Instinct"[http://www.dcs.qmul.ac.uk/%7Epc/research/education/puzzles/reading/cwcch2.pdf](http://www.dcs.qmul.ac.uk/%7Epc/research/education/puzzles/reading/cwcch4.pdf)

(also on the Computing SharePoint site, Introduction to Programming page)

**This will enable you to correct the errors in the following source code:**

using System;

namespace Circle

{

public class Program

{

static void Main(string[] args)

{

double radius;

double circumference;

Console.WriteLine("Program to calculate the circumference of a circle");

Console.WriteLine("Enter radius: ");

radius = Convert.ToDouble(Console.ReadLine());

circumference = radius \* 2 \* Math.PI;

Console.WriteLine("The circumference of the circle = " + circumference);

Console.ReadLine();

}

}

}

**You will need the corrected code in the next lesson**

## Self Test: complete and bring to the next lesson

1. What data type would you use to store the following:

|  |  |
| --- | --- |
| 1. Surname: string | string |
| 1. size of a file (in bytes): integer | integer |
| 1. telephone number: integer | integer |
| 1. area of a circle: real number | Real number |
| 1. whether a student owns a car (or not): boolean | boolean |

C:\TEMP\Temporary Internet Files\Content.IE5\YTVM5DIR\MC900078622[1].WMF

1. Write a C# statement to declare an integer called sum and initialise it to zero.

|  |
| --- |
| Int sum = 0; |

1. Write a C# statement that adds the two integer variables digit and value and assigns the result to the variable total.

|  |
| --- |
| Int value1 = 1;  Int value2 = 1;  Int total = value1 + value2; |

# Temperature Converter–Variables and Data Types

## C:\TEMP\Cache\Content.IE5\B84RKII7\MP900409423[1].jpgObjectives

Learn how to declare and use **constants** and **variables** and understand the importance of data types.

**The problem**

Write a program that converts a temperature in degrees Fahrenheit into Centigrade/Celsius.

Example:

If the user inputs 68 the program will output 20.

## Advance Reading: Constants, variables and Data Types

C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMFA **constant** is an item of data that has a fixed value, determined by the programmer, which never changes while the program is running. For example:

const double PI = 3.14159; //from the Circle program

Defining a constant is useful when a value is used several times in a program. It avoids having to type out the value each time, which can lead to mistakes and inconsistencies. Defining it once at the beginning of the code makes it quick and efficient if the programmer needs to change it.

A **variable** can be thought of as a place to store an item of data in memory.

Compare the following:

* 4
* “4”
* “four”
* 4.0

Each is data; each represents 4 but in a different way i.e. each is a different **data type**. The first is a positive whole number with a numeric value; the second is a single character with no numeric value; the third is a collection of individual characters that make up the word four, but once again, the word has no numeric value; the last is a number that could hold a fractional component, it has a numeric value.

The values of the first and last happen to be the same but **they will be stored in a computer's memory in very different ways.** This is true for all the above types of data: **each require a different storage format and amount of memory to be set aside**.

### Data types and their declaration

C# is an “**explicitly,** **statically-typed”** language[[1]](#footnote-2)(unlike Python, which you might have used previously). This means that all variables must be assigned a type before they can be used and that type can’t be changed. It is vital that you spend some time thinking about the data to be used in your programs and decide on a suitable name for the variable (**identifier**) and also a suitable data type for the variable.

Here are the most commonly-used types in C#:

|  |  |
| --- | --- |
| **int** | Integer - whole number, positive or negative eg. 56, -3419 |
| **double (or float)** | Positive and negative with fractional component. eg -3.65, 0.0008, also called floating point values |
| **bool** | Either true or false |
| **char** | Single characters eg ‘a’, ‘R’, ‘4’, ‘%’ |
| **string** | Collection of characters eg “dog”, “65£4” |

A type is associated with a variable when it is declared. In C# all variables must be declared before they can be used.

Examples:

string firstName;

int age;

bool hasDrivingLicence;

A type is associated with a variable when it is declared. In C# all variables must be declared before they can be used.

Examples:

string firstName;

int age;

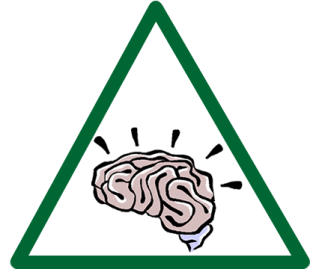
bool hasDrivingLicence;

You can initialise variables when you declare them:

string firstName = “Mark”;

## Lesson Task

## Lesson Task



**Develop an algorithm to solve this problem. Write it in pseudocode or structured English. Remember it should be precise and unambiguous.**

C:\TEMP\Cache\Content.IE5\GAW7HHIT\MC900282180[1].wmf

In Programming Pairs, write, compile and test a program to implement the temperature conversion algorithm. Swap the “driver” and “navigator” roles when you are halfway through.

### Things to Think About

* What variables will you need, what should you call them and what type should they be?
* How can you test the program? What runtime errors might occur?

### Extra Tasks

* Create a new project to enter, compile and test the Circle code you corrected for homework.
* Write a program which inputs the length and width of a rectangular garden. Calculate the area of the garden and the cost of turfing a lawn if a 1m unturfed border is around the perimeter of the garden. Assume the cost of turf is 10 per square metre.

## Homework

**Read**

* The “Advance Reading” Section of the next chapter in this booklet

1. What is the value of the integer variable *answer* after executing each of the following statements: (% is the modulus operator)

|  |  |
| --- | --- |
| * 1. answer =35/7; |  |
| * 1. answer =17/2; |  |
| * 1. answer =17 % 2; |  |

1. How would the value stored in answer differ if it was of double type

|  |  |
| --- | --- |
| * 1. answer =35/7; |  |
| * 1. answer =17/2; |  |
| * 1. answer =17 % 2; |  |

1. Look at this statement from a temperature converter program:

CelTemp = (FahrTemp - 32) \* 5/9;

Modify the statement in three different ways, to generate the three different types of error.

|  |  |  |
| --- | --- | --- |
| **Type of error** | **Modified statement** | **Effect of error** |
| Syntax error |  |  |
| Logic error |  |  |
| Runtime error |  |  |

# Card Game –Arithmetic, Error Handling and Selection

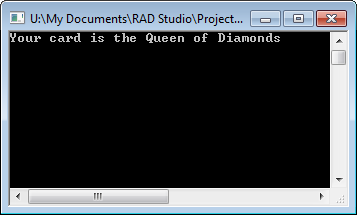
## C:\TEMP\Cache\Content.IE5\YC0R459I\MC900431593[1].pngObjectives

This problem will give you practice in using a range of **variable types** and the **arithmetic operations** you can perform on them.

**The problem**

Write a program that will display the rank and suit of a card chosen randomly from a standard pack.

**Example output**



## C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMFAdvance Reading: integer division and remainders – the div (/) and mod (%) operators

You should have no problems with addition, subtraction and multiplication. However, division is not quite so straightforward. When you divide an integer by another integer in C#, the result will also be an integer. In fact it will be the integer part of the answer, ignoring anything after the decimal point. Note that the result is always truncated (rounded down), it is not rounded to the nearest integer.

So after:

int cake = 15;

cake = cake/4;

the variable cake will contain 3, not 3.75 or 4 Indeed it can’t possibly contain 3.75 as it is an integer variable.

However after:

double cake = 15;

cake = cake/4;

cake will contain 3.75 as it is now of double type, so can store decimals

Also very useful is the modulus operator **%**. Modulus carries out integer division, but gives the **remainder** as the result: 9 % 2 gives a result of 1.

### Program errors

C:\TEMP\Cache\Content.IE5\CM28VMZL\MC900104748[1].wmfIt is important to ensure that the program actually does what it set out to do. There are various kinds of error that can occur when programming:

1. **Syntax error** – Your program does not follow the rules of the programming language, thus the program will not compile.
2. **Logic error** - All the code compiles, the program runs but the output data is incorrect. In this case the error is in the algorithm.
3. **Runtime error** – When the program is running, an error occurs usually resulting in the application (and perhaps the operating system) crashing.

Each type of error will be detected differently:

**Syntax errors** will be detected by the **compiler**, though the error messages it produces are not always easy to understand.

**Logic errors** are the responsibility of the **programmer** to detect and correct, require careful testing and you will learn more about this in a later lesson. For now, use values like 32oF and 212oF, as well as others with known results.

**Runtime errors** are also the **programmer**’s responsibility to deal with. Try entering a string of characters for the temperature in the above program and observe the nasty results!

### Exception Handling – dealing with run-time errors

Most programming languages include a feature called Exception Handling to deal with run-time errors.

When a run-time error occurs, the system generates an “exception”. The programmer can include instructions in the code, telling the system what to do when an exception occur. It is usually possible to detect the type of exception and this means you can handle different errors in different ways.

#### C:\Users\icrosby.HRSFC\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\Z3Y2LQU4\lgi01a201405162200[1].jpgExample

In the circle program, (and in most of the programs you will write this term) the user can make the program crash by entering a non-numeric value (eg “sheep”) for the radius of the circle, because the program can’t convert “sheep” into a number. When this happens this program will raise an exception which the programmer can handle like this:

.

Console.WriteLine("Program to calculate the circumference of a circle");

Console.Write("Enter circle radius: ");

try

{

radius = Convert.ToDouble(Console.ReadLine());

diam = radius \* 2;

circ = PI \* diam;

Console.WriteLine("The circumference of the circle = " + circ.ToString());

Console.ReadLine();

}

catch (System.FormatException Exception)

{

Console.WriteLine(Exception.Message);

Console.WriteLine("Please enter a number");

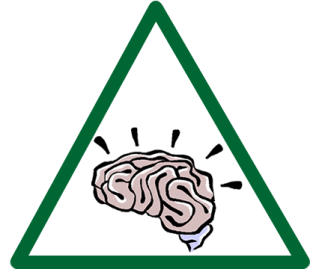
Console.ReadLine();

}

There is a lot more to exception handling than this: you can customise the action to be taken according to the type of error and much more besides, but this is enough to get you started.

#### Cautionary Note

Exception Handling is powerful and useful but can make code hard to follow and is often overused by inexperienced programmers. It is mentioned here for completeness but we recommend that you do not use exception handling until later in the course. It is not even actually required for A Level, but you may wish to use it in your project next year.



**Develop an algorithm to solve the Card Game problem. You must use the div(/) and mod(%) operators. You will also need to use a random number and if statements. Don’t worry about the C# syntax for these: just use pseudocode for now.**

**The Program**

You are provided with an incomplete Card Game program (code below).

It already includes the statement to generate a random number and some incomplete code to output the rank and suit for the card (the **if** and **case** statements). You have already met the concept of selection in Scratch and Raptor so you should be able to understand the principle. You’ll be learning more about selection in the next lesson.

Your task is to:

* complete it:
* add in the operations to generate the rank and suit of the card (this is the part that uses **%** and **/** (mod and div in Delphi)
* complete the selection (**if** and **case**) statements
* test your algorithm by running the program. If necessary revise your algorithm until you have fixed any bugs.

C:\TEMP\Cache\Content.IE5\GAW7HHIT\MC900282180[1].wmf

Here is the code:

using System;

namespace PCardGameIncomplete

{

public class PCardGameIncomplete

{

public static void Main(string[] args)

{

int i, number, ranknum, suitnum;

string rank = “”;

string suit = “”;

// "seeds" (initialises) the random number generator

Random random = new Random();

//selects a random number between 0 and 51,

//adds one to it and assigns the result to number

number = random.Next(52) + 1;

// ??? complete this using the method you came up with in your triangles

ranknum =

suitnum =

// ??? complete this case statement for the other ranks

switch (ranknum)

{

case 1:

rank = "Ace";

break;

case 2:

rank = "Two";

break;

case 3:

rank = "Three";

break;

}

// add if statements for the other suits

if (suitnum == 0)

{

suit = "Clubs";

}

// output using an interpolated string

Console.WriteLine($"Your card is the {rank} of {suit}”);

// first alternative old format

// Console.WriteLine("Your card is the " + rank + " of " + suit);

// second alternative format for the output

// Console.WriteLine(("Your card is the {0} of {1}", rank, suit);

Console.ReadLine();

}

}

}

## Extra tasks:

(These require the use of loops and are **optional**. You can find out how to use loops by looking further ahead in the booklet)

1. Add an exception handling block to your program, following the example on page 17.
2. Extend the program so that it repeatedly chooses a random card until the user enters “q’.
3. Extend the program so that it shows every card in a standard pack in order from the two of clubs to the ace of spades.

## Homework

**Read**

Tutorial on selection in C#: <https://csharp-station.com/Tutorial/CSharp/Lesson03>

**Study**

* The Advance Reading and example programs in the next chapter of this booklet: Triangle, Payroll and Grades

#### C:\TEMP\Temporary Internet Files\Content.IE5\YTVM5DIR\MC900078622[1].WMFSelf-Test: bring your answers to the next lesson

Nested if… then… else statements can be very hard to understand and debug. Rewrite the following, without any nested ifs, using:

1. complex Boolean expressions (see Example 3 in the next Chapter)
2. a case statement (also in Example 3)

if (rank == 13)

{

Score = 11

}

else if (rank >= 10)  
 {

score = 10

}

else

{

score = rank;

}

You can assume that rank has a value between 1 and 13

|  |
| --- |
| Using complex Boolean expressions: |
| Using a case statement: |

# Rock, paper, scissors – Making Decisions

## [File:Rock-paper-scissors.svg](http://upload.wikimedia.org/wikipedia/commons/6/67/Rock-paper-scissors.svg)Objectives

Learn how to use **if… else…** and **case** statements to control the flow of execution in a program.

**The Problem**

Write a program which allows the user to play “rock, paper, scissors” against the computer. The computer should choose at random and then display the result of the contest.

## Advance Reading and Examples

### Example 1 - triangle.cs – if… else statement

C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMFRead in the three whole numbers sides of a triangle and test to see if the triangle is right-angled using Pythagoras” Theorem.

The structure of the if.. else.. statement is very similar to the equivalent in Scratch.



using System;

namespace triangle

{

public class triangle

{

public static void Main(string[] args)

{

//get three sides of the triangle

Console.WriteLine("Enter shortest side > ");

int SideA = Convert.ToInt32(Convert.ToInt32(Console.ReadLine()));

Console.WriteLine("Enter mid length side > ");

int SideB = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter longest side > ");

int SideC = Convert.ToInt32(Console.ReadLine());

// Test for right angled triangle

if ((SideC \* SideC) == ((SideA \* SideA) + (SideB \* SideB))

{

Console.WriteLine("A right angled triangle");

}

else

{

Console.WriteLine("Not a right angled triangle”);

}  
Console.ReadLine();

}

}

}

### C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMFExample 2 - Payroll.dpr – compound statements

Another example of an **“if”** (selection structure), which uses a **compound statement.**

Here is an example in Scratch:



The *if* block contains **two** blocks: the execution of both is dependent on the *if* expression being true. In Scratch this is very easy to create and understand. In C# , you have to be more careful to get it right:

using System;

namespace payroll

{

class payroll

{

static void Main(string[] args)

{ /// note the use of a constant. Why do this?

const double PayRate = 5.45;

int Hours, Overtime;

double Pay;

Console.WriteLine("Enter number of hours worked: ");

Hours = Convert.ToInt32(Console.ReadLine());

// calculate basic pay

Pay= Hours \* PayRate;

// calculate the overtime and add to basic

if (Hours > 40)

{ //start of compound statement

Overtime= Hours - 40;

Pay = Pay + (Overtime \* PayRate);

} //end of compound statement

// display total pay

Console.WriteLine("Total pay due is £" + Pay.ToString("N2"));

Console.ReadLine();

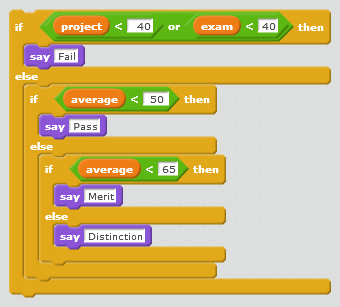
}

}

}

The statements within the if block must be enclosed by curly brackets **{** and **}**. This structure is called **a compound statement**. **One of the commonest mistakes when starting to learn to program is to forget curly brackets.** If you miss them out then only the first statement is attached to the *if* or *else* (see the Triangle example on the previous page: only the *write* statement belongs to the *else*). Remember that indentation means nothing to the compiler: it is important to aid readability but unlike in Python, it will not group the statements together.

### Example 3 - Grades.cs - Nested ifs and complex Boolean Expressions

C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMFIn order to pass an exam a student must achieve a minimum of 40% on both a project and a written paper. A student who satisfies this requirement will be awarded a Pass if the average mark is > 50, a merit if it is >= 50 and < 65, and a Distinction for 65 and over. Marks are all whole numbers and the average is rounded up to the nearest whole number.

**Program:**

using System;

namespace grades

{

class grades

{

static void Main(string[] args)

{

int Project, Exam, Average;

{

// get marks

Console.Write("Enter project mark: ");

Project = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter exam mark: ");

Exam = Convert.ToInt32(Console.ReadLine());

// calculate the rounded average

Average = (Project + Exam + 1) / 2;

Console.WriteLine("Overall average is "+ Average.ToString());

// determine grade

if ((Project < 40)||(Exam < 40))1

{

Console.Write("Student has failed");

}

else if (Average < 50)

{

Console.Write("Student has gained a pass");

}

else if (Average < 65)

{

Console.Write("Student has gained a merit");

}

else

{

Console.Write("Student has gained a distinction");

}

Console.ReadLine();

}

}

}

}

**Notes**

This is a **complex boolean expression**; note that the individual expressions as well as the overall expression **must be enclosed in round brackets**. Expressions can also be connected using the and operator **&& (and)**.

### Switch statement: an alternative to nested ifs

C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMFNested *if*s, as in the example above, can be confusing and hard to get right. C# provides a more readable alternative: the *switch* statement. Look at the following example:

static void Main(string[] args) {

char grade = 'B';

switch (grade) {

case 'A':

Console.WriteLine("Excellent!");

break;

case 'B':

case 'C':

Console.WriteLine("Well done");

break;

case 'D':

Console.WriteLine("You passed");

break;

case 'F':

Console.WriteLine("Better try again");

break;

default:

Console.WriteLine("Invalid grade");

break;

}

Console.WriteLine("Your grade is {0}", grade);

Console.ReadLine();

}

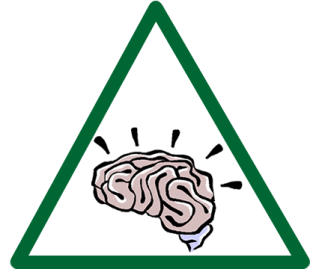
}

}

**NOTES**:

1. Don’t forget the **break;** statement after the code for each case. In C# execution ‘drops through’ from one case to the next, the **break;** stops this happening. Sometimes however you do want it to happen as per B and C grades in the above example, where the message is the same for both.

## Lesson Task – Rock, Paper, Scissors



**Develop an algorithm for Rock, Paper, Scissors.**

C:\TEMP\Cache\Content.IE5\GAW7HHIT\MC900282180[1].wmf**The Program**

You are provided with an incomplete version of the program (printed below). Complete and test it using the algorithm you designed in your triangles.

using System;

namespace RockPaperScissors

{

class RockPaperScissors

{

static void Main(string[] args)

{

string computerobject = "";

int computerchoice = 0; ;

bool computerwins = false;

Random random = new Random();

Console.WriteLine("Enter your choice r(ock),p(aper),s(cissors): ");

string userobject = Console.ReadLine();

char userobjectFirst = userobject[0];; **//what does this do?**

computerchoice = random.Next(3); **// what does this do? Explain here**

**//replace the following three if statements with a case statement**

if (computerchoice == 0) computerobject = "rock";

if (computerchoice == 1) computerobject = "paper";

if (computerchoice == 2) computerobject = "scissors";

Console.WriteLine("Computer choses " + computerobject);

char computerobjectFirst = computerobject[0]; **//what does this do?**

if (String.Equals(computerobjectFirst, userobjectFirst)) Console.WriteLine("It's a draw");

else

{

**//complete the missing logic to decide who has won**

if (computerwins == true)

Console.WriteLine("Computer wins!");

else

Console.WriteLine("You win!");

}

Console.ReadLine();

}

}

## Extra Task – Leap Year

A year is a leap year if it is divisible by 4, unless it is a century year, in which case it is only a leap year if it is divisible by 400 (e.g. 2000 is a leap year, but 1900 was not). Design and write a program to accept a year and output a message indicating if it is a leap year.

## Homework

**Read**

* Tutorial on iteration: <https://csharp.net-tutorials.com/control-structures/loops/>
* The Advance Reading and Examples section in the next chapter of this booklet.

## C:\TEMP\Temporary Internet Files\Content.IE5\YTVM5DIR\MC900078622[1].WMFSelf-Test: bring your answers to the next lesson

In each case write a loop in C#

|  |
| --- |
|  |

1. to output the numbers 1 to 100.
2. to output the letters of the alphabet from ‘A’ to ‘Z’.

|  |
| --- |
|  |

1. to repeatedly ask the user to input an integer until they enter 7.

|  |
| --- |
|  |

# Guessing Game - Repetition (looping)

## Objectives

* Learn how to use **for loops** and **while** loops to control the flow of execution in a program.
* C:\TEMP\Cache\Content.IE5\4TNAPH14\MC900441523[1].wmfLearn **which is the best type** of loop to use to solve a particular problem.

**The problem**

Write a program to get the computer to generate a random number between 1 and 100. Set up a loop to keep inputting the guess. If the guess is below the number then output “too low’. If the guess is above the number then output “too high’. Stop the program when the user guesses the number.

## C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMFAdvance Reading and Examples

**Loops** (also called **iteration** or repetition constructs) **are the most important structure to master in algorithm design.** Many program errors are caused by poorly constructed loops. The key feature is the exit strategy: **when should the loop terminate?** This should be your focus when you design a loop.

There are fundamentally two types of loop:

* **for** loops: when the number of iterations (repetitions) is **pre-determined**
* **while** loops: when the number of repetitions is **not** predetermined

For example, consider the following trivial problem:

Design an algorithm and write a program to find the average of a series of numbers.

There are two main ways to decide how many numbers are to be entered:

* ask the user in advance how many numbers they are going to enter: the number is **predetermined** so a **for** loop is needed
* ask them each time whether they want to enter another number: the number is not predetermined so a **while** or **repeat** loop is needed.

### For loops

The **for** keyword indicates a loop in C#. The for loop executes a block of statements repeatedly until the specified condition returns false.

Syntax:

for (variable initialization; condition; steps)

{

//execute this code block as long as condition is satisfied

}

As per the syntax above, the for loop contains three parts: initialization, conditional expression and steps, which are separated by a semicolon.

variable initialization: Declare & initialize a variable here which will be used in conditional expression and steps part.

condition: The condition is a boolean expression which will return either true or false.

steps: The steps defines the incremental or decremental part

Consider the following example of a simple for loop.  
for (int i = 0; i < 10; i++)

{

Console.WriteLine("Value of i: {0}", i);

}

**Pseudocode Algorithm for averaging programing using a for loop:**

Total =0

input count

**for I = 1 to count do**

**{**

Input number

Total =total+number

}

Average =total/count

output average

**Average Calculation using a for loop:**

using System;

namespace ForAverage

{

class ForAverage

{

static void Main(string[] args)

{

int Count, i;

double Average, Number, Total;

// must be able to hold non-whole numbers

Total= 0;

Console.Write("How many numbers do you want to enter ? ");

Count = Convert.ToInt32(Console.ReadLine());

for (i = 1; i <= Count; i++)

// meaning: start with i equal to 1, increment i by one each time round loop while i is less than of equal to count.

{

Console.WriteLine("Enter number ");

Number = Convert.ToDouble(Console.ReadLine());

Total = Total + Number;

} //end of for loop

Average= Total / Count;

Console.WriteLine ("Average is: " + Average.ToString("N2"));

Console.ReadLine();

}

}

}

**Notes:**

1. The number of iterations is controlled by a variable (the "control variable"), in this case called "i”.
2. The control variable is given a starting value (1) and a finishing value (count).
3. The **compound statement**, which is the block of code to be repeated, is between curly brackets and is indented for readability.
4. It is good practice to use a *for* loop if you can. You are less likely to make mistakes with for loops than with *while* loops.

### While and Do While loops

*While* and *do* *while* loops are both examples of loops where the number of iterations is **not** **pre-determined**. They differ in the location of the test to determine whether to go round again. In the while loop the test is at the beginning so the loop won’t execute at all if the loop condition isn’t met.

**Pseudocode Algorithm for averaging program using a while loop**

Total = 0

Count = 0

Input number

**While number != -99**

**{**

Count = count+1

Total = total + number

Input number

}

Average=total/count

Output average

Here the program will loop until the user enters -99.

**Program:**

namespace WhileAverage

{

class WhileAverage

{

static void Main(string[] args)

{

int Count = 0;

double Average, Number, Total;

Total = 0;

Console.WriteLine("Enter number ");

Number = Convert.ToDouble(Console.ReadLine());

while (Number != -99)

{

Console.WriteLine("Enter number ");

Number = Convert.ToDouble(Console.ReadLine());

Total = Total + Number;

Count = Count + 1; //count how many numbers have been entered

} //end of for loop

Average = Total / Count;

Console.WriteLine("Average is: " + Average.ToString("N2"));

Console.ReadLine();

}

}

}

**Question**

Why must the first number be entered before the loop begins?

### Do While loops

*Do while* loops are very similar to *while* loops. The key difference is that the test to decide whether to carry on looping is made at the **end** of the loop. So the loop always executed at least once.

A do while loop is not suitable for our averaging program because the -99 would be used in the calculation before the loop exits.

Here is a simple program using a do while loop.

using System;

namespace Loops

{

class Program

{

static void Main(string[] args)

{

int a = 10;

do

{

Console.WriteLine(a);

a = a + 1;

} while (a < 20);

Console.ReadLine();

}

}

}

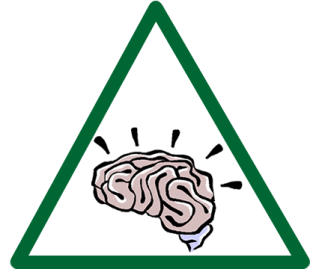
**What does this program do?**

### The break statement

The break statement can be used to break out of a loop (for, while, do-while, etc.). The program drops down immediately to the statement that follows the loop code. Inexperienced programmers sometimes use break in an if statement to compensate for choosing the wrong type of loop, resulting in code that is confusing to read. You should avoid using the break statement inside loops so that you get practice in choosing the correct type of loop.

## Lesson Task – Guessing Game

**The Algorithm**



**Once again, you will develop this with your triangle.**

**The Program**

Now write your guessing game program.

### Generating a random number for the guessing game program

You’ve met this before, in the Card Game program. Here’s how to generate a random number between 1 and 100:

Random random = new Random();

int myRandomNumber = random.Next(100)+1 ; // generate random number

Don’t worry about the first statement – this will make more sense once we start the object-orientated module

In the second statement, random.Next(n) generates a random integer between 0 and n-1, in this case between 0 and 99. So, we need to add 1 to get a number between 1 and 100.

## Extra Exercises (lots, because loops are very important)

**In each case, first decide whether to use a for loop (if the number of iterations is pre-determined) or a repeat/while loop (if the number of iterations is not pre-determined)**

|  |  |
| --- | --- |
| Problem | Type of loop (For or repeat/while) |
| 1. Display the ASCII characters associated with integers from 32 to 126 inclusive. Do not try to display those less than 32. (Why?) (Hint: use the *chr* function to convert an integer into its corresponding ASCII value.) |  |
| 2. If you have not already done so, add loops to the Card Game program (the Extra tasks at the top of page 20). |  |
| 3. In compound interest calculations, the amount in an account at the start of the year is multiplied by 1 + R/100, where R% is the interest rate, to give the amount at the end of the year. Display the amount in the account at the end of each year when an amount (input) is invested for N (input) years at R% (input) per annum. |  |
| 4. Write a loop that finds the first power of two that is equal to, or greater than, a number entered by the program user. Be sure that it will work correctly if the user enters zero. |  |
| 5. Imagine you have a can of whipped cream. Every second the nozzle stays open, 2% of the remaining cream squirts out. How long, to the nearest second, will it take to empty half the can? |  |
| 6. Write a loop that generates random numbers in the range 1 to 100 until ten numbers are generated in the range 1 to 10 and ten are in the range 91 to 100. How many numbers were generated in all? |  |
| 7. One plus two is three. One plus two plus three is six. One plus two plus three plus four is ten. If we keep this up long enough, we'll find that the sum of the numbers 1 to n is a perfect square. Find the first three numbers for which this is true. |  |

## Homework

**Do:**

* For each of the exercises above, write down the most appropriate type of loop to use.
* For **two** of the exercises above, write an algorithm **in pseudocode** to solve the problem. Write your solutions in the boxes on the next page.

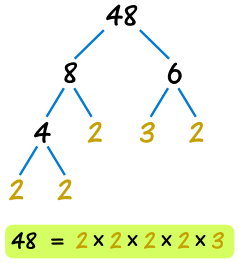
Pseudocode solutions:

ProblemNo:

ProblemNo:

### Extension Exercises for Confident Programmers

1. Distinct Prime Factorisation (BIO round 1 2012)

A *prime number* is a whole number, greater than 1, that can only be divided by itself and the number 1.

Every integer greater than 1 can be uniquely expressed as the product of prime numbers (ignoring reordering those numbers). This is called the *prime factorisation* of the number*.*

For example:

* 100 = 2 ! 2 ! 5 ! 5
* 101 = 101 (since 101 is a prime number)

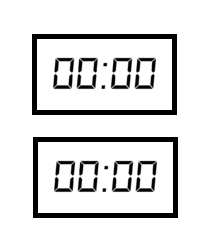
In this question we are interested in the product of the distinct prime factors of a given number; in other words each number in the prime factorisation is to be used only once.

For example:

* Since 100 = 2 x 2 x 5 x 5 the product we require is 10 (i.e. 2 x 5)

Write a program which reads in a single integer *n* (1 < *n* < 1,000,000) and outputs a single integer, the product of the distinct prime factors of *n*.

1. Watching The Clock (BIO Round 1 2013)



Two *clocks*, which show the time in hours and minutes using the 24 hour clock, are running at different speeds. Each clock is an exact number of minutes per hour fast. Both clocks start showing the same time (00:00) and are checked regularly every hour (starting after one hour) according to an accurate timekeeper.

What time will the two clocks show on the first occasion when they are checked and show the same time?

**NB: For this question we *only* care about the clocks matching when they are checked.**

For example, suppose the first clock runs 1 minute fast (per hour) and the second clock runs 31 minutes fast (per hour).

* When the clocks are first checked after one hour, the first clock will show 01:01 and the second clock will show 01:31;
* When the clocks are checked after two hours, they will show 02:02 and 03:02;
* After 48 hours the clocks will both show 00:48.

Write a program which reads in a two integers, each between 0 and 50,000 inclusive, indicating the number of minutes fast (per hour) of the first and second clock respectively.

You should output the time shown on the clocks when they first match. Both the hour and the minutes should be displayed with two digits.

1. Fibonacci Letters (BIO Round 1 2011)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |

Each letter in the alphabet can be given a value based on its position in the alphabet, A being 1 through to Z being 26. Two letters can produce a third letter by adding together their values, deducting 26 from the sum if it is greater than 26, and finding the letter whose value equals this result.

For example:

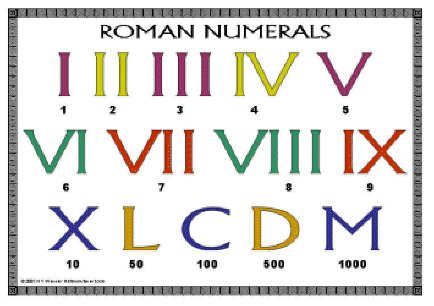
* A and B produce the letter C since 1+2=3 and A, B and C are respectively letters 1, 2 and 3 in the alphabet.
* P and Q produce the letter G since 16+17=33, 33-26=7 and P, Q and G are respectively letters 16, 17 and 7 in the alphabet.

We can generate a sequence of letters by starting with two letters and repeatedly using the last two letters in the sequence to produce another letter.

For example (starting with A and A) we have: A, A, B, C, E, H, M, U, H, C, …

Write a program which reads in two capital letters (the 1st letter in a sequence followed by the 2nd letter) then an integer *n* (1 <= *n* <= 1,000,000). You should output a single capital letter, the *nth l*etter in the sequence that starts with the input letters.

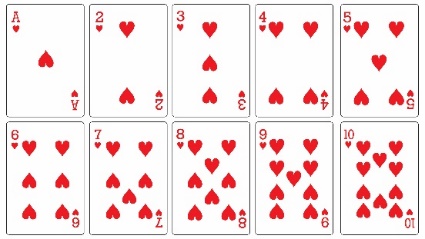
1. Roman Numerals (BIO Round 1 1998)

[](http://www.historyonthenet.com/shop/images/detailed/0/romnuma3.gif)

In Roman times, numbers were represented using letters. The way of doing this, known as Roman Numerals, is often seen depicting the copyright date on films and television.   
Roman numerals are conventionally defined to represent numbers using seven letters: I=1, V=5, X=10, L=50, C=100, D=500 and M=1000. Numbers other than these are formed by placing letters together from left to right, in descending order of size, and adding their values. The basic rule is to always use the biggest numeral possible (e.g. 15 is represented as XV, but never as VVV, VX or XIIIII). Letters may not appear more than three times in a row, so there are six exceptions to these rules - the combinations IV, IX, XL, XC, CD and CM. In these cases a letter is placed before one of greater value, and the smaller value is subtracted from the larger, e.g. CD = 400.  
Examples:  
26 XXVI  
94 XCIV  
555 DLV  
1998 MCMXCVIII

Write a program which accepts a number, between 1 and 3999 inclusive, and outputs the same number in Roman numerals.

1. Cards (BIO Round 1 2007)

[](http://www.google.co.uk/url?sa=i&rct=j&q=cards+&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRw&url=http://learnpatch.com/2013/05/rewiring-corporate-learning-10-know-when-to-stick-twist-or-quit/&ei=B05bVeHeIKSM7AbFg4HICg&bvm=bv.93564037,d.ZGU&psig=AFQjCNHDl-MzQKUcemVbJdallXZMlW0K5g&ust=1432133509991059)A card game is played with a deck of forty cards, containing each of the numbers from 1 to 10 exactly four times. The game is scored according to the following two rules: a point is given for each pair of cards with identical numerical values and for any group of cards whose numerical values sum to 15.

For example, the set of cards 8, 8 and 8 is worth three points since there are three different pairs of cards with identical numerical values. The set 10, 5, 2 and 3 is worth two points since there are two groups of cards whose numerical values sum to 15.

Write a program which inputs 5 numbers (each of which will between 1 and 10 inclusive) indicating the numerical values on 5 different cards. Your program should print out the number of points these 5 cards are worth and then terminate.

# Assignment 1 - Guessing Game Mark 2

**This programming task will be collected in and marked.** You will have time to do most of the programming in class but will have to finish it off at home.

You must include:

* Pseudocode algorithms
* Program listings which are well-presented and fully-commented
* Screen shots showing your programs working with typical data

## Linear and Binary Searches

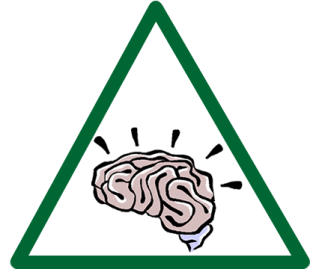
This task is based on the program you wrote in the previous section where you had to guess what number the computer had chosen. **This time you are going to choose the number and the computer is going to do the guessing**. This problem will introduce you to a very important group of algorithms: those which **search** for information.

### Algorithm 1 – Linear Search

This is just a fancy name for trying every possible number in turn. So your program must start by “guessing” 1, then 2, 3, etc. up to 100. By now you should have enough experience of loops to design the algorithm and code the solution yourself but work with a partner if you need to.

The linear search is an *inefficient* algorithm. You will discuss what this means. This leads on to:

### Algorithm 2 - Binary Search

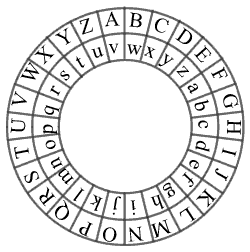
This alternative searching algorithm is much more efficient than the linear search. Your teacher will introduce the concept, you will probably have come across it before, but you will need to work together to develop an algorithm. **Don’t make the mistake of trying to write the program before you have written an algorithm.**

### Extensions for confident programmers

1. Combine the two algorithms into a single program.
2. Add a menu and a loop which asks the user to choose between the two functions and continues until they opt to quit.
3. Improve the binary search algorithm to stop the user cheating by entering contradictory answers.
4. Use functions/functions to improve the structure of your code (see Chapter 6)

# Encryption – Introducing Strings

## Objectives

* Learn how to use **strings** to store text and how to manipulate individual characters within the strings.
* Learn how to use **nested loops.**
* Learn how to test a program using **hand-tracing** and **functional testing.**

**The Problem**

Write a program to **encrypt** a string. A simple “Caesar shift”: each character in the plaintext is replaced by the one *shift* places from it in the alphabet eg if *shift*=3 then a is replaced by d, b by e, …., x by a, y by b, z by c.

**The Program**

Here is a very basic, incomplete Caesar Shift program.

Note that it uses an array of characters to represent the the ciphertext instead of a string. This is because strings are immutable in C# (can’t be changed). For both the string and the array, square brackets refer to individual characters.

It also ‘nests’ one loop inside another. Here the ‘shift’ loop happens for every letter of the input text.

using System;

using System.Text;

namespace CaesarBasic

{

class CaesarBasic

{

static void Main(string[] args)

{

int shift;

char letter;

string plaintext;

string ciphertext = "";

// get the message

Console.WriteLine("Enter your message to be encrypted: ");

plaintext = Console.ReadLine();

Console.Write("Enter the shift value (key): ");

shift = Convert.ToInt32(Console.ReadLine());

// encrypt it

// outer loop: step through the plaintext, one character at a time

for (int i = 0; i < plaintext.Length; i++)

{

letter = plaintext[i];

// note the use of square brackets to refer to an individual character within

//the string

for (int n = 0; n < shift; n++) // inner loop

{

letter++; // move the ith letter on to the next in the ASCII table,

// ++ is short for 'add one'

}

ciphertext += letter; //This appends the shifted letter to the ciphertext

}

// output the encrypted version

Console.WriteLine("The encrypted message is " + ciphertext);

Console.ReadLine();

}

}

}

## Strings

You have already used strings in earlier programs. The encryption program introduces some new techniques:

1. You can refer to individual characters using the square brackets [] notation. This code reads in a string and then displays each character on a new line.

using System;

namespace StringPrint

{

class StringPrint

{

static void Main(string[] args)

{

string name;

Console.WriteLine("Enter Some Text:");

name = Console.ReadLine();

for (int n = 0; n < name.Length; n++)

//the length function is very useful when dealing with strings

{

Console.WriteLine(name[n]);//write the nth character of the string

}

Console.ReadLine();

}

}

}

1. Another very useful tool for manipulating strings is the “concatenation operator”, used to add one string to another. It’s a very long name for something very simple. C# uses “**+=**” for this . Here is an example:

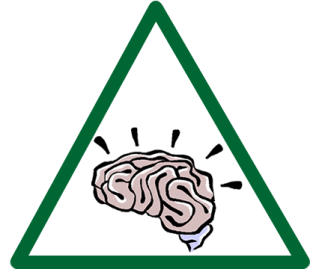
string space = “ “;

fullName = firstName += space += surname;

If FirstName is a string containing “John” and Surname contains “Smith” then FullName will become “John Smith”.

## Testing the Caesar Shift Program

The Caesar Shift program does not work correctly.



How can we test it systematically?

There are many different ways of testing an algorithm or program. Here are two of them for you to try on the base converter program:

### Dry-run testing (also called hand-tracing)

This test function does not require a computer, just pencil and paper; however, it does require access to the code or pseudocode. In this case you are going to use it to test the Caesar Shift program:

Start with a table, with a column for each variable in the program, like this:

|  |  |  |
| --- | --- | --- |
| i | N | plaintext |
|  |  |  |
|  |  |  |

I have chosen the order of the columns carefully, to match the order in which the variables are changed in the for loops: this makes the table easier to complete and understand. To carry out a test, choose an initial value for the *plaintext*: say “**cat”** and a value for *shift*, say **3**. Now work systematically through the code, recording the values of each variable as they change:

|  |  |  |
| --- | --- | --- |
| i | N | plaintext |
|  |  | “cat” |
| 1 | 1 | “dat” |
| 1 | 2 | “eat” |
| 1 | 3 | “fat” |
| 2 | 1 | “fbt” |
| 2 | 2 | “fct” |
| 2 | 3 | “fdt” |
| 3 | 1 | “fdu” |
| 3 | 2 | “fdv” |
| 3 | 3 | “fdw” |

The function has passed this test, though that doesn’t mean that it will work for **all** values of *plaintext*. Dry-running an algorithm or code in this way is also very useful for understanding how someone else’s code works.

**Task:** Complete the table below, by dry running the Caesar Shift program, using an initial value of “**axe”** for plaintext. **Be careful:** don’t make any assumptions.

|  |  |  |
| --- | --- | --- |
| i | n | plaintext |
|  |  | “axe” |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

If you think the code passes this test, then think again!

### Functional Testing

In contrast to dry-run testing, functional testing **does** require a computer, but **not** access to the code, only the final executable and the specification of what the program is supposed to do. It is often called **black-box testing** because the tester sees the program as a black box and cannot see the internal workings.

The aim of functional testing is to determine whether the program does what it’s supposed to, for as wide a range of inputs *as possible*. Notice the last two words in the previous sentence. For most programs it is **impossible** to test that they work correctly for **all** possible inputs, because there is an infinite number of them. So, the tester has to design tests for all reasonably likely inputs: this includes those resulting from user error and ignorance!

Some general principles can be applied when designing data for functional tests:

* All users make **mistakes**
* Programs are most likely to fail with **extreme data**
* Programmers most often make mistakes at data **boundaries**

Applying these principles to the Caesar Shift program:

**User errors**:

non-numeric data for the shift value: eg 5q, non-integer data: eg 3.5

**Extreme data**:

0 or very large numbers for the shift value, an empty or very long string for *plaintext*

**Boundary data**:

For the shift value: 0 (the minimum acceptable value), -1 (only just unacceptable),

For *plaintext*: a string containing letters near the end of the alphabet, which should “wrap around” to the beginning.

In order to choose test data correctly it is essential that the tester has a precise specification for the program: what data should it accept?; how should it respond to invalid data?; what are the expected outputs for a range of valid data?

Functional, black-box testing can be recorded using a table like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test no. | Test Data | Explanation | Expected Result | Actual Result |
| 1 | Shift=3  Plaintext=”cat” | **Valid** data | Outputs “fdw’ | As expected |
| 2 | Shift =q | **Invalid**, Non-numeric data | Fatal error – not handled |  |
| 3 | Shift =3  Plaintext=”xyz” | **Boundary** data | Outputs “abc” |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

### Task

1. Add additional, appropriate tests to the above table for the CaesarBasic program.
2. Make your suggested changes and re-test it.

## Homework

**Read**

* Pages 41 and 42 in the next chapter of this booklet “Base Converter – Introducing Functions”

## Self-Test: bring your answers to the next lesson

|  |
| --- |
|  |

1. Write a loop to display a string called *theword* in reverse order.

C:\TEMP\Temporary Internet Files\Content.IE5\YTVM5DIR\MC900078622[1].WMF

1. #Study the Base Converter program on the next page. Write down the names of:

|  |  |
| --- | --- |
| * 1. a global variable |  |
| * 1. a function |  |
| * 1. a local variable |  |

1. Explain briefly how you could modify the program to keep on displaying the menu until the user enters a “0” to indicate they want to quit.

|  |
| --- |
|  |

# Base Converter – Introducing Functions

## http://t3.gstatic.com/images?q=tbn:ANd9GcQWSOAomNMfK8gSlfmO-d-wTMCDlYahwAVqeZ6LzEsmrd871H2bsJax1OHAObjectives

* Reinforce understanding of binary and decimal number bases.
* Revisit algorithms to convert between number bases.
* Learn how to use **functions** to split programs up into sections.
* Understand **why** the use of functions makes programming easier
* Learn how a Hierarchy Chart can show the structure of a program

**The Problem**

Write a program to convert a decimal number to binary and vice versa.

**The Program**

Here is a program to convert from **decimal to binary**, using the algorithm you met at the start of the course:

namespace DectoBin

{

class DecToBin

{

static void Main(string[] args)

{

Console.WriteLine("Enter the decimal integer to be converted: ");

int decimal = Convert.ToInt32(Console.ReadLine());

int bit;

string result = "";

while (decimal > 0)

{

bit = decimal % 2;

result = bit.ToString() + result;

decimal = decimal / 2;

}

Console.WriteLine("The binary equivalent is " + result);

Console.ReadLine();

}

}

}

Your teacher will explain the main features of the program, especially the use of **functions** to break it up into sections. Also known as **sub-routines, functions and procedures** in other languages, functions help to divide your code into sensible, manageable chunks, each one dedicated to a specific task. These “chunks” can then be **called** by different parts of your program. When designing a program, break the problem down into its constituent tasks and assign each one to its own function.

**Notes:**

1. Give the function a sensible identifier which describes what it does, this significantly helps in debugging any problems later – make your code **self-documenting**.
2. Each function is like a mini-program, i.e. each can declare its own **local variables** for use within that function (but not available to the rest of the program). Variables declared in the main program block, like *choice* in the program above, are called **global** variables. It is very important that you keep the number of global variables to a minimum and use local variables whenever possible. Using local variables:

* makes programs more readable and easier to maintain
* makes functions more portable, that is easier to reuse in other programs
* saves memory, because functions and their local variables only take up memory when they are executing.

## The binary to decimal algorithm

Here is pseudocode for converting a number from binary to decimal (this is essentially the same as the one you met at the start of the course):

input binary

decimal =0

power= 1

for i= length(binary)-1 down to 0

{

if binary[i]=”1” then

decimal = decimal + power

power = power\*2

}

## Hierarchy Charts

A Hierarchy Chart can be used to show graphically the structure of a program. This can be used as a design tool to help the programmer decide how to break down the program into functions. Once the program is written the chart illustrates how the functions interact to make up the complete program.

Here is a simple Hierarchy Chart for the base converter:

Base Converter

Menu

DecToBin

BinToDec

## Tasks for Lesson

The Hierarchy Chart above shows a function that has not yet been written: to convert a number from binary to decimal.

1. add a function to convert from binary to decimal, using the algorithm on the previous page
2. add a choice to the menu to include your new function
3. call your function from the main program
4. add code to repeat displaying the menu until the user enters “0”.

### Extra Exercise:

* Redesign and code your Guessing Game program to use functions.

## Homework

**Read**

* <https://www.tutorialspoint.com/csharp/csharp_arrays.htm>
* The next chapter of this booklet “Bubblesort – Introducing Arrays”

#### Self-Test: bring your answers to the next lesson

1. An array of five integers is declared and initialised as follows:

int[] nums = new int[5]{1,2,3,4,5};

Write a loop to double the numbers in the array.

|  |
| --- |
|  |

1. Write a loop to double the value of each number in this 2-dimensional array:

int[,] Nums2D = new int[5,2] {{1,6}{2,7}{3,8}{4,9}{5,10}};

|  |
| --- |
|  |

# Bubblesort – Introducing Arrays

## C:\TEMP\Cache\Content.IE5\OCUUZSUD\MC900438788[1].jpgObjectives

* Experience the **bubblesort** algorithm!
* Learn how **arrays** can be used to store multiple items of data
* Gain more practice in using functions

**The Problem**

Write a program to sort 100 numbers into numerical order.

**The Algorithm**

You will learn how the bubblesort algorithm works by acting it out.

**The Program**

The program below sorts 100 numbers stored in an **array**.

using System;

namespace BubbleSortNoFunc

{

public class BubbleSortNoFunc

{

public static void Main(string[] args)

{

// initialise the array in reverse order:99 to 0

int[] numbers = new int[100];

int current;

int temp;

bool swapped;

for (int i = 0; i <= 99; i++)

{

numbers[i] = 100 - i;

}

// display the array on the screen

Console.WriteLine("Initial array contents are:");

for (int i = 0; i <= 99; i++)

{

Console.Write(Convert.ToString(numbers[i]) + " ");

}

Console.WriteLine();

//bubblesort algorithm

current = 99;

do

{

swapped = false;

for (int i = 0; i <= current - 1; i += 1)

{

if (numbers[i] > numbers[i + 1])

{

/// swap

temp = numbers[i];

numbers[i] = numbers[i + 1];

numbers[i + 1] = temp;

swapped = true;

}

}

current = current - 1;

}

while (!(swapped == false));

// display the sortedarray on the screen

Console.WriteLine();

Console.WriteLine("Array sorted in ascending order is:");

for (int i = 0; i <= 99; i += 1)

{

Console.Write(Convert.ToString(numbers[i]) + " ");

}

Console.ReadLine();

}

}

}

## Arrays

So far you have worked mainly with *simple* data types, such as integer, real and char. As programs become more complex, it becomes essential to store larger amounts of data and it is impractical to use simple types to do so. For example, you might need to store and process the marks for a class of 20 students over a series of 5 assignments. It is not practical to use 100 separate integer variables to do this: what would you call them??, how would you carry out the same operation on each??

An **array** is the most common and useful structured data type. You can use it to store many items of data as long as they are of the same type: for example an array could store 100 integers or 2000 reals. Structured data types such as arrays are **objects** in C#. We will learn much more about objects in the object-orientated programming module. But for now this means that arrays must be declared in a different way from simple data types such as integers.

An array is declared like this:

int[] MarksArray = new int[100];

// declares an array called MarksArray to hold 100 integers

Array **elements** are always indexed from zero in C# so MarksArray[0] would be the first integer in the array and MarksArray[99] the last. Remember this when you are looping though an array so that you don’t try and index outside the array. Most operations on arrays are done on a single element:

MarksArray[1]= 45; //assign 45 to element number 1

Max = HeightArray[4]; //assign the value of element 4 to Max

You **can’t** assign values to all the elements at once:

MarksArray = 45 //**not allowed**

However, it is very common to want to perform the same operation to every element in an array. How do you do it? By using a loop, typically a for loop. For example, suppose we want to set all the marks in the array to 0:

for (int n = 0; n<100 ; n++) //note declaration of n in the for statement.

{

MarksArray[n] = 0; //assign 0 to element n

}

### Two-dimensional arrays

It is also possible for arrays to have multiple dimensions. For example, here is the declaration for a 2D array:

int[,] Grid = new array[10,10];

Elements are accessed like this:

Grid[4,6] = 55;

To traverse (visit each element) in a 2D array you need to use **nested for loops**:

for ( int col = 0; col < 10 ; col++)

{

for ( int row = 0; row < 10 ; row++)

{

Grid[col,row] = 100;

}

}

Two- and three-dimensional arrays are quite common, for example in games and graphics programming. You can also create arrays with 4, 5… dimensions but these are much less common and rather mind-blowing.

### Arrays and Lists

A problem with traditional arrays is that their sized is fixed (**static**). The MarksArray declared above has space to store 100 marks and no more. What happens if the size of the class increases and more marks need to be stored? Also, a static array always takes up the same amount of space even if it is empty: not very efficient!

In C# we use a **List** data structure when the number of elements we want to store is not known in advance. You will cover lists in the Data Structures module next year.

## Tasks

* Improve the structure of the bubblesort program by breaking it up into functions as illustrated by this **Hierarchy Chart**:

BubbleSort program

InitialiseArray

DisplayArray

BubbleSort

Swap

NB:

* + The DisplayArray function will be called **twice** by the main program.
  + The Swap function is called by the BubbleSort function, not by the main program.
  + Remember to re-organise your program to use local variables whenever possible.

## Extra Exercises

* Remember the Card Game program (page 17)? Rewrite it so that:
  + the same card cannot be drawn twice.
  + There are options to display the whole pack: both in order and randomly.

Hint: use an array to record which cards have already been drawn. What is the most appropriate data type for the array?

C:\TEMP\Cache\Content.IE5\4TNAPH14\MC900241151[1].wmf

* Write a basic Battleships program:
  + Use a 2D array to represent the “ocean” and the ships on it
  + The player(s) guess(es) the location of the ships
  + You could extend this to use ASCII graphics to display the ocean

## Homework

**Read**

* The next chapter in this booklet: “Selection sort – Re-using code with arguments”

#### C:\TEMP\Temporary Internet Files\Content.IE5\YTVM5DIR\MC900078622[1].WMFSelf-Test: bring your answers to the next lesson

1. A function, OutputSymbols, is required which will take in two arguments: an integer n and a character symbol. The function is to display, on the same line, the symbol n times.
   1. Write the code for this function.

|  |
| --- |
|  |

* 1. Write the full code for this function to display 10 exclamation marks.

|  |
| --- |
|  |

1. A function, *Sort*, is required, which will take in a single argument: an array of 100 integers *Nums.* The function will sort the array and return the sorted array to the main program.
   1. Write the code for this function.

|  |
| --- |
|  |

* 1. Write the full code for this function to sort an array called *List*.

|  |
| --- |
|  |

# Selection Sort – Re-using Code with Arguments

## Objectives

* Learn how to use **arguments** to supply data to functions
* Appreciate **why** arguments make functions much more flexible and re-usable
* Understand the key features and advantages of **Structured Programming**.

**The Algorithm**

You will act out the SelectionSort algorithm in class. Outline pseudocode for the algorithm looks like this:

Initialise the array into reverse order

For i =0 to 99

{

Find the smallest number in the array starting at position i

Swap the smallest number with the one at position i

}

**The Problem**

Like BubbleSort, SelectionSort includes the swapping of two values in the array. It would be good to have a general purpose Swap function that we could use in BubbleSort and SelectionSort programs, not just because it would save time but because we know it works and if we re-write it we might get it wrong.

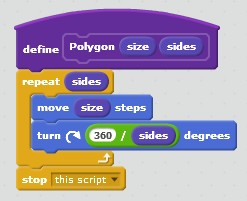
However, there is a problem:

* The Swap function in BubbleSort swaps Numbers[i] with Numbers [i+1].
* For the SelectionSort, we want to swap Numbers[i] with Numbers[smallest]

This time we are going to use **general-purpose** Swap function which will swap **any** two variables. We can achieve this by using **arguments**.

## Scratch recap

We solved this problem in Scratch by adding “inputs” to our blocks:



The size and sides inputs were used to specify the exact polygon we wanted.

**Arguments** in C# are roughly equivalent to (but more powerful than) these “inputs” in Scratch.

## Arguments

Arguments are the crucial step in making sub-routines re-usable. They are a special sort of local variable, declared in the header of the function.

Arguments are used to pass information between the main program and the function or function. In other words, they act as an **interface** between the main program and the routine. The list of arguments is sometimes called the interface.

### Value and Variable Arguments

There are two types of argument:

* **in-only** or **value** arguments allow the main program to pass data in a variable to the function, but this variable cannot be changed by the function. Instead the function creates its own local copy of the data and works on that. The So value arguments provide a **one-way** flow of data from the main program to the function. This is the most common way of passing data to a function.
* **in-out** or **reference** arguments allow the main program to pass a variable to the routine, and any changes made to the variable are **passed back** to the main program variable. Here the function works on the variable itself rather than a local copy. So variable arguments provide a **two-way** flow of data between the main program and the routine. These are known as **reference arguments**, because the address (reference) of the global variable is passed to the function.

### Examples from the Selection Sort program on the next page

a and b are **reference** arguments because the global variables a and b must be changed

Swap function:

public static void Swap(**ref** int a, **ref** int b);

For now don’t worry about the meaning of the public static void part of the function. We cover the *void* part below, while the *public static* is part of C#’s object orientated functionality that we will cover in the Object-Orientated Programming Module.

The header for the FindSmallest function:

Smallest is a **reference** argument because its value will be changed by the function

Nums and i are **value** arguments: no changes are made to these variables by the function

public static void

FindSmallest(int[100] Nums, int i, **ref int** smallest);

By default, arguments are value (in-only). To make them reference (in-out) you have to put **ref** in front of the name.

Here is the function call in the main program:

FindSmallest(Numbers,i,**ref** smallest);

Note that when the function is called, the data types of the arguments are **not** specified, because the variables have already been declared.

## The Program

Here is a program to implement SelectionSort, including several functions which use **arguments**.

using System;

namespace SelectionSort

{

class SelectionSort

{

public static int[] Numbers = new int[100]; //declare array as global

static void Main(string[] args)

{

int smallest = 0;

InitialiseArray();

Console.WriteLine("Initial array contents are:");

DisplayArray();

//doing selection sort ion main program

//work through the array, finding the smallest and doing the swaps

for (int i = 0; i < 100; i++)

{

FindSmallest(Numbers, i, ref smallest);

Swap(ref Numbers[smallest], ref Numbers[i]);

}

Console.WriteLine("\n\nArray sorted in ascending order is:");

// \n means new line

DisplayArray();

Console.ReadLine();

}

public static void FindSmallest(int[] Nums, int i, ref int smallest)

{

/\* find which element in the array Nums is the smallest, starting at Nums[i]. On completion, smallest will be the index of the smallest value \*/

smallest = i;

for (int current = smallest; current < Nums.Length; current++)

{

if (Nums[current] < Nums[smallest])

{

smallest = current;

}

}

}

public static void Swap(ref int a, ref int b)

{

//swap the values of a and b

int temp = a;

a = b;

b = temp;

}

public static void InitialiseArray()

{

/// initialise the array in reverse order:99 to 0

for (int n = 0; n <= 99; n++)

{

Numbers[n] = 99 - n;

}

}

public static void DisplayArray()

{

/// display the array on the screen

for (int n = 0; n <= 99; n++)

{

Console.Write(Convert.ToString(Numbers[n]) + " ");

}

}

}

}

## Structured Programming

The term “structured programming” is often applied to a (desirable) style of programming with the following features:

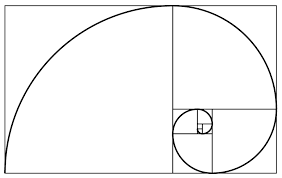
* use of subroutines (functions and functions) to break the overall problem down into sub-problems. Each subroutine tackles a specific, well-defined sub-problem.
* use of local variables wherever possible. Global variables should be used sparingly. In particular, subroutines should not modify global variables as this can easily lead to hard-to-trace bugs (“side effects”).
* use of arguments to pass data between subroutines and the main program: this reduces the need to use global variables and makes re-use of subroutines much easier.
* use of for, while and repeat loops with single exit points. Avoid jumping out of loops half-way through. All exit conditions should be specified in the while or until expression.
* Use of meaningful identifiers for variables and subroutines.
* Use of indentation and white space to clarify program structure.

The above list is not definitive or comprehensive; programmers differ in what they mean by structured programming. You should try to use these techniques in the programs you write.

## Lesson Task

1. Replace the Swap function in the BubbleSort program (which has no arguments) with the parameterised one from SelectionSort. You will need to modify the Bubblesort program to send the correct arguments to Swap.
2. Add arguments to the other functions in BubbleSort: InitialiseArray, DisplayArray and BubbleSort. Think carefully about whether they should be value or variable arguments.

### Extra Exercise:

* [](http://www.google.co.uk/imgres?imgurl=http://upload.wikimedia.org/wikipedia/commons/9/93/Fibonacci_spiral_34.svg&imgrefurl=http://commons.wikimedia.org/wiki/File:Fibonacci_spiral_34.svg&h=578&w=914&tbnid=heVAmOoB-wzVcM:&zoom=1&q=fibonacci&docid=xsSV6h42yM-E1M&hl=en&ei=c49cVdWOFY6y7Qbk4oG4Dg&tbm=isch&ved=0CB8QMygAMAA)You have probably come across the Fibonacci sequence:

1 1 2 3 5 8 13 21 34 …

Each term is the sum of the two previous terms. The first two terms are defined as being 1.

* Write a function to generate the *i*th term of the Fibonacci sequence. The value of i must be passed in using an argument.

## Homework

**Read**

* The Advance Reading section in the next chapter in this booklet “Introducing Functions”
* “Bigger and Better” by Paul Curzon: <http://www.dcs.qmul.ac.uk/%7Epc/research/education/puzzles/reading/cwcch7.pdf>

### C:\TEMP\Temporary Internet Files\Content.IE5\YTVM5DIR\MC900078622[1].WMFSelf-Test: bring your answers to the next lesson

1. A function, *double*, is required which will take in an argument, *r*, of type real, double it and return the result.
   1. Write the **header** for the function

|  |
| --- |
|  |

* 1. Write the **call** to the function to double the variable, *num*, and assign the result to the variable *doublenum*.

|  |
| --- |
|  |

# Using Functions to return a value

## Objectives

* Learn how to use **functions**.

## C:\TEMP\Temporary Internet Files\Content.IE5\0IRO0WZL\MC900383640[1].WMF Advance Reading: Functions Example: modified base converter

Here is a different version of the base converter program from chapter 6:

The type that the function returns

using System;

namespace baseConverter

{

class baseConverter

{

static string decToBin(int dec)

{

The argument that is passed to the function, and its type

int bit;

string result = "";

while (dec > 0)

{

bit = dec % 2;

result = bit.ToString() + result;

dec = dec / 2;

}

return result;

}

static int menu()

{

Console.WriteLine("Welcome to the base converter");

Console.WriteLine("Menu: please enter your choice:");

Console.WriteLine("1. Decimal integer to binary");

Console.WriteLine("9. Quit");

return Convert.ToInt32(Console.ReadLine());

static void Main(string[] args)

{

int choice;

The result (output) of the function

do

{

choice = menu();

if (choice == 1)

{

Console.Write("Enter the decimal you would like to convert to binary: ");

int num = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("The equivalent in binary is " + decToBin(num));

Console.ReadLine();

}

} while (!(choice == 9));

}

}

}

This program introduces the concept of functions **returning a value** to the code that called them. Typically we use this to pass the result of the function’s operations back to the code that called the function. The data type that the function returns is declared in its header. So, the DecToBin function above produces a string as its output. The **return** instruction is used to set the data to be returned. Note that the data is returned **by value** (see page 54). When the function finishes execution, whatever is stored in the **return** variable is returned to the calling program as the value of the function.

Now look at how the function is used (“called”) in the main program:

Console.WriteLine("The equivalent in binary is " + **decToBin(num)**);

Similarly, a function which returned an integer result would be used just as if it were an integer variable. This is different from the way functions are called. Look back and compare.

**int** dec

37

‘100101’

**return** string

DecToBin

## Lesson Tasks

Improve the base converter program:

1. Modify the DecToBin function to allow the user to specify conversion from decimal to **any base** between 2 and 9. To do this, add an Argument to the function. The argument should be used to pass the required base to the function. You will have to add some statements to the main program, asking the user for the required base.
2. Harder: extend the DecToBin function to accept any base up to 16.
3. Add a BinToDec function, based on your earlier base converter program. The function should accept a string as an argument and return an integer.

## Extra Exercises:

1. Write a program that simulates coin tossing. For each toss of the “coin” the program should print **heads** or **tails**. Let the program toss the coin 100 times, and count the number of times each side of the coin appears. Print the results (to screen). The program should call a separate function **Flip** that takes no arguments and returns **0** for tails and **1** for heads.
2. Write a program that can add fractions, finding the appropriate common denominator, doing the addition, and then reducing the fraction to the lowest terms. If the user enters 2,3,4,5 for the fractions 2/3 and 4/5 respectively the output will be:

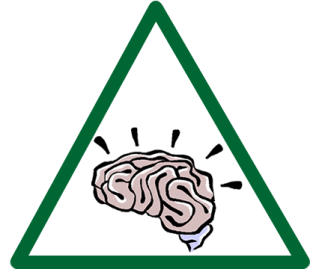
# Assignment 2 – Palindrome Tester

## <http://www.erikahammerschmidt.com/erika/palindrome003.jpg>The Problem

Write a program which inputs a string and reports on whether or not it is a palindrome. Spaces and letter cases (upper or lower) should be ignored.

Example: “Dennis and Edna sinned” is a palindrome, “banana” is not a palindrome.

## The Algorithm



You will discuss this in your triangles. There are at least two alternatives.

## Task

Write a Palindrome Tester, using a function for the core of the program. The function should accept a string as an argument and return a boolean as its result.

**Hand in:**

* Commented source code
* Screen-shots of functional tests

**Extensions**

* Implement **both** the algorithms discussed in class and provide a menu for the user to choose which to use.
* Comment on their relative efficiencies.

# Assignment 3 - Anagrams

## Objectives

This exercise will consolidate and reinforce what you have learned about programming so far, including:

* Sorting algorithms
* Basic constructs: selection and repetition
* Functions with arguments
* Testing

## [BIO Logo]Task

Here is a question from Round 1 of the British Informatics Olympiad 2006, a competition which we hope you will enter later this term:

Question 1: Anagrams

Two words are anagrams of each other if they can both be formed by rearranging the same combination of letters. For example, GADGET and TAGGED are anagrams because they both contain one occurrence of each of the letters A, D, E and T, and two occurrences of the letter G.

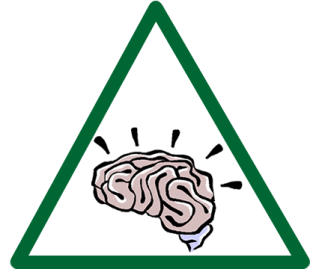
Write a program which inputs two words and then prints **Anagrams** if they are anagrams of each other, or prints **Not anagrams** otherwise. Your program should then terminate.

You have met this problem before and discussed several possible functions of solution:

* “sort and compare”: sort the two strings and compare them
* “letter count”: how many times does each letter of the alphabet appear in each of the two strings?
* “match and cross off”: for each character in the first string, find a match in the second, but make sure each character is only matched once

You are going to use the “sort and compare” function, but do try one of the others if you have time.

**(Sorting) the sorting algorithm**



You are supplied with some pseudocode for the selection sort algorithm below.

**Selection sort**

for i =1 to length(string)

{

Smallest = i

for current = i to length(string)

{

if string[current]>string[smallest] then smallest = current

}

Temp = string[i]

string[i] = string[smallest]

string[smallest] =temp

}

}

Your program will have to perform four sub-tasks:

* Get the words from the user
* Sort each word into alphabetical order
* Compare the two sorted strings
* Output “Anagrams” or “Not Anagrams”

These could be four separate functions or functions, or you might decide that the last two tasks are so trivial that they should be incorporated into the main program block.

**Task 1**

Create a main program skeleton.

**Task 2**

Incorporate the corrected selection sort function(s) into your program.

**Task 3**

Design and write a function or function to get the two strings from the user.

**Task 4**

Complete your program by completing the comparison and output sections. You are allowed to use direct comparison with the equality (=) operator.

**Task 5**

Test your program using the test data from the BIO mark scheme:

|  |  |  |
| --- | --- | --- |
| Input | Expected Result | Actual Result |
| OLYMPIAD  OLYMPIAD |  |  |
| LEMON  MELON |  |  |
| COVERSLIP  SLIPCOVER |  |  |
| TEARDROP  PREDATOR |  |  |
| ABBCCCDDD DDDCCCBBA |  |  |
| I  A |  |  |
| FORTY  FORT |  |  |
| ONE  SIX |  |  |
| GREEN  RANGE |  |  |
| ABBCCCDDD  AAABBBCCD |  |  |

**Hand in:**

* Annotated code
* Completed test plan

**Extensions**

* Implement the alternative algorithms to compare the strings and allow the user to choose which to use.
* Comment on their relative efficiencies.

# Text Files and Re-using code

## C:\TEMP\Cache\Content.IE5\E2I63CXT\MC900431578[1].pngObjectives

* Learn how to use **text files** to store data permanently.
* Learn how to use a **unit** to re-use code.

**The Problem**

Write a program that can encrypt the contents of a text file and store the ciphertext in a new text file. It should also be able to decrypt an already encrypted file, turning it into a file of plaintext.

**The Algorithm**

You are going to re-use the Caesar shift algorithm you developed previously. All you need to add are straightforward algorithms to read and write text files.

## Example: Using Text Files in C#

For many applications, it is essential to store data permanently in files. In this topic, you will learn how to use text files in C# .

A text file is the easiest type of file to use. It stores all data as ASCII characters and has a very simple structure. You can think of it as just a continuous stream of characters. There are special characters to mark the ends of lines but otherwise there is no real structure to it.

The contents of a text file can be read and displayed on screen by a text-editing program such as Notepad. These programs start a new line whenever they encounter an end-of-line character.

The example below is an extremely minimalistic text editor. In fact, it is so simple that we can only read one file and then write new content to it, and only a single line of text at a time. But it shows how to read, write and append to text files in C#.

using System;

using System.IO;

namespace FileHandlingArticleApp

{

class Program

{

static void Main(string[] args)

{

if(File.Exists("test.txt"))

{

string content = File.ReadAllText("test.txt");

Console.WriteLine("Current content of file:");

Console.WriteLine(content);

}

Console.WriteLine("Please enter new content for the file:");

string newContent = Console.ReadLine();

File.WriteAllText("test.txt", newContent);

}

}

}

## Re-using code – using namespaces and classes to create a library of functions

We have already discussed how important it is to **re-use code**, not just to save time or because programmers are lazy but because code that has been tried and tested previously is much less likely to cause problems. Almost all programming languages allow code to be divided into different files to make it easy to re-use. In C# we use **namespaces** to do this. You can put code that you want to reuse into a separate file, using the **namespace** instruction to give that block of code a unique name. In your main code, a **using** instructionwill pull in the code from the separate file as if it is part of the main program. You will have noticed that all the programs so far incorporate a **using system** instruction. This allows us to re-use the code for library functions such as **Console.WriteLine** and **Console.ReadLine**. These functions are stored in a file supplied by Microsoft that contains a namespace called system.

Study the code below, it is in a **namespace** and is not a full program because it has no **main** function. It contains two functions to encrypt and decrypt strings using a Caesar cipher.

Note that because C# is an object orientated language it requires you to define a **class** as well as a **namespace**. We will study classes in the object-orientated module, for now just remember to include it. Note that you also need to declare your functions as **public,** thisensures that they can be accessed from the main program. As before, don’t forget to include the **static** declaration, again this is an object-orientated feature that we will look at later.

Store your code in a .cs file with the same name as the namespace, in this case CaeserCipher.cs

using System;

using System.Text; // needed so that code can access library of text manipulation functions

namespace CaeserCipher

{

class Encrypt\_Decrypt

{

public static string EncryptC(string plaintext, int shift)

{

char letter;

string ciphertext = "";

// encrypt it

// outer loop: step through the plaintext, one character at a time

plaintext = plaintext.ToUpper(); //work in upper case

for (int i = 0; i < plaintext.Length; i++)

{

letter = plaintext[i];

// note the use of square brackets to refer to an individual character within the string

for (int n = 0; n < shift; n++) // inner loop

{

if (plaintext[i] == 'A')

{

letter = 'Z'; // loop back if end of alphabet reached

}

else

{

letter++; // move the ith letter on to the next in the ASCII table,

} // ++ is short for 'add one'

}

ciphertext += letter; //This appends the shifted letter to the ciphertext

}

return ciphertext;

}

public static string DecryptC(string ciphertext, int shift)

{

char letter;

string plaintext = "";

// encrypt it

// outer loop: step through the ciphertext, one character at a time

ciphertext = ciphertext.ToUpper(); //work in upper case

for (int i = 0; i < ciphertext.Length; i++)

{

letter = ciphertext[i];

// note the use of square brackets to refer to an individual character within the string

for (int n = 0; n < shift; n++) // inner loop

{

if (ciphertext[i] == 'Z')

{

letter = 'A'; // loop back if end of alphabet reached

}

else

{

letter--; // move the ith letter on to the previous in the ASCII table,

} // -- is short for 'minus one'

}

plaintext += letter; //This appends the shifted letter to the ciphertext

}

return plaintext;

}

}

}

**The provided functions EncryptC and DecryptC contain logic errors. You will need to correct them in order to get a correct result.**

For your file encryption program, instead of copying and pasting the Caesar Shift code you are going to use the code stored in the namespace file. First you need to say where the Caser cipher functions are by using the namespace

using CaeserCipher;

To call your functions you need to include the namespace and class. For instance, if your class was Encrypt\_Decrypt, you would call the encrypt function as follows:

Console.WriteLine("The encrypted message is " + **CaeserCipher.Encrypt\_Decrypt.EncryptC(plaintext, shift)**);

Here plaintext and shift are the arguments passed to the function.

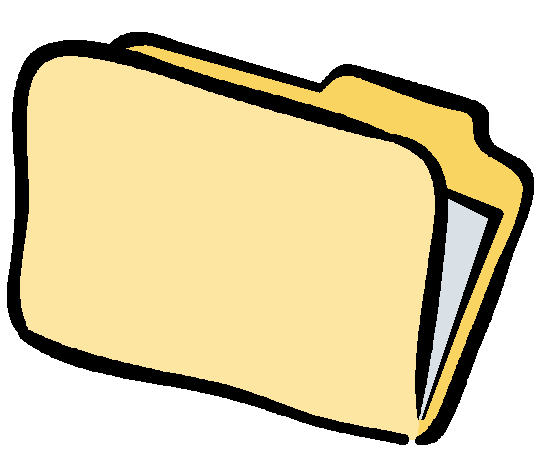
Now you can use the EncryptC and DecryptC functions in your program, just as if you had written them in there yourself.

Note that the functions are reusable because:

* They don’t use any global variables, only local variables and arguments.
* They don’t interact with the user, no ReadLines or WriteLines. This means they could be used in a GUI (Graphical User Interface) program as well as a text-based console one.

## Tasks using text files:

### File analyser

Write a program which reads a text file and reports:

* How many characters it contains
* How many lines it contains
* How many words it contains
* Other statistics of your choice

### Anagrams:



You are provided with a dictionary file containing words in alphabetical order, one per line.



Write a program which inputs a word and then outputs a list of all the words in the dictionary file which are anagrams of that word.

### Searching a File:

Write a program to search for a user-specified string in the file. The program should display the line containing the string or report that the string is not present in the file. Extension: display the whole file, highlighting each occurrence of the word.

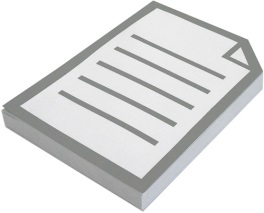
# Assignment 4 - File Encryption



Write a program that can encrypt the contents of a text file and store the ciphertext in a new text file. It should also be able to decrypt an already encrypted file, turning it into a file of plaintext. It must use the functions in the UEncrypt unit to perform the encryption and decryption.

## Extensions – alternative encryption algorithms:

**Polyalphabetic substitution** A "Vigenere"-type cipher. In this case the key is a string, rather than an integer. The characters of the key are used in turn to provide the shift value for the plaintext. For example, if the key is “dog', then the shift values are 4 ('d'), 15 ('o') and 7 ('g') applied in turn. So the first character in the plaintext is shifted 4 places, the second 15, the third 7, the fourth 4, the fifth 15 and so on. Ideally, make the functions/functions you write to do this re-usable and store them in UEncrypt.

****

**Vernam cipher (One Time Pad)** A further extension of the principle of the Vigenere cipher in which the key is an entirely random, non-repeating sequence of characters the same length as the message. This cipher is provably unbreakable.

**You will learn more about these encryption functions in the Data Representation lessons**

**Hand in:**

* Source code for main program and the UEncrypt unit if you have added to or modified it
* Completed Test plan with sample screenshots of your program working

# Records and Binary Files- Address Book

## [File:Address Book Icon.png](http://upload.wikimedia.org/wikipedia/en/1/13/Address_Book_Icon.png)Objectives:

* Learn about structures and their uses
* Learn about binary files, how they differ from text files and gain practical experience in using them

## Structures and Records

The main reason to work with structures is to create records that contain custom data. You use these custom data records to hold complex information. It’s easier and faster to pass a single structure than it is to pass a collection of variables. In the assignment below we use a structure to group together four strings containing firstname, surname, email and phone number of a person. This is declared as follows (ignore the ‘public’ for now, we’ll cover that later):

struct ContactDetails

{

public string firstname;

public string surname;

public string email;

public string phone;

}

A struct object can be created with or without the new operator, same as primitive type variables.

When you create a structure object without using new keyword, you must assign values to each member before accessing them, otherwise it will give a compile time error.

ContactDetails contact;

Console.Write(contact.firstname); // Compile time error

contact.firstname = "Ian";

Console.Write(contact.firstname); // prints "Ian"

With new (new creates an object, object-oriented paradigm will be cover later)

ContactDetails contact = new ContactDetails();

Console.Write(contact.firstname); // prints ""

## Binary Files

A binary file is a file that contains information stored only in form of bits and bytes.(0’s and 1’s). They are generally faster and smaller than text files but are not human readable. A binary file has to be read by specific programs to be useable. For example, a Microsoft Word binary file can only be read by the Word application. Try opening a binary .doc/.docx Word file in a text editor such as notepad and marvel at the garbage it produces!

In the assignment below we write a ContactDetails structure to a file and then read it back. The code for writing and reading is below:

Writing:

static void WriteBinaryFile(ContactDetails contact)

{

using (BinaryWriter writer = new BinaryWriter(File.Open("binaryfile.bin", FileMode.Create)))

{

writer.Write(contact.firstname);

writer.Write(contact.surname);

writer.Write(contact.email);

writer.Write(contact.phone);

}

}

Reading:

static ContactDetails ReadBinaryFile()

{

using (BinaryReader reader = new BinaryReader(File.Open("binaryfile.bin", FileMode.Open)))

{

//first two strings in file are first name and surname

ContactDetails contact;

contact.firstname = reader.ReadString();

contact.surname = reader.ReadString();

contact.email = reader.ReadString();

contact.phone = reader.ReadString();

return contact;

}

}

Don’t worry about the syntax of the ‘using’ statements, you do not need to remember this for the exam!

## Assignment

A student wants to create a computerised address book, containing contact details of their friends.

You are provided with a, not very useful, address book program, which stores just one address in a binary file. Study it carefully and make the following improvements.

1. Extend the ContactDetails structure to support the extra data you have identified in your Triangles, for instance to store a postcode or a salutation (Mr/Ms/Mrs, etc)
2. Modify the Enter Contact and DisplayContact functions to deal with your new record structure.
3. Ad a feature to allow the user to enter the filenames for saving and reading.
4. Extend the email verification function, for instance to check that there is a ‘dot’ character as well as an @ character.
5. Add a phone number verification function to check that it contains only numbers or spaces.
6. Any other useful features you can think of.

We could make this program more useful by using advanced data structures such as a List or Dictionary to store lots of structures in the file. We will cover Lists and Dictionaries in Year 13.

# Appendix 1 – Useful Library Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Mathematical routines** | | | |
| Name | Namespace | What it does | Example |
| **double Truncate**(double x) | Math | Returns, as a double, the whole number part of a double value. In other words, it rounds towards zero. | Math.Truncate(56.53) returns 56.0 |
| **int Round**(double x) | Math | Returns the nearest integer. If the fractional part is 0.5 then the even integer is returned. | Math.Round(56.53) returns 57  Round(56.5) returns 56 |
| **int Floor**(double x) | Math | Rounds towards –ve infinity. | Math.Floor(-1.9) returns -2 |
| **int Ceiling**(double x) | Math | Rounds towards +ve infinity | Math.Ceiling(-1.9) returns -1 |
| **double Pow**(double base, double exponent) | Math | Raises base to exponent | Math.Pow(2,4) returns 16.0 |
| **Random rand = new Random**  (class) | System | Creates an instance (object) of the random number generator | Call this once before calling Random or RandomRange |
| **int Random.Next**(int range)  (method) | System | Returns a random integer between 0 and range-1 | Call using the object you created: rand.Next(100) returns a random integer between 0 and 99 |
| **int Random.Next**(int from,To) | System | Returns a random integer between From and To-1 | Rand.Next(1,101) returns a random integer between 1 and 100 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Conversion routines** | | | |
| Name | Unit | What it does | Example |
| **int32 Convert.ToInt32**(char ch) | System | Converts a character into its ASCII code | Convert.ToInt32(‘A’) returns 65 |
| **char Convert.ToChar**(int x) | System | Converts an ASCII value into the character | Convert.ToChar(65) returns ‘A’ |
| **string Convert.ToString**(int Value) | System | Converts an integer to a string | Convert.ToString(36) returns “36” |
| **Int32 Convert.ToInt32**(string s) | System | Converts a string to an integer | Convert.ToInt32(“36”) returns 36  StrToInt(‘bob’) raises an exception |
| **string Convert.ToString**(double Value) | System | Converts a double (floating-point) value to a string | Convert.ToString(56.43) returns “56.43” |
| **double Convert.ToDouble**(string s) | System | Converts a string to a double | Convert.ToDouble(“56.43”) returns 56.43 |
| **string Convert.ToStr**(DateTime DT) | System | Converts a DateTime to a string | Convert.ToString(DateTimeNow) returns the current date and time as a string, similar to this: “27/06/2017 12:09:03” |
| **DateTime Convert.ToDateTime(string** s) | System | Converts a string to a DateTime | If the string cannot be converted to a valid date then an exception is raised |

|  |  |  |  |
| --- | --- | --- | --- |
| **String-handling routines (Assume the string animal == “sheep”)** | | | |
| **int s.Length** | System | Returns the length of s | animal.Length returns 5 |
| **string s.Substring**(int index, int length) | System | Returns a new string, containing length characters, starting at index | animal.SubString( 2,3) returns “eep” |
| **int s.IndexOf**(string substr) | System | Returns the index of the first occurence of substr in S  Returns -1 if it’s not present. | Animal.IndexOf(“he”) returns 1 |
| **string s.Insert**(int index, string value) | System | Returns a new string with value inserted into s at position Index. | Animal.Insert(0,”my” returns “mysheep” |
| **String s.Remove**(int Index, int Count) | System | Returns a new string with Count characters removed from s, starting with s[Index] | Animal.Remove(2,2) returns “shp’ |

# Appendix 2 – C# Coding conventions

It is important to make your programs easily readable, both by you and others who may have to modify (or mark) it. The following example is based on the standard recommended by several websites eg <https://www.dofactory.com/reference/csharp-coding-standards> . You don’t have to follow it rigidly but be **consistent.** Above all, use **indentation** to clarify the structure of your program: your teacher will not accept unindented code.

static double BinaryToFloat(string bin) *// Use****PascalCasing****for function names*

{

if (!bin.Contains('.'))

{ *//****vertically align*** *curly brackets*

bin = bin + '.';

}

*// use****camelCasing****for arguments and local variables:*

int pointIndex = bin.IndexOf('.');

string wholepart = bin.Substring(0, pointIndex);

double result = binToDec(wholepart);

double fraction = 0.5;

for (int i = pointIndex + 1; i < bin.Length; i++)

{

result = result + fraction \* Convert.ToInt32(bin[i].ToString());

fraction = fraction / 2;

}

return result;

}

Also,

* Do **not** use **SCREAMING CAPS** for constants
* Do **not** use **Under\_scores** in identifiers.

1. As are many other languages, for example Java and C++ . [↑](#footnote-ref-2)