Teaching Guide: Subroutines and structured programming

The first section of this guide will outline some key concepts and terminology for the use of subroutines. The second section will outline the principles of the structured approach to programming and how this links with the topic of subroutines.

There are a number of examples throughout which demonstrate some of the key concepts. Some of the examples also implement subroutines that most languages have built in (such as `LEN`). These have been used because it helps to explain what actually happens when different built-in subroutines are used and is a good way to reinforce both the advantages of subroutines, as well as the abstraction that already exists in programming languages.

All of the example code is provided using the AQA pseudo-code format. At the end of the document are appendices containing the same code examples in all of the supported languages.

Defining and calling subroutines

A subroutine is the term given to a named 'out of line' block of code that can be run from a main program (or main routine) or from another subroutine simply by using the name.

It is good practice, and in some cases essential, to create/define subroutines at the start of your code, which is why they are referred to as 'out of line' or 'out of sequence'.

Example

```
SUBROUTINE printMessage()
   OUTPUT 'This is a message inside a SUBROUTINE'
ENDSUBROUTINE

printMessage()
```

This is a very simple subroutine that displays a message at any point in the program it is called. The subroutine is created (or defined) and given a name, in this case `printMessage`. The subroutine contains code to output a message. Finally, the subroutine is executed when `printMessage` is called.

Passing data into the subroutine with parameters

As seen in the example above, calling a subroutine allows you to reuse code, but there is more that subroutines can do, and using parameters, as explained here, will allow you to reuse sections of code in a more flexible way.

A parameter is a piece of data that we can ‘pass into’ a subroutine when it is called. This allows us to give the subroutine specific data which can then be used within the subroutine. The example below starts to explain how parameters work with subroutines.

```
1    SUBROUTINE addTwoNumbers(num1, num2)
2       total ← num1 + num2
3       OUTPUT total
4    ENDSUBROUTINE
5
6    addTwoNumbers(5, 10)
```

The code above is probably the simplest example of using parameters. In reality, this program is so small that it would not normally be created as a subroutine, but it does help to show how data is passed into them.

In this example a subroutine named `addTwoNumbers` has been created. When the subroutine is called, two pieces of data are passed into the subroutine, in this example the integers `5` and `10`.

When the subroutine is called the first piece of data passed into the subroutine is `5`. When you look at where the subroutine is defined (line 1) you see that the first parameter to be used within the subroutine has been named `num1`. This means that within the subroutine, the value of `num1` when the subroutine starts is `5` for the call `addTwoNumbers(5, 10)`.

```
SUBROUTINE addTwoNumbers(num1, num2)
   total ← num1 + num2
   OUTPUT total
ENDSUBROUTINE

addTwoNumbers(5, 10)
```

This continues now with the second piece of data being passed into the subroutine, `10`. When the subroutine was defined, the second parameter was named `num2`. This means that in the subroutine, the value of `num2` when the subroutine starts is `10` for the call `addTwoNumbers(5, 10)`.

```
SUBROUTINE addTwoNumbers(num1, num2)
    total ← num1 + num2
    OUTPUT total
ENDSUBROUTINE

addTwoNumbers(5, 10)
```

Important note

Often the terms parameters and arguments are used to describe the passing of variables into a subroutine. Technically arguments are the values passed into the subroutine when it is called (in the example above `5` and `10`), and parameters are the variables used by the subroutine when it is defined to hold these values (in the example above `num1` and `num2`). However, for the purpose of AQA GCSE Computer Science, the term parameters will be used to refer to both.

Common mistakes with parameters

It is important that the number of values being passed into the subroutine and the number of parameters given when the subroutine is defined are the same. Both examples below are invalid and would cause an error, either because the subroutine was expecting another value which it didn't get so there are effectively variables without any values, or because the subroutine was only expecting two values yet it is being sent three.

Too few values passed into a subroutine

```
SUBROUTINE addTwoNumbers(num1, num2)
    total ← num1 + num2
    OUTPUT total
ENDSUBROUTINE

addTwoNumbers(5)
```

Too many values passed into a subroutine

```
SUBROUTINE addTwoNumbers(num1, num2)
    total ← num1 + num2
    OUTPUT total
ENDSUBROUTINE

addTwoNumbers(5, 10, 15)
```

Return values and scope

The following code is another example of using a subroutine and then passing in a value to be used inside the subroutine. This example also adds a further feature, namely a return value.

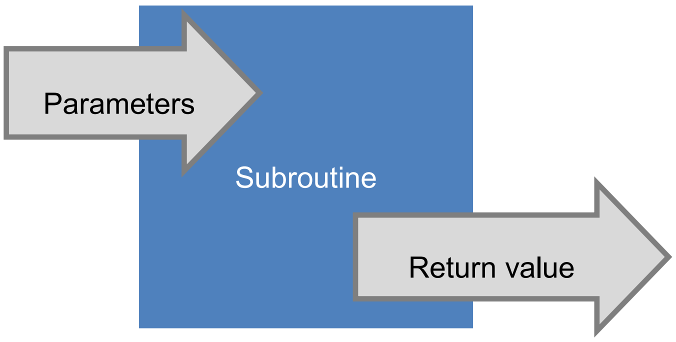
```
SUBROUTINE lenOfStr(string)
   length ← 0
   FOR character IN string
      length ← length + 1
   ENDFOR
   RETURN length
ENDSUBROUTINE

phrase ← 'Computer Science'
phraseLen ← lenOfStr(phrase)
OUTPUT phraseLen
```

This subroutine calculates the length of a string. This is a built-in feature in most modern high-level languages (often called the `LEN` function), but programming your own can be a very useful process that will not only help you to understand the use of subroutines, but also understand that this is happening in the background all the time while programming. The use of subroutines is one way that modern languages abstract detail so that the programmer can focus more clearly on the task in front of them.

In this example, the subroutine `lenOfStr` has one parameter and uses the string value passed to it to determine the number of characters that it has.

This code also shows a further feature of subroutines, return values. As subroutines are self-contained blocks of code, if they need to pass a value back to the calling routine they need a way to do that. The return value is what is passed back to the calling routine when the subroutine finishes; this value can be assigned to a variable, or used within an expression or other statement, for example

```
IF lenOfStr(s) = 0 THEN
   OUTPUT 'The string is empty'
ENDIF
```



In the example code above, the **`RETURN`** statement returns the value of the expression (a simple variable or a more complicated expression) following the **`RETURN`** keyword to the calling routine.

Any variables that are created inside the subroutine will not be visible outside it. The places within the code that the variable can be seen and accessed from is referred to as the 'scope' of the variable.

Local variables

Variables created inside a subroutine are local to that routine. They are said to have 'local scope'. This means they only exist within that subroutine and so can only be accessed from within that subroutine.

If you create a variable inside a subroutine and then try and use that variable from a point in the code that is outside the subroutine what happens will depend on the programming language you are using.

Global variables

The other type of scope that variables can have is 'global scope'. When a variable is created it can be given global scope which means it can be accessed and used from anywhere within the code. While at first this seems like a very useful way of programming, large scale use of global variables usually points to code that has not been very well planned.

Because they can be altered anywhere in the program it can be difficult to find where an erroneous change to a global variable has been made. The widespread use of global variables often results in logic errors that are hard to find and their use is to be discouraged when writing structured, robust and secure programs.

Common errors with return values

There are a number of logic or syntax errors that can be made when using return values, but there are two in particular that are quite common.

Missing RETURN inside a subroutine

In the first example, the return value has been removed from inside the subroutine. The variable `phraseLen` has still been declared but without a return value there is nothing to assign to it. Different languages respond to this error differently, and while some will still run and the mistake would result in a logic error where the program does not do what is expected, others will detect the error when they are compiling and therefore the code will not even run.

```
SUBROUTINE lenOfStr(string)
   length ← 0
   FOR character IN string
      length ← length + 1
   ENDFOR
ENDSUBROUTINE

phrase ← 'Computer Science'
phraseLen ← lenOfStr(phrase)
OUTPUT phraseLen
```

Missing variable to assign the return value to

In this second example, the variable `phraseLen` has been removed, so the return value has nowhere to be assigned. Again, different languages may handle this slightly differently, but in the majority of cases this would result in an error and again the code will not run.

```
SUBROUTINE lenOfStr(string)
   length ← 0
   FOR character IN string
      length ← length + 1
   ENDFOR
   RETURN length
ENDSUBROUTINE
phrase ← 'Computer Science'
lenOfStr(phrase)
OUTPUT phraseLen
```

Note that `lenOfStr(phrase)` can be used directly in the **`OUTPUT`** statement, and that the phrase being passed into `lenOfStr` can passed as the parameter value, eg

```
OUTPUT lenOfStr('Computer Science')
```

would have exactly the same effect as the last three lines of code above.

Subroutines, functions and procedures

Another common misconception with subroutines is the terminology “function” and “procedure”. Both functions and procedures are examples of subroutines, and the three phrases (subroutine, function and procedure) are often used interchangeably in many conversations between programmers and on many programming forums.

However, there is a subtle difference:

- a subroutine is any named ‘out of line’ block of code that can be called from a main program (or main routine) or from within another subroutine.
- a function is a type of subroutine that returns a value.
- a procedure is a type of subroutine that does not return a value.

The examples below show the same piece of code that was used to outline the use of return values and scope, but with the slight modification that outlines the difference between functions and procedures.

Procedure

```
SUBROUTINE lenOfStr(string)
   length ← 0
   FOR character IN string
      length ← length + 1
   ENDFOR
   OUTPUT length
ENDSUBROUTINE

phrase ← 'Computer Science'
lenOfStr(phrase)
```

Function

```
SUBROUTINE lenOfStr(string)
   length ← 0
   FOR character IN string
      length ← length + 1
   ENDFOR
   RETURN length
ENDSUBROUTINE

phrase ← 'Computer Science'
phraseLen ← lenOfStr(phrase)
OUTPUT phraseLen
```

Both of these subroutines are defined, called and have data passed to them using parameters in the same way. The subtle difference is in how the result (the length of the string passed in) is finally handled.

In the procedure a value is not returned, which means the subroutine has to output the value of the `length` variable because the local scope of the variable would mean it could not be accessed from outside the subroutine.

In the function the **`RETURN`** keyword has been used which means the variable `phraseLen` can be assigned the value that is being returned. One advantage of this is that the value stored in `phraseLen` is then available to the rest of the program and can be used in other calculations as required.

Advantages of using subroutines

Using subroutines can help to make programs quicker to write and simpler to modify and debug, and, when implemented well, can make your programs far easier to read and follow.

Reusing code

Once a subroutine has been created (defined) it can be called as many times as needed. When first starting to program it is quite common to come across problems that require you to repeat code. The way most people get around this to begin with is to copy and paste the code they've already written, but this can lead to errors if they change some copies of the code but not the rest.

The more effective way is to create a subroutine and then call this subroutine when required. There is no limit to the number of times a subroutine can be called so this ability to reuse code can be very beneficial.

Simpler to modify

Another advantage of using subroutines is that your code will be easier to change. If you have large sections of duplicated code and need to make a change, then as mentioned above, you need to make sure you have changed the code in every location. Using subroutines allows for a cascading change, so changes to a subroutine means that those changes take effect every time the subroutine is called.

Easier to debug

Using subroutines also makes your code easier to test and debug. When a program is broken into subroutines, you can test each one of these individually. Once thoroughly tested the subroutine can be used knowing that any errors are much more likely to be caused by other code.

Summary of the power of subroutines

The example below shows a summary of the power of using subroutines, and how much more effective and efficient programs are when using them effectively.

This example looks at three different ways of building a simple 3x3 grid, with each space being taken up by `' O '`, as shown in the example output below.

Example output 1

```
O | O | O
O | O | O
O | O | O
```

Example code 1

This first example does not use any subroutines and just builds the grid using a nested for loop to output 3 cells per line on 3 lines.

```
height ← 3
width ← 3
FOR row ← 1 TO height
   FOR column ← 1 TO width
      IF (column MOD 3) = 0 THEN
         OUTPUT ' O '
      ELSE
         OUTPUT ' O |'
      ENDIF
   ENDFOR
ENDFOR
```

Example code 2

If we wanted to output more than one grid in our program, we have two choices: we can copy and paste the code wherever the grid is needed, or we can create a subroutine for the grid and then just call the subroutine when we want to use it.

```
SUBROUTINE grid3x3()
   height ← 3
   width ← 3
   FOR row ← 1 TO height
      FOR column ← 1 TO width
         IF (column MOD 3) = 0 THEN
            OUTPUT ' O '
         ELSE
            OUTPUT ' O |'
         ENDIF
      ENDFOR
   ENDFOR
ENDSUBROUTINE
```

```
# Other Code
grid3x3()
# Other Code
grid3x3()
# Other Code
```

This code has now been placed inside a subroutine. The subroutine is called `grid3x3` and can be called as many times as needed.

If we wanted to make a change such as replacing all of the `Os` with `Xs`, then changing it inside the subroutine means that every time it is called those changes will be seen. If you had just copied and pasted the code then it would be very easy to miss one of these copies.

Example code 3

However, the advantages of using subroutines can go much further than this. Using the example above, we can now create a subroutine that instead of drawing a 3x3 grid, will draw a grid of whatever size we might want. The pseudo-code example below shows how this can be done using parameters to pass data into the subroutine.

```
SUBROUTINE drawGrid(height, width)
   FOR row ← 1 TO height
      FOR column ← 1 TO width
         IF (column MOD width) = 0 THEN
            OUTPUT ' O '
         ELSE
            OUTPUT ' O |'
         ENDIF
      ENDFOR
   ENDFOR
ENDSUBROUTINE

drawGrid(3, 3)
drawGrid(9, 9)
drawGrid(7, 2)
```

In this example, the subroutine has now been renamed to `drawGrid`, and expects two values to be passed into it when it is called. This subroutine can now draw grids of different dimensions.

Structured programming

Structured programming is an approach that combines a range of different skills and techniques to ensure that programs are quicker to write, test, debug and modify, and easier to think about and understand. Structured programming is tightly linked to subroutines because the structured approach requires a programmer to analyse a problem and then decompose it into a series of modular sub-problems (modularised programming). The solution to each one of these sub-problems is then coded as a subroutine, with information passed between these subroutines using parameters and return values.

The following generic example could be developed further and once a coded solution exists, it could be reused in a wide range of other possible programs. This example has been chosen because it is a common requirement in many programs that involve a menu or require the user to be able to choose from multiple options.

> *Create a program that will display a menu to a user, who can then select from four possible options. Once the user has selected one of the options a suitable message should be output and the menu should be displayed again for the user to choose a new option (unless they decided to leave the program). If the user selects an option that is not available to them, they should be told their choice was not valid and that they need to select the option again. The final option should end the program when chosen.*

There are many different ways of solving this problem. Some of them would involve while loops or repeat until loops to manage both the validation and the return to the menu section. However, we will now show how this type of program could be solved using the structured approach.

Decompose

Analyse the problem and break it down into smaller, similar sized areas that need their own solutions. You may come up with a slightly different set of sub-problems, but one possible set would be the following:

1. Show the menu
2. User enters their menu choice
3. Validate the user input
4. Display the relevant message for their chosen option.

Plan and code

Once you have decomposed the problem into different sub-problems you can then create a subroutine for each of these. The code below shows subroutines that have been created as solutions to the sub-problems above.

```
SUBROUTINE showMenu()
   OUTPUT 'Choose a menu option:'
   OUTPUT 'Option A - Select 1'
   OUTPUT 'Option B - Select 2'
   OUTPUT 'Option C - Select 3'
   OUTPUT 'Quit     - Select 4'
ENDSUBROUTINE

SUBROUTINE validateNumber(lowest, highest)
   # This will work for user input that is an
integer
   num ← USERINPUT
   WHILE (num < lowest) OR (num > highest)
      OUTPUT 'Invalid choice, try again'
      num ← USERINPUT
   ENDWHILE
   RETURN num
ENDSUBROUTINE

SUBROUTINE showChoice(num)
   IF num = 1 THEN
      OUTPUT 'You have chosen Option A'
   ELSE IF num = 2 THEN
      OUTPUT 'You have chosen Option B'
   ELSE IF num = 3 THEN
      OUTPUT 'You have chosen Option C'
   ELSE IF num = 4 THEN
      OUTPUT 'You have chosen to Quit'
   ENDIF
ENDSUBROUTINE
```

If we are following this example through, we see that the main program would be very simple:

```
choice ← 0
WHILE choice ≠ 4
   showMenu()
   choice ← validateNumber(1, 4)
   showChoice(choice)
ENDWHILE
OUTPUT 'Thank you for using my program.'
```

When the `showMenu` subroutine runs it will display the possible menu choices. The function `validateNumber` is then called to get the user’s choice. The choice is validated using the two parameters as the minimum and maximum allowed values for the choice. The actual valid choice made is then returned to the calling routine and stored in the variable named `choice` and this value is then passed into the `showChoice` subroutine, which then uses this to display the appropriate option.

Once the `showChoice` subroutine has been completed the program will use the while loop condition to check if the menu needs displaying again.

Appendices

The following appendices contain the code from the pseudo-code examples written in each of the supported languages, C#, Python and VB.NET. In each case, you may need to make slight changes to the code depending on your programming environment. The code is provided by way of example.

Appendix 1 – C# implementations of the pseudo-code examples

Defining and calling subroutines

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        # In C# void declares that the subroutine being defined is a
        # procedure because it does not return anything.
        # The keyword static is needed so that the subroutine can
        # be called in the way shown in the subroutine Main
        static void printMessage() {
            Console.WriteLine("This is a message inside a SUBROUTINE");
        }

        static void Main(string[] args) {
            printMessage();
        }
    }
}
```

Passing data into the subroutine with parameters

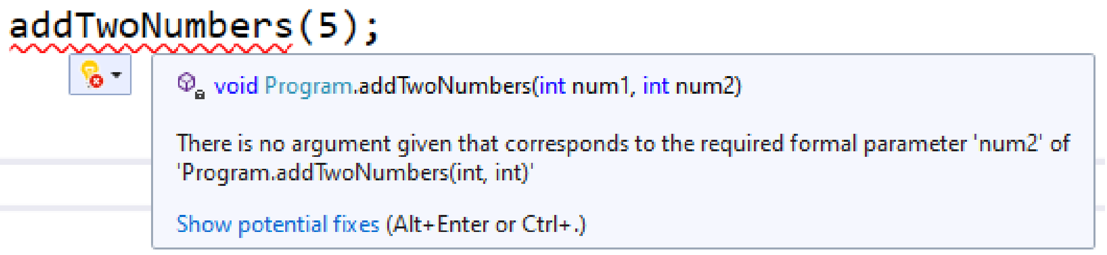
```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        # In C# the type of the parameters comes before the name of the
        # parameter. Here both the parameters are defined to be of
        # the int type (integers or whole numbers).
        # To amend the subroutine so that it can be passed real
        # numbers, eg 25.7, 19.6, etc, change int to single. You can always
        # pass integers to a subroutine expecting floats,
        # but not the other way around.
        static void addTwoNumbers(int num1, int num2) {
            int total = num1 + num2;
            Console.WriteLine(total);
        }

        static void Main(string[] args) {
            addTwoNumbers(5, 10);
        }
    }
}
```

Common mistakes with parameters

Too few values passed into a subroutine

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        static void addTwoNumbers(int num1, int num2) {
            int total = num1 + num2;
            Console.WriteLine(total);
        }

        static void Main(string[] args) {
            addTwoNumbers(5);
        }
    }
}
```

In Visual Studio `addToNumbers` will be underlined in red and hovering the mouse over it will show the following:



```
addTwoNumbers(5);
```

void Program.addTwoNumbers(int num1, int num2)

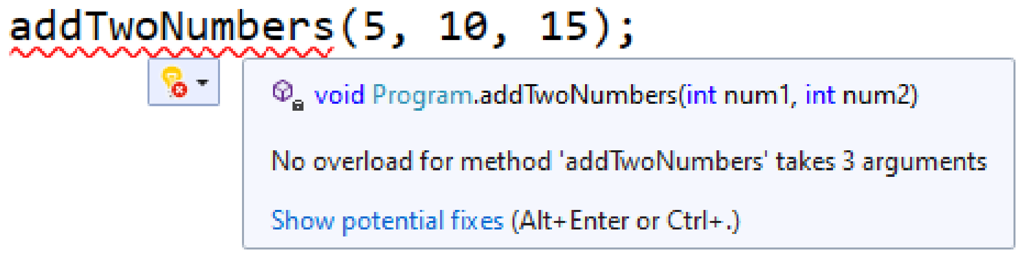
There is no argument given that corresponds to the required formal parameter 'num2' of 'Program.addTwoNumbers(int, int)'

Show potential fixes (Alt+Enter or Ctrl+.)

Too many values passed into a subroutine

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        static void addTwoNumbers(int num1, int num2) {
            int total = num1 + num2;
            Console.WriteLine(total);
        }

        static void Main(string[] args) {
            addTwoNumbers(5, 10, 15);
        }
    }
}
```

In Visual Studio `addToNumbers` will be underlined in red and hovering the mouse over it will show the following:



```
addTwoNumbers(5, 10, 15);
```

void Program.addTwoNumbers(int num1, int num2)

No overload for method 'addTwoNumbers' takes 3 arguments

Show potential fixes (Alt+Enter or Ctrl+.)

Return values and scope

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        static int lenOfStr(string String) {
            int length = 0;
            foreach (char character in String) {
                length = length + 1;
            }
            return length;
        }

        static void Main(string[] args) {
            string phrase = "Computer Science";
            int phraseLen = lenOfStr(phrase);
            Console.WriteLine(phraseLen);
            string s = "";
            if (lenOfStr(s) == 0) {
                Console.WriteLine("The string is empty");
            }
        }
    }
}
```

Notes:

1. Rather than use `void` to indicate that a subroutine is a procedure we give the type of what is returned by the subroutine (its return type) to indicate that this is a function
2. Because `string` is a reserved word in C# we have had to use `String` as the name of the parameter
3. Strings in C# are enclosed with double quotes, eg `"Computer Science"`, not single quotes as used in pseudo-code
4. `length` is a local variable within the subroutine `lenOfStr` and must be defined with its type, in this case `int` for integer

Summary of the power of subroutines

Example 1

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        static void Main(string[] args) {
            int height = 3;
            int width = 3;
            for (int row = 1; row <= height; row++) {
                for (int column = 1; column <= width; column++) {
                    if ((column % 3) == 0) {
                        Console.WriteLine(" O ");
                    }
                    else {
                        Console.Write(" O |");
                    }
                }
            }
        }
    }
}
```

Notes

1. A `for` statement in C# has quite a different look to the `FOR` statement in pseudo-code but they do exactly the same thing
2. Because we want to make each row of three symbols show on a different line we output a new line (using `Console.WriteLine`) after every three symbols which are output using `Console.Write` which doesn't output a new line.

Example 2

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        static void grid3x3() {
            int height = 3;
            int width = 3;
            for (int row = 1; row <= height; row++) {
                for (int column = 1; column <= width; column++) {
                    if ((column % 3) == 0) {
                        Console.WriteLine(" O ");
                    }
                    else {
                        Console.Write(" O |");
                    }
                }
            }
        }

        static void Main(string[] args)
        {
            grid3x3();
        }
    }
}
```

AQA Realising potential

Example 3

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {
        static void drawGrid(int height, int width) {
              for (int row = 1; row <= height; row++) {
                for (int column = 1; column <= width; column++) {
                    if ((column % width) == 0) {
                        Console.WriteLine(" O ");
                    }
                    else {
                        Console.Write(" O |");
                    }
                }
            }
        }

        static void Main(string[] args)
        {
            drawGrid(3, 3);
            drawGrid(9, 9);
            drawGrid(7, 2);
        }
    }
}
```

Notes

1. C# uses `==` to test for two numbers or strings being equal
2. Because the subroutine will no longer **always** produce a grid that is three wide the `if` statement inside the inner loop `if ((column % width) == 0)` has had to be changed to use the width of the grid that the current call wants (you could leave it as it was and see what happens when you run the program)

3. You could make the subroutine even more general so that it could print a grid of `X` characters, or of `@` characters, or any character that you wanted as well as being of any height and width. How?

Plan and code - Menu

```
using System;
namespace TRB_Subroutines_And_Structured_Programming {
    class Program {

        static void showMenu() {
            Console.WriteLine("Choose a menu option");
            Console.WriteLine("Option A - Select 1");
            Console.WriteLine("Option B - Select 2");
            Console.WriteLine("Option C - Select 3");
            Console.WriteLine("Quit     - Select 4");
        }

        static int validateNumber(int lowest, int highest) {
            int num;
            num = Convert.ToInt32(Console.ReadLine());
            while ((num < lowest) || (num > highest)) {
                Console.WriteLine("Invalid choice try again");
                num = Convert.ToInt32(Console.ReadLine());
            }
            return num;
        }

        static void showChoice(int num) {
            if (num == 1) {
                Console.WriteLine("You have chosen Option A");
            }
            else if (num == 2) {
                Console.WriteLine("You have chosen Option B");
            }
            else if (num == 3) {
                Console.WriteLine("You have chosen Option C");
            }
            else if (num == 4) {
                Console.WriteLine("You have chosen to Quit");
            }
        }

        static void Main(string[] args) {
            int choice = 0;
            while (choice != 4) {
                showMenu();
                choice = validateNumber(1, 4);
                showChoice(choice);
            }
            Console.WriteLine("Thank you for using my program");
        }
    }
}
```

Notes

1. Because the C# subroutine `Console.ReadLine` returns a string, C# needs the programmer to convert this to an integer using the `Convert.ToInt32` subroutine
2. C# uses the symbol `||` to mean `OR`

Appendix 2 – Python implementations of the pseudo-code examples

Defining and calling subroutines

```
def printMessage():
    print('This is a message inside a SUBROUTINE')


printMessage()
```

Passing data into the subroutine with parameters

```
def addTwoNumbers(num1, num2):
    total = num1 + num2
    print(total)


addTwoNumbers(5, 10)
```

Common mistakes with parameters

Too few values passed into a subroutine

```
def addTwoNumbers(num1, num2):
    total = num1 + num2
    print(total)


addTwoNumbers(5)
```

In the Python WingIDE `addToNumbers` will be highlighted in red and the following message will be displayed in the exceptions area:

```
builtins.TypeError: addTwoNumbers() missing 1 required positional argument: 'num2'
```

Too many values passed into a subroutine

```
def addTwoNumbers(num1, num2):
    total = num1 + num2
    print(total)


addTwoNumbers(5, 10, 15)
```

In the Python WingIDE `addToNumbers` will be highlighted in red and the following message will be displayed in the exceptions area:

```
builtins.TypeError: addTwoNumbers() takes 2 positional arguments but 3 were given
```

Return values and scope

```
def lenOfStr(string):
    length = 0
    for character in string:
        length = length + 1
    return length


phrase = 'Computer Science'
phraseLen = lenOfStr(phrase)
print(phraseLen)
```

Summary of the power of subroutines

Example 1

```
height = 3
width = 3
for row in range(1, height + 1):
    for column in range(1, width + 1):
        if (column % 3) == 0:
            print(' O ')
        else:
            print(' O |', end='')
```

Notes:

1) Python uses the `%` operator for the `MOD` (remainder) function
2) The `, end=''` element in the print command keeps the output on the same line and does not issue a CRLF

Example 2

```
def grid3x3():
    height = 3
    width = 3
    for row in range(1, height + 1):
        for column in range(1, width + 1):
            if (column % 3) == 0:
                print(' O ')
            else:
                print(' O |', end='')


grid3x3()
# Other code
grid3x3()
```

Example 3

```
def drawGrid(height, width):
    for row in range(1, height + 1):
        for column in range(1, width + 1):
            if (column % width) == 0:
                print(' O ')
            else:
                print(' O |', end='')


drawGrid(3, 3)
drawGrid(9, 9)
drawGrid(7, 2)
```

Plan and code - Menu

```
def showMenu():
    print('Choose a menu option:')
    print('Option A - Select 1')
    print('Option B - Select 2')
    print('Option C - Select 3')
    print('Quit     - Select 4')


def validateNumber(lowest, highest):
    num = int(input('Enter your choice'))
    while (num < lowest) or (num > highest):
        print('Invalid choice, try again')
        num = int(input('Enter your choice'))
    return num


def showChoice(num):
    if num == 1:
        print('You have chosen Option A')
    elif num == 2:
        print('You have chosen Option B')
    elif num == 3:
        print('You have chosen Option C')
    elif num == 4:
        print('You have chosen to Quit')
```

```
choice = 0
while choice != 4:
    showMenu()
    choice = validateNumber(1, 4)
    showChoice(choice)
print('Thank you for using my program.')
```

Appendix 3 – VB.NET implementations of the pseudo-code examples

Defining and calling subroutines

```
Module Module1
    Sub PrintMessage()
        Console.WriteLine("This is a message inside a SUBROUTINE")
    End Sub

    Sub Main()
        PrintMessage()
    End Sub
End Module
```

Notes:

1. In Visual Studio subroutine names start with a capital letter. The code will execute with a lower case letter but will give warnings
2. Strings in VB are enclosed with double quotes, eg `"Computer Science"`, not single quotes as used in pseudo-code

Passing data into the subroutine with parameters

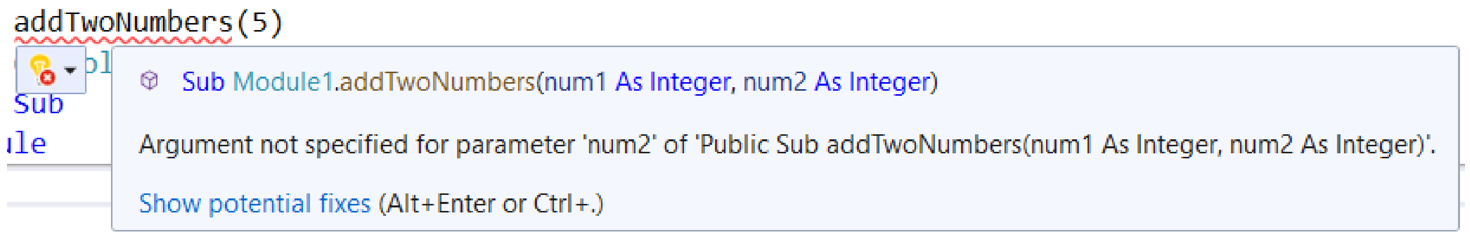
```
Module Module1
    Sub addTwoNumbers(num1 As Integer, num2 As Integer)
        Dim total As Integer
        total = num1 + num2
        Console.WriteLine(total)
    End Sub

    Sub Main()
        addTwoNumbers(5, 10)
    End Sub
End Module
```

Common mistakes with parameters

Too few values passed into a subroutine

```
Module Module1
    Sub addTwoNumbers(num1 As Integer, num2 As Integer)
        Dim total As Integer
        total = num1 + num2
        Console.WriteLine(total)
    End Sub

    Sub Main()
        addTwoNumbers(5)
    End Sub
End Module
```

In Visual Studio `addToNumbers` will be underlined in red and hovering the mouse over it will show the following:



```
addTwoNumbers(5)
```

Sub Module1.addTwoNumbers(num1 As Integer, num2 As Integer)

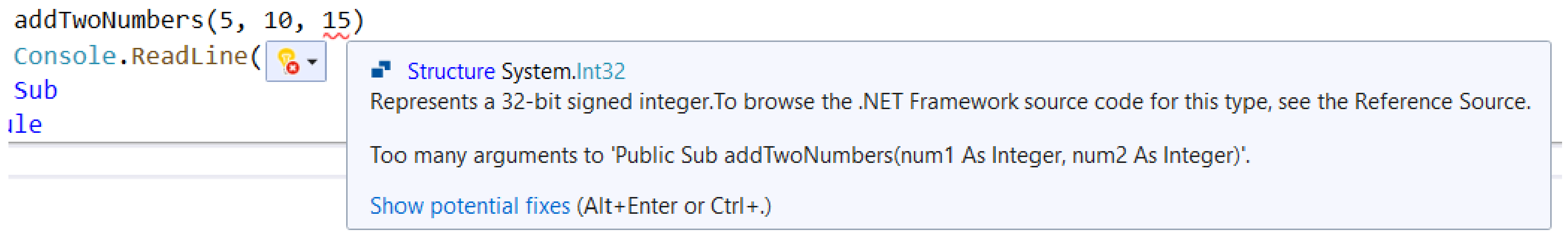
Argument not specified for parameter 'num2' of 'Public Sub addTwoNumbers(num1 As Integer, num2 As Integer)'.

Show potential fixes (Alt+Enter or Ctrl+.)

Too many values passed into a subroutine

```
Module Module1
    Sub addTwoNumbers(num1 As Integer, num2 As Integer)
        Dim total As Integer
        total = num1 + num2
        Console.WriteLine(total)
    End Sub

    Sub Main()
        addTwoNumbers(5, 10, 15)
    End Sub
End Module
```

In Visual Studio `addToNumbers` will be underlined in red and hovering the mouse over it will show the following:



```
addTwoNumbers(5, 10, 15)
Console.ReadLine(
Sub
le
```

Structure System.Int32
Represents a 32-bit signed integer.To browse the .NET Framework source code for this type, see the Reference Source.

Too many arguments to 'Public Sub addTwoNumbers(num1 As Integer, num2 As Integer)'.

Show potential fixes (Alt+Enter or Ctrl+.)

Return values and scope

```
Module Module1
    Function LenOfStr(str As String) As Integer
        Dim length = 0
        For Each character As Char In str
            length = length + 1
        Next
        Return length
    End Function

    Sub Main()
        Dim phrase = "Computer Science"
        Dim phraseLen = LenOfStr(phrase)
        Console.WriteLine(phraseLen)
    End Sub
End Module
```

Notes:

1. Because `string` is a reserved word in VB.NET we have had to use `str` as the name of the parameter
2. Strings in VB.NET are enclosed with double quotes, eg `"Computer Science"`, not single quotes as used in pseudo-code

Summary of the power of subroutines

Example 1

```
Module Module1
    Sub Main()
        Dim height = 3
        Dim width = 3
        For row = 1 To height
            For column = 1 To width
                If (column Mod 3) = 0 Then
                    Console.WriteLine(" O ")
                Else
                    Console.Write(" O |")
                End If
            Next
        Next
    End Sub
End Module
```

Example 2

```
Module Module1
    Sub Grid3x3()
        Dim height = 3
        Dim width = 3
        For row = 1 To height
            For column = 1 To width
                If (column Mod 3) = 0 Then
                    Console.WriteLine(" O ")
                Else
                    Console.Write(" O |")
                End If
            Next
        Next
    End Sub

    Sub Main()
        Grid3x3()
    End Sub
End Module
```

Example 3

```
Module Module1
    Sub DrawGrid(height As Integer, width As Integer)
        For row = 1 To height
            For column = 1 To width
                If (column Mod width) = 0 Then
                    Console.WriteLine(" O ")
                Else
                    Console.Write(" O |")
                End If
            Next
        Next
    End Sub

    Sub Main()
        DrawGrid(3, 3)
        DrawGrid(9, 9)
        DrawGrid(7, 2)
    End Sub

End Module
```

Plan and code - Menu

```
Module Module1
    Sub ShowMenu()
        Console.WriteLine("Choose a menu option")
        Console.WriteLine("Option A - Select 1")
        Console.WriteLine("Option B - Select 2")
        Console.WriteLine("Option C - Select 3")
        Console.WriteLine("Quit     - Select 4")
    End Sub

    Function ValidateNumber(lowest As Integer, highest As Integer)
        Dim num As Integer
        num = Convert.ToInt32(Console.ReadLine())
        While (num < lowest) Or (num > highest)
            Console.WriteLine("Invalid choice try again")
            num = Convert.ToInt32(Console.ReadLine())
        End While
        Return num
    End Function

    Sub ShowChoice(num)
        If num = 1 Then
            Console.WriteLine("You have chosen Option A")
        ElseIf num = 2 Then
            Console.WriteLine("You have chosen Option B")
        ElseIf num = 3 Then
            Console.WriteLine("You have chosen Option C")
        ElseIf num = 4 Then
            Console.WriteLine("You have chosen to Quit")
        End If
    End Sub
    Sub Main()
        Dim choice = 0
        While choice <> 4
            ShowMenu()
            choice = ValidateNumber(1, 4)
            ShowChoice(choice)
        End While
        Console.WriteLine("Thank you for using my program")
    End Sub
End Module
```