Chapter 8: Programming

In this chapter you will learn:

- about the three basic constructs of programming
- about different types of data
- how to input, output and store data in a program
- about the common mathematical components that are used in programs
- how to read and write data into a file from a program
- how to manipulate strings of data
- what is meant by a sub-program
- what is meant by logical and syntax errors in a program
- how SQL can be used to search data in a database
- about the need to create maintainable code
- about the need for good design in a program
- about the purpose of testing and the different types of testing that can be carried out
- about the difference between high-level and low-level language
- what is meant by a translator
- what is meant by an IDE

In this chapter there will be some example programs or sections of programs shown. All the programs have been written in the pseudocode standard provided in the pseudocode guide published by OCR.

What are the three basic programming constructs?

OCR specification reference:

☑ the use of the three basic programming constructs used to control the flow of a program:
- sequence
- selection
- iteration (count and condition-controlled loops)

There are three basic constructs that are the foundations of every program that we create; these are **sequence**, **selection** and **iteration**:

Construct	Description	
Sequence	The order in which the instructions need to occur in an algorithm is called a sequence. If the instructions are not carried out in the correct sequence, a program may not have the desired output. Therefore, we need to carefully consider the order in which instructions should occur when we are designing an algorithm.	
Selection	When writing a program, it may be necessary to create a way for the program to follow different paths. This is often dependent on a question and a **condition**. The method for creating different paths in a program is called selection. The path the program takes will depend on the answer to the question and the condition.	**Condition** – a feature in an algorithm that can be met or not met, causing a different action to be taken
Iteration	Sometimes when we write a program we will need to repeat instructions. The repetition of instructions is called iteration. We can put the instructions we would like to repeat inside a **loop**. We can then set the loop to repeat a set number of times (a counting loop) or we can set the loop to repeat until a condition is met, or stops being met (a condition loop).	**Loop** – a sequence of instructions that is continually repeated a set number of times, or until a condition is met

The sequence of instructions in a program is represented by the order in which the instructions are written. We also need to represent selection and iteration in a program.

Sequence – the order in which a set of instructions is carried out in an algorithm

Selection

Selection is used to change the path or flow of data in a program. There are two types of selection that we need to know; these are:

Selection – a way of creating multiple paths and decisions in an algorithm

- if-then-else statements
- switch-case statements

An if-then-else statement allows us to check whether a condition is true or false and carry out a different instruction depending on whether it is or not. They basically follow the logic that if x is true then do y, else do z.

For example:

```
if name == "me" then
      print ("Hello me!")
else
      print ("Who are you?")
endif
```

This program will check if the data stored in the **variable** 'name' is equal to 'me'. If that is true it will output 'Hello me!'; if it is false it will output 'Who are you?'. We then end the statement we are making. This allows the program to recognise that we have finished our selection and to move on to the next instruction in the sequence.

Variable – a named storage location that contains a value that can change during the execution of a program

If we have more than one condition that we want to check, we can use the command **'elseif'** in the statement. For example:

```
if name == "me" then
      print ("Hello me!")
elseif name == "you" then
      print ("Hello you!")
else
      print ("Who are you?")
endif
```

Another way to do the above is using switch-case statements, especially useful when we have a number of different options that we want to create a different action for. For example:

```
MeatCount = 0
FishCount = 0
VegetarianCount = 0
meal = input("What mean would you like to eat?")
switch meal:
      case "Meat":
            MeatCount = MeatCount + 1
      case "Fish":
            FishCount = FishCount + 1
      case "Vegetarian":
            VegetarianCount = VegetarianCount + 1
      default:
            print ("We do not recognise your selection")
      endswitch
```

This program is designed to give a user three options for their meal choice: a meat, a fish and a vegetarian option. Whichever option they choose, it will increment a counter for that meal by one to register their choice. It also contains a default option. It will choose to follow this option if it cannot find an input that matches one of the other case options.

Iteration

Iteration is used to repeat instructions in a program. There are two types of iteration that we need to know; these are:

Iteration – a way of repeating a set of instructions in an algorithm

- for-to-next counting loops
- while-endwhile condition loops

A for-to-next loop allows us to repeat an instruction or a sequence of instructions a set number of times. For example:

```
for n = 0 to 5
      print ("Hello world!")
next n
```

This program will output the text 'Hello world!' six times.

Discussion point: Why, when we have set the number of loops to stop when *n* reaches 5, does it output the text six times?

We should choose a counting loop when we want to repeat the instructions a set number of times. If we do not know how many times we want to repeat a set of instructions, chances are it will be until a condition is true or false.

A while-endwhile loop allows us to repeat an instruction or a sequence of instructions until a condition is true or false. This type of loop is controlled by a condition at the start of the loop. For example:

```
answer = false
while answer == false
      print ("Computer Science is the best subject")
      answer = input("True or false")
endwhile
```

This program will keep outputting the statement 'Computer Science is the best subject' until the user inputs that this statement is true.

What are data types and why do we use them?

OCR specification reference:

☑ the use of data types:

- integer
- character and string
- Boolean
- real
- casting

When we create programs we need to set different data types for the data that is input, processed and output by our program. It is important that we set the right data type for the right data. This enables us to get the correct input or output for our user. We need to be aware of the following data types:

Data type	Description	Example data
Integer	These are whole numbers only	0, 1, 2, 3
Real	These are numbers that can have a decimal part as well	0.1, 1.2, 3.4
Boolean	This has two values only, true and false	True/False, 1/0, Y/N
Character	This is a single letter, number or symbol	A, B, C
String	This is used for text, and can include any character	Computer Science is fun!

There may be a point in a program where we need to manually change the data type of an item of data that we have stored. This is called casting. If you have ever used Python to program, you will know that we need to change any data that we store to the string data type if we want to output that data.

How do we input, output and store data in a computer program?

OCR specification reference:

- ☑ the use of variables, constants, operators, inputs, outputs and assignments
- ☑ the use of arrays (or equivalent) when solving problems, including both one- and two-dimensional arrays

When we create a program, we will need to create a way for our user to input data into a program. We will also need to create a way to store the input from our user, and a way to output any data we require to our user.

To store the data that is input into our program we will need to create a data store. We can do this by using a **variable**, a **constant** or an **array**.

Constant – a named storage location that contains a value that cannot be changed during the execution of a program

Array – a set of data that is stored together, that is of the same data type

Assignment – this is setting or resetting a value to a data location, such as a variable

Variable

A variable is a data store that is given a name. It stores data that can be changed throughout the execution of a program. The name of the variable links it to the memory location where the data is stored. The way we store data in a variable is by **assigning** it. The assignment symbol equals (=) is used to assign data to a variable.

For example:

```
name = "me"
```

The data stored in the variable. The quotation marks give it a string data type.

The name given to the variable

We can reassign the data that is stored in our variable at any point that is required in our program.

For example:

```
name = "Bob"
name = "Jane"
```

In this instance **name** will store 'Bob' until it is reassigned to store 'Jane'.

We can give any name that we like to our data store, but we should choose the name carefully. The name must be meaningful and relate to the data that is stored. We should also avoid putting spaces in our data store names as this can sometimes cause problems in our program.

Local variable – a variable that is declared within a function or procedure and can therefore only be used within that function or procedure

Global variable – a variable that is visible throughout the whole of the program

Procedure – a named section of instructions that perform a specific task

A variable can also be set to a **local variable** or a **global variable**. A global variable is one that is recognised throughout the whole of our program. If we use the variable name, it will recognise it anywhere in the program and use the data that is stored in it at any point in our program. When we use sub-procedures (see later in the chapter) we use local variables.

Any variables that are assigned in the sub-procedure are not recognised by the program outside of the sub-procedure. This means that they are local variables.

Constant

A constant is a data store that is also given a name. It stores data that cannot be changed throughout the execution of the program. If we have a value that we need to use repeatedly in a program, we can either:

1. Type that value into the code lines where it is used (some people call this *hardcoding*)
2. Use a constant

Hardcoding values is considered bad programming practice, as it makes the code less maintainable. If we need to change the value, we need to find and change all instances of it. If a value is typed in 50 times, across 5,000 lines of code – it can be difficult and time-consuming to identify all of the places it has been used. If any of the values are missed, this has the potential to cause errors in the program.

Therefore, best practice is to use a constant to store the value. We assign the value we want to use throughout the program to the constant. This means that:

- Code is more maintainable. It is easy to update the value stored in the constant – usually located at the start of the program.
- If we change the value stored in the constant, it will automatically update all the places that it is used in the program.

A constant is declared by using a keyword, for instance CONST. We store data in a constant in the same way that we store it in a variable, by assigning it to a constant name. For example:

```
CONST pi = 3.1415
```

Array

An array is a data store that is designed to store a set of data that is all of the same data type. For example, a list of the names of the students in your class could be stored in an array. Each item of data in an array is called an element; each element can be accessed by referring to its position in the array. For example:

```
ClassList = ["Bob", "Jane", "Bill", "Annie", "Ali"]
```

If we want to reference an element of data in the list, we refer to the data location of the element. One thing to note is that many programming languages label the first data location in an array as location 0. This means that the data stored in element 3 of our class list array would be Annie. When we create an array to store data we declare a name for the array and how big we want the array to be. For example:

```
array ClassList[5]
```

Then to write each element of data we want to store into the array we write this as:

```
ClassList[0] = "Bob"
ClassList[1] = "Jane"
ClassList[2] = "Bill"
ClassList[3] = "Annie"
ClassList[4] = "Ali"
```

This writes the data we require to each element in the array.

This type of array is a called a one-dimensional array. We also need to know about a second type of array called a two-dimensional array. If we think of a one-dimensional array like a list of data that is stored, we can think of a two-dimensional array as a table of data that is stored. The reference to elements in this type of array has two parts. The first is like the column reference in a table and the second is like the row reference in the table. For example:

```
array TestScores [3,5]
```


This two-dimensional array will store three test scores for each of the five students in the class. For example:

Test Score	Student Number				
	0	1	2	3	4
0	30	35	32	25	22
1	29	21	29	25	28
2	27	23	32	26	19

We can find out the test score stored for a student for a particular test by referring to the student number and the test number. For example:

```
TestScores [3,1]    This would return the test score 25
TestScores [0,2]    This would return the test score 27
```

We write data to a two-dimensional array in the same way we write it to a one-dimensional array, by referencing the element. For example:

```
TestScores [0,0] = 30    This writes the test score 30 to student 0 for test number 0
TestScores [3,2] = 26    This writes the test score 26 to student 3 for test number 2
```

We can also change the data stored in a particular element of an array by simply reassigning it. If student 1's test for test 1 is remarked and a new test score needs to be entered, we can reassign the new test score to the correct location in the array. For example:

```
TestScores [1,1] = 25    This would overwrite the previous test score stored (21)
```

We can also declare an array in which we do not want to set an amount of elements to be stored. This may be because we do not know how many elements we need to store at that point. This is called a **dynamic array**. In a **static array** we declare the array and say how many elements we want to include. For example:

Dynamic array – an array that has does not have a fixed length applied to it when it is declared

Static array – an array that has a fixed length applied to it when it is declared

```
array ClassList [5]
```

But with a dynamic array we simply leave the number of elements blank. For example:

```
array ClassList [ ]
```

In a dynamic array, we cannot add data to it in the same way that we can a static. We could not add the names into the class list by stating the position of the element like we did with the static array. For example:

```
ClassList [0] = "Bob"
ClassList [1] = "Jane"
ClassList [2] = "Bill"
ClassList [3] = "Annie"
ClassList [4] = "Ali"
```

In a dynamic array we have to add in each name by appending it to the array. Each name we add will be appended to the end of the array. We use the .append command to do this. For example:

```
array ClassList [ ]
ClassList.append("Bob")
ClassList.append("Jane")
ClassList.append("Bill")
ClassList.append("Annie")
ClassList.append("Ali")
```

Once we have added the names into the list, we can then refer to their element in the same way that we can with a static array.

Inputs and outputs

When we want to allow data to be input into our program, we need to create an input for our user. For example:

```
name = input("Please enter the name of the student")
```

This will allow us to create a variable called 'name'. It will then allow us to store whatever data the user inputs. It also provides the user with a prompt of 'Please enter the name of the student'.

When we want to output data to our user, we need to create an output for our data. For example:

```
print ("Hello user! Welcome to my program")
```

This would output the message inside the quotation marks to the user. We can also output the contents that are in a data store. For example:

```
name = "Jane"
print (name)
```

This would output the name 'Jane' to the user as the data 'Jane' is stored in the variable 'name'.

What are the common mathematical operators that are used in programming?

OCR specification reference:

- ☑ the common arithmetic operators
- ☑ applying computing-related mathematics:
  - addition
  - subtraction
  - division
  - multiplication
  - exponentiation
  - MOD
  - DIV

There are many arithmetic operators that we need to understand in order to effectively use them in programs:

Operator	Description	Example
+	Addition	1 + 2 = 3
-	Subtraction	2 – 1 = 1
/	Division	6 / 3 = 2
*	Multiplication	6 * 3 = 18
Exponential (^)	When one number increases exponentially (the number of times) to another	2^3 = 8 (2 * 2 * 2 = 8)
Modulus (MOD)	The remainder that is left over when a number is divided by another	10 MOD 3 = 1
Quotient (DIV)	The number of times a number can be divided by another	10 DIV 3 = 3
==	Is equal to – this can have a true or false outcome	1 == 1 would be True
<> or !=	Is not equal to – this can have a true or false outcome	1 != 1 would be False
<	Is less than – this can have a true or false outcome	4 < 5 would be True
>	Is greater than – this can have a true or false outcome	4 > 5 would be False
<=	Is less than or equal to – this can have a true or false outcome	4 <= 4 would be True
>=	Is greater than or equal to – this can have a true or false outcome	4 >= 5 would be False

We can use these arithmetic operators on integers and real numbers in our program. The rules of programming follow the same rules as maths in terms of the order in which operators are carried out. They follow the rules of BODMAS.

How do we read data to and write data from a file?

OCR specification reference:

☑ the use of basic file-handling operations:

- open
- read
- write
- close

File handling can be one of the trickier aspects of programming. It is when we take data that we have stored in a variable or an array and store it in an external file. The most common types of external file that are used .txt and .csv files.

There are four main commands we use when handling files:

- Open
- Close
- Read
- Write

Putting data into a file is called writing data to a file. Using data from a file in a program is called reading data from a file. In order to access an external file, we need to open it first. When we have finished reading data to or writing data from an external file, we need to close it afterwards.

These are the steps we should follow when reading data to and writing data from a file:

- We firstly need to open the file.
- Once opened, the program will start reading the file or writing to the file from the beginning.
- When reading or writing is complete, we must close the file. If we do not close the file it can cause us problems when we try to edit it next time.

To read from a file, we use an open command and follow this with a command to read from the file:

```
MyFile = openRead ("test.txt")
```

This will allow us to open a file called 'test.txt' and allow us to read data from the file.

If we then want to read the first line of data from the 'test.txt' file into our program, we need to use the following command:

```
name = MyFile.readline()
```

To write to a file, we use an open command and follow this with a command to write to the file:

```
MyFile = openWrite ("test.txt")
```

This will allow us to open a file called 'text.txt' and allow us to write data to the file.

If we want to write data to our file 'test.txt', we need to use the following command:

```
MyFile = writeLine ("Bob")
```

This will allow us to write the data 'Bob' to the file 'test.txt'.

When we have finished editing our file we need to close it using the command:

```
MyFile.close()
```

If we want to create a small program that will write the name 'Bob' to the file 'test.txt' we would use the following:

```
MyFile = openWrite ("test.txt")
MyFile = writeLine ("Bob")
MyFile.close()
```


We may want to read all the lines of data from a file into an array. We may not know how many lines of data are stored in the file and we want the program to keep reading the data in till it gets to the end of the data. We need to use another file-handling command to do this:

```
MyFile.endOfFile()
```

If we wanted to create a program that read all the data from the file 'test.txt' into an array called 'names', we would use:

```
array names[]
MyFile = openRead("test.txt")
position = 0
while NOT MyFile.endOfFile()
      names[position] = MyFile.readline()
      position = position + 1
endwhile
MyFile.close()
print (names)
```

This program does the following:

- We start the program by declaring the array 'names'. We don't declare the size of the array as we don't know how many names are stored in the file.
- The program then opens the file 'test.txt' where the data is stored. It opens it in read mode as we want to read data into the program.
- We then use a while loop that will keep carrying out the instructions inside the loop till it reaches the end of the file.
- The instructions it will carry out cause it to read the line of data from the file into the array names.
- Each time data is written to an element in the array, the position variable keeps track of which element it is reading data to and increments it by one each time. This will stop the same element being overwritten each time.
- Once the end of the file is reached, the program will stop looping.
- We then close the file.
- The program will then output the contents that have been read into the array 'names'.

What is string manipulation?

OCR specification reference:
☑ the use of basic string manipulation

When we looked at the different data types, we described the string data type as a text-based one. It can contain any type of character, including letters, numbers and symbols. When we write a string in programs it is normally written inside single or double quotation marks. We have been using double quotation marks.

When we are writing programs we can join together strings of data to form a new string of data. This is called **concatenation**. We concatenate strings of data together by using the + symbol. For example:

Concatenation – the joining together of characters in a string

```
greeting = "Hello"
name = "Bob"
question = "how are you? "
message = greeting + name + question
print (message)
```


This program stores three strings of data, 'greeting', 'name' and 'question'. It then concatenates these three strings together and stores this in the variable 'message'. It then outputs the data that is stored in 'message'. At the moment there is a problem with the output; it would currently output the following:

HelloBobhow are you?

This is because we haven't included a way to create spaces in between the strings of data that we have concatenated together. To create spaces we need to add the following:

```
message = greeting + " "  + name + " " + question
```

The inclusion of a space in between quotation marks allows us to add spaces in between strings of data. We need to remember to use the + sign to add together each part that we want to concatenate, this includes for the space (" ") as well.

There are a number of other actions we can perform on strings of data that we have stored. We will use the variable **name = "Bob"** in our example. We are able to use the following commands:

Command	Description	Example	Output
.length	This gives the number of characters that are in the string. This will include any spaces.	name.length	3
.upper	This changes all the characters in the string to upper case.	name.upper	BOB
.lower	This changes all the characters in the string to lower case.	name.lower	bob
[n]	This selects a particular character in the string.	name[2]	b
.subString(x, y)	This selects a particular part of a string. It starts at the place in the string set by x and selects the next number of characters set by y.	name.subString(1,2)	ob
str()	This converts any data into the data type of a string.	str(name)	-

We can make use of string manipulation in many ways. Have you ever noticed that shops always have codes for all of the products they stock? They can use string manipulation to create these codes. For example:

```
colour = input("What colour is the product? ")
size = input("What size is the product? ")
type = input("What type of product is it? ")
productCode = colour.subString(0,2) + type.subString(0,2) + size
print (productCode)
```

This program creates a product code for each product by taking the first two characters of the colour of the product, joins this to the first two characters of the type of the product and then joins the size. If we input the colour Black, the type Trousers and the size 10, we would get the product code BlTr10.

What are sub-programs and when would we use them?

OCR specification reference:

☑ how to use sub-programs (functions and procedures) to produce structured code

When we create programs, we may find that we have sequences of instructions that we need to use on regular occasions in our programs. We don't use them repeatedly in a loop, but we do use the same instructions at various points in our program. If we have sets of instructions that we use in this way, we can create a sub-program to store them. We then just need to refer to the name we give to the sub-program, when we want to use the particular set of instructions.

Using sub-procedures in our program can make our code neater and more efficient. They increase the reliability of our code and can reduce the amount of code that we need to write.

When we create sub-procedures, we may need to pass data into our program. By this we mean use a value or data in the sub-procedure we are creating. To do this we use parameters and arguments. A parameter is a type of variable that is used to store the value. As with a normal type of variable, a parameter is given a name. It can also be given a default value. An argument is the actual value that we want to use and pass into our sub-procedure.

Function – a named section of instructions that perform a specific task and return a value from this

There are two types of sub-procedure that we can use: these are procedures and **functions**. We can show this by using a sub-procedure called 'greeting'.

```
greeting ()        This is the sub-procedure without a parameter or argument
greeting (name)    This is the sub-procedure with a parameter called name
greeting ("Bob")   This is the sub-procedure with an argument of 'Bob'
```

Procedures

A procedure is a set of instructions that we can store and then use at any point we want to in our program. We need to create our procedure, and then we can use it any time we like by referring to the name we give it. We refer to this as 'calling' it. For example:

```
procedure greeting ()
      print ("Hello!")
      print ("Computer Science is so much fun!")
endprocedure
```

We can then call our procedure. For example:

```
greeting ()
```

Every time we call the procedure it will output:

Hello!
Computer Science is so much fun!

This is using the procedure without a parameter. We can add a parameter to the procedure to make our message more personal. For example:

```
procedure greeting (name)
      print ("Hello " + name + "!")
      print ("Computer Science is so much fun!")
endprocedure
```

The parameter is given an argument of 'Bob'. When we call the procedure it will output:

Hello Bob!
Computer Science is so much fun!

Functions

A function is similar to a procedure in that it stores a set of instructions to be carried out. It is different in that it returns a value when it is called. This means that it outputs a value from the instructions that it carries out. For example:

```
function avergeTestScore (t1, t2, t3)
      total = t1 + t2 + t3
      return total/3
endfunction

average = averageTestScore(20, 12, 25)
print (average)
```

This function has three parameters that are passed into the function. The output would be 19. The value returned by a function must be stored in a variable or the value will be lost.

What are syntax and logical errors?

OCR specification reference:
☑ how to identify syntax and logic errors

When we create computer programs we may encounter a number of issues. Two of these issues could be **syntax errors** and **logic errors**.

Syntax error – an error in the language written in a program

Logical error – an error in a program that causes it to produce unexpected results

These errors are normally highlighted when we try to run our program. Our programming environment will often instruct us that we have an error in our program. We will need to work out whether the error is a syntax one or a logical one.

A syntax error is when the **translator** does not understand the text that we have typed. The most common type of syntax error is a variable or a command that is spelt incorrectly.

Translator – software that translates a program written in a programming language into machine code that can be understood by a computer

A logical error is when our program will run but we get an unexpected result. This means we have made an error in the logic of our program at some point that is not giving us the output that we expected.

An example of a syntax error could be:

```
name = "Bob"
print ("Hello " + naem)
```

This would cause the translator to advise us that we do not have a variable declared called 'naem'.

An example of a logic error could be:

```
while n < 5
      n = 0
      print (n)
      n = n + 1
next n
```

This would cause a logical error as it would create what is called an infinite loop. The variable n will never reach 5 because each time the loop is run it is reset to 0. The declaration n = 0 should occur at the start of the program, outside the loop.

What is a database?

OCR specification reference:
☑ the use of records to store data

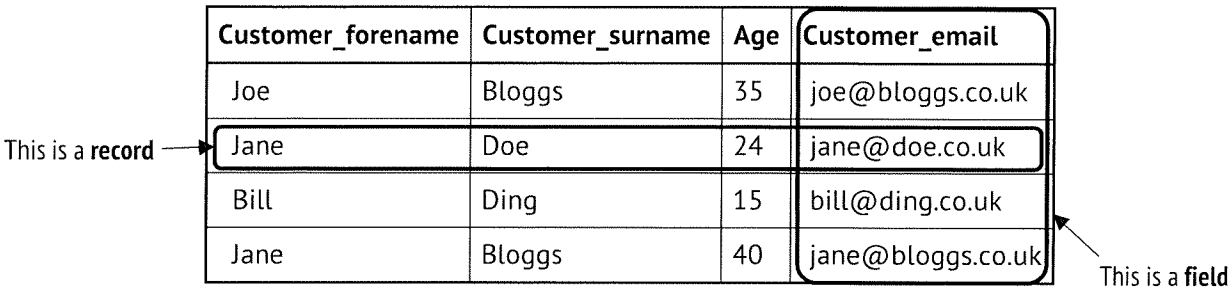
One of the common ways we often store large amounts of data is in a **database**. database is separated into tables, records and fields.

Database – a structured way of storing data

A table will store a collection of records and fields that are all related. An organisation may have a table in a database that is dedicated to storing their customers' personal details.

A field in a database stores a particular item of data. An example of a field in a table that stores customer personal details would be 'customer forename'.

A record in a database stores the data from a single collection of fields in a table. An example of a record in a table that stores customer personal details would be all the data for a single customer.



Customer_forename	Customer_surname	Age	Customer_email
Joe	Bloggs	35	joe@bloggs.co.uk
Jane	Doe	24	jane@doe.co.uk
Bill	Ding	15	bill@ding.co.uk
Jane	Bloggs	40	jane@bloggs.co.uk

The whole collection of data is a **table**.

We mostly store data in databases so that it is stored in a structured format for future use. One reason we need the data in a structured format is so that we can search through it effectively.

How can we use SQL commands to search for data in a database?

OCR specification reference:
☑ the use of SQL to search for data

Structured query language (SQL) is a language that is used for creating searches for a database. SQL allows users to state what field and table we want to search and what data we want to extract. There are a number of SQL commands that we need to be able to use; these are:

SELECT	This states what data we want to look at. It is often a field name.
FROM	This states where the data is stored. It is often a table name.
WHERE	This is used to describe the data we want to extract. It often uses a condition.
LIKE	This is also used to describe the data we want to extract.
AND	This is a Boolean operator that will join together search criteria.
OR	This is a Boolean operator that will search for either one criterion or another.

We combine these commands to create searches for a database. For example:

```
SELECT Customer_forename FROM Customers WHERE
Customer_surname = "Bloggs"
```

This search would return two records: *Jane* and *Joe*.

We can add to our search:

```
SELECT Customer_forename, Customer_surname FROM Customers WHERE
Customer_surname = "Bloggs"
```

This search would still return two records, but with additional data: *Jane Bloggs* and *Joe Bloggs*.

We can make our search more specific:

```
SELECT Customer_forename, Customer_surname FROM Customers WHERE
Customer_surname = "Bloggs" AND Age >35
```

This search would just return one record: Jane Bloggs

We can open up our search criteria a little:

```
SELECT Customer_forename, Customer_surname FROM Customers WHERE
Customer_forename LIKE "J%"
```


This search will return three records: Joe Bloggs, Jane Doe and Jane Bloggs. The LIKE commands allows us to search for partial data. The criteria J% means that the text needs to start with 'J' but what comes after that is not important.

If we want to display all the records in a table we use an asterisk (*):

```
SELECT * from Customers
```

How do we create maintainable code and why is it important to do this?

OCR specification reference:

☑ maintainability:
- comments
- indentation

It is very important, when we create code, to make sure that our code is maintainable. This means that when we look at the code again after a period of time, or someone else wants to use the code at a later point in time, it is clear to follow the code and understand how it has been structured.

There are two things that we can do to make sure our code is maintainable and easy to understand and use in the future: these are adding comments to our code and indenting our code.

When we add comments to our code, we describe what the different parts of our code are doing. We will add comments to describe how a certain section, such as an iteration, selection or sub-procedure is working.

We add comments in our pseudocode by using two forward slashes (//) followed by our description.
For example:

```
name = "Bob"        // This is a comment. This is a variable that stores a name.
```

By adding comments to our code, this means that when we look at it again in the future, or someone else wants to use our code, we can read the comments and follow the structure of how the program has been created.

We can also use indentation to make our code more readable in the future. Some programming languages insist on us indenting sections to show what instruction should occur, for example inside a loop or selection; others do not insist on this but we should still do it. By indenting our code, it makes it easier for us to see what instructions are meant to be executed inside the loop or as a result of a condition in a selection.

How can we design an effective program?

OCR specification reference:

☑ defensive design considerations:
- input sanitisation/validation
- planning for contingencies
- anticipating misuse
- authentication

When we design our programs we should think about making them as user-friendly and as secure as possible. We want our program to be unbreakable and for no errors to occur when it is used. In order to achieve this, we need to try to anticipate what errors our user might make. Thinking about designing our program in this way is called defensive design.

One of the main areas in which a user may make a mistake is in the data that they are required to input into a program. They may enter the wrong data accidentally, by mistyping, or they may misunderstand the data they are required to enter, and enter the wrong type of data. To stop errors from occurring in this way we can do

three things: we can create clear and instructive prompts for our user interface that clearly tell our user what data we want them to input, we can validate the inputs they give, and we can also remove any unwanted data from the input, sanitising it. For example, an input might not allow numbers in it, so you could write a function that sanitises the data by removing all digits from the input.

When we use **input sanitisation** on a user's input, we remove any data that we do not want, or could be potentially harmful. This is often done by anticipating what kind of incorrect data a user may input, storing it in a list, and asking the program to check the list to remove any of the data it finds that matches the list.

Input sanitation – technique of modifying input to ensure it is valid

Validation – techniques to check if the input meets a set of criteria

When we use **validation** on our inputs, we create rules that the input must follow to be acceptable. There are some common validation rules that we may choose to use:

Validation rule	Description
Length check	Checks that the data entered contains a set number of characters
Range check	Checks that the data entered is within a certain number range
Type check	Checks that the data entered is a certain data type
Format check	Checks that the data entered has a particular format, e.g. has an @ symbol

By limiting or checking the data our user enters, this can make the data more reliable and stop errors occurring in our program.

In order to increase the security of our program we can create a method of authentication for a user. This way we can make sure that only the users we choose have access to using the program.

Authentication is a way of confirming the identity of the user, and a common way of doing this is asking them to enter a username and a password. When a user wants to access the program they will be asked to input a username and password. This data will need to match data that has previously been set in the program.

Authentication – techniques used to confirm a users' identify

What is the purpose of testing?

OCR specification reference:

- ☑ the purpose of testing
- ☑ types of testing:
  - iterative
  - final/terminal
- ☑ selecting and using suitable test data

During the creation of our program, and also when we have finished, we should be carrying out regular testing. When we test our program we are checking that it does not contain any errors and that it works in the way that we want it to. The more testing we can carry out on our program, the less likely it is to break in the future due to a user encountering an error that had not previously been discovered.

We should be testing our program while we are building it as well as when we have finished building it. When we are testing our program during the process of building it, this is called systematic testing. This is a form of iterative testing, which is when we test our program and go back and improve it as a result of the tests that have been carried out. If we systematically test our program while we build it, we will find many of the errors as we go along. This will mean we have fewer errors to find and correct at the end.

We will also carry out testing on our program when we have finished building it. This type of testing is called final or terminal testing. This type of testing is often aimed more at testing the functionality of the system to find out whether it meets the requirements of the user that were laid out when a design was developed for the program.

When we carry out anything we must make sure testing is destructive. We must try to test with as much test data as we can to try to break our program. This will mean not just entering the data that an input should accept, but testing that it does not accept any other kind of data that we do not want it to accept.

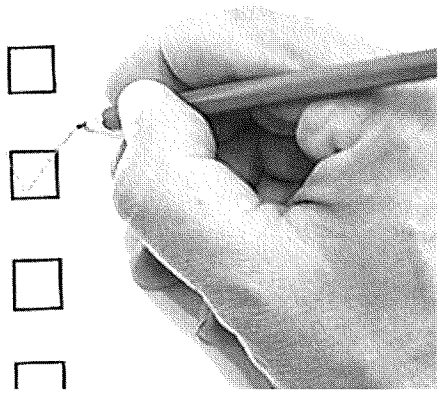
For example, let's consider the following input:

```
age = int(input("Please enter your age"))
```

We have created an input that is limited to accepting only integer data. We will want to test whether the input accepts data that is of the data type integer. But we will also need to check that the input rejects real data types or any string data types. Therefore, we may test it with the following data:

Test number	Test description	Test data	Test type	Expected outcome
1	Testing the input for the age variable accepts an integer data type	16	Normal	Data will be accepted
2	Testing the input for the age variable rejects a number that is real	10.2	Erroneous	Data will be rejected
3	Testing the input for the age variable rejects a character that is a letter	A	Erroneous	Data will be rejected
4	Testing the input for the age variable rejects a string of characters	hello	Erroneous	Data will be rejected
5	Testing the input for the age variable rejects a character that is a symbol	!	Erroneous	Data will be rejected

This selection of data thoroughly tests our input, making sure that it should only accept data that is an integer. We have recorded our testing in a testing table. You will need to produce testing tables like this for the testing that you carry out in your coursework. You will need to make sure that you thoroughly test your program with data that should be accepted and also rejected by your program. You will also need to test the logic of your program and test to see whether the design requirements have been met.



Testing needs to be destructive to make sure that programs do not break in the future

We have used two main types of test data in our testing table: normal and erroneous test data. There is a third type of test data called extreme (or boundary) test data.

- Normal test data refers to data that the program should process normally, data that it should accept.
- Erroneous data refers to data that the program should not accept, data that it should reject.
- Extreme data refers to data that will fall on the edge, or boundary, of any ranges or limits that have been set.

Discussion point: What would be the boundary data for the example given in the table above?

What is the difference between high-level and low-level language?

OCR specification reference:

☑ characteristics and purpose of different levels of programming language, including low-level languages

A computer programming language can be either what is described as a **high level-language** or a **low-level language.**

High-level programming languages are the languages that most people use to program. They use language statements and commands that are close to what we use on a daily basis as humans. They are much easier for users to read and write programs in. Examples of high-level languages would Python, Java, JavaScript and Visual Basic.

High-level language – a computer language, in which programs are written, that is closer to human language

Low-level language – a computer language, in which programs are written, that is closer to what a computer understands

Low-level programming languages are much closer to the language that a computer understands. Examples of low-level languages would be assembly code and machine code. Low-level language often will often be written using mnemonics as instructions. It requires more specialist knowledge to write programs in low-level languages.

In a low-level language each line of code will perform only one task. In a high-level language one line of code can perform many tasks. Low-level language is still used to program certain software, such as drivers for hardware.

What is a translator?

OCR specification reference:

☑ the purpose of translators

☑ the characteristics of an assembler, a compiler and an interpreter

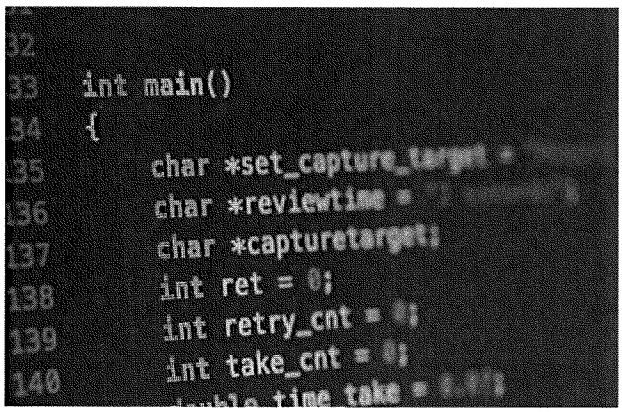
Computer will only understand instructions and data that are in machine-code form. So any high-level language or low-level assembly language will need to be translated into machine code for a computer to be able to process it and carry out the instructions. There are three types of translator that can be used: a **compiler**, an **interpreter** and an **assembler**.

Compiler – software that translates a high-level language into machine code

Interpreter – software that translates a high-level language by analysing and executing it line by line

Assembler – software that translates assembly code into machine code

A compiler takes code written in a high-level language and translates it into machine code. The code produced when we write a program in a high-level language is called source code. The source code needs to be translated into machine code for the computer to understand it. Running the source code through a compiler is called compiling the code.



High-level code is easily read by humans, unlike machine code, which is a series of 1s and 0s.

A compiler translates the whole of the source code in one go. If there are any errors in the code, it will not compile and it will therefore not run. The compiling process will stop and these errors will need to be removed before the program can be compiled. The machine code from the compiler is output as an executable file (.exe when using the Windows operating system). This file contains the machine code needed for the computer to process the instructions. The file can then be stored for future use whenever the program is needed.

A compiler will have a feature called error handling. The compiler will run through the program and produce a report of all the errors it detects. We can then use this report to locate and fix the errors in our source code.

An interpreter can also take source code written in a high-level language and translate it into machine code. Unlike a compiler, an interpreter translates the source code into machine code one line at a time. The programming language Python, for example, is interpreted, unlike the programming language C which is compiled.

If there are any errors in the program, they will be detected when that line of source code is reached. We can then fix the error and run the source code through the interpreter again.

As an interpreter translates each line of code in turn, it takes up less memory than a compiler that needs to generate an executable file to be stored.

An interpreter may seem like it would be quicker when translating code. However, as an interpreter needs to translate each line of code to run the program each time, they are generally slower than compilers. A compiler will take more time to initially run the program as it takes time to compile the code, but will then run it much more quickly once compiled.

An assembler is a computer program that will take the mnemonics used in assembly language and convert them into machine code so they can be processed by the computer.

What is an IDE?

OCR specification reference:

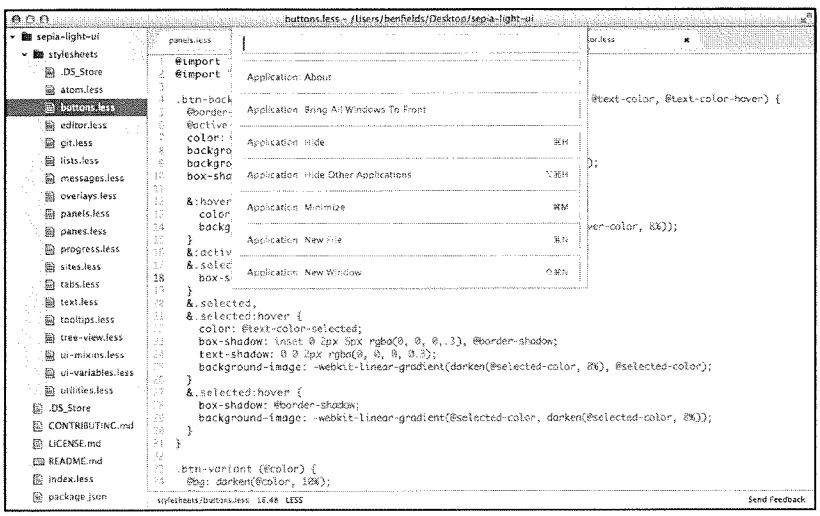
☑ common tools and facilities available in an integrated development environment (IDE):

- editors
- error diagnostics
- run-time environment
- translators

When we write our wonderful programs we normally use an integrated development environment (IDE) to do this. This type of programming environment has additional features that help and guide us in our program writing.

They have a code editor that allows us to write and edit the code for our program. Many code editors will have helpful aids such as automatic line numbers for the lines of code that we write. They can also often have features such as colour coding for the different commands that we use when writing our program. This can help us understand whether the IDE recognises the command we are trying to use. More advanced IDEs may also be able to autocorrect any commands that we mistype, or auto-indent any iteration or selection that we use.

An IDE will have a window where we can see the output of the code when we run the code. This is called the run-time environment. Having a window that allows us to immediately see this allows us to create and edit our code much more quickly.



Atom is an IDE with syntax colouring and other features to help with development.

An IDE will have a method of error diagnostics. This will help us identify where we may have made an error in our code so that we can look to fix it. This will either be an error report if it is a compiler or an error message if it is a translator.

Each IDE will have a built-in complier or interpreter that will allow the source code that we write in our high-level language to be translated into the machine code needed to run the program on the computer.

Chapter Summary

- There are three main programming constructs: sequence, selection and iteration.
- When we write a program we need to apply data types to the data that it handles. There are five main data types: integer, real, Boolean, character and string.
- We store data in various ways in a program, including variables, constants and arrays.
- Arrays can be static or dynamic, and they can also be one-dimensional or two-dimensional.
- We may need to store data externally to a program; we do this using an external file. We need to be able to open and close the file to use it, and to also read and write data to and from the file.
- We may need to manipulate the strings of data that we store in a program, extracting sections from them and joining strings together.
- We can store sections of instructions that we may want to use in a program in a sub-program. There are two main types of sub-program: a function and a procedure.
- There are two main types of error that can occur when writing programs: syntax errors and logical errors.
- We can store data in a database, and we can use SQL to create queries that we can use to quickly and easily search through the data in a database.
- We need to ensure that when we write our code, we make it maintainable. We can do this in a variety of ways, including adding comments to our code and making sure that we correctly indent our code where necessary.
- We need to make sure that we thoroughly test the programs that we create, during the creation of the program with iterative testing, and at the end of the program with terminal testing.
- We can write programs in a high-level language or low-level language. We mainly choose high-level languages as they are closer to human language. Low-level languages are closer to what a computer understands.
- We need a translator to translate our programming language into machine code for a computer. There are three main types of translator: a compiler, an interpreter and an assembler.
- We often use an IDE to write our computer programs. IDEs have a number of features that we can make use of; these include editors, error diagnostics and translators.

Practice Questions

1. Describe what is meant by the term 'selection'. [2]

2. Explain the difference between a high-level language and a low-level language. [2]

3. Explain two ways in which a programmer can make sure that a program they create is effective. [4]

4. A hospital wants a program that records and monitors the temperature of each baby that is currently in its maternity unit. It wants to store the temperature of each baby; it will have 15 babies in the unit at a time.

   If a baby's temperature is below 36.5 degrees, the program should alert a nurse with a suitable message. If a baby's temperature is above 37.5 degrees, the program should alert a nurse with a suitable message.

   Write an algorithm that the hospital could use to store and monitor the temperatures of the babies in the unit. [6]

5. Describe two features of an IDE that help a user to write programs. [4]

6. a) A teacher needs a program that records two test scores for each student in a class of 20. Test 1 has a maximum of 15 marks, test 2 has a maximum of 20 marks. Each mark entered should be validated and any invalid marks must be rejected.

      Write an algorithm that the teacher could use to store the student name and test scores for each student in the class. [7]

   b) The teacher now wants the program to be able to output the students name and their total score of both tests. They also want the program to output the classes average test score for each test.

      Extend the algorithm created in part (a) to include these two new requirements. [7]

