

---

SCHOLAR Study Guide

# Higher Computing Science

## Unit 1: Software design and development

---

### **Authored by:**

Ian King (Kelso High School)

Mark Tennant (Community School of Auchterarder)

Charlie Love (CompEdNet)

Andy McSwan (Knox Academy)

### **Reviewed by:**

Jeremy Scott (George Heriot's School)

### **Previously authored by:**

Jennifer Wilson (Denny High School)

**Heriot-Watt University**

Edinburgh EH14 4AS, United Kingdom.

First published 2018 by Heriot-Watt University.

This edition published in 2018 by Heriot-Watt University SCHOLAR.

Copyright © 2018 SCHOLAR Forum.

Members of the SCHOLAR Forum may reproduce this publication in whole or in part for educational purposes within their establishment providing that no profit accrues at any stage, Any other use of the materials is governed by the general copyright statement that follows.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without written permission from the publisher.

Heriot-Watt University accepts no responsibility or liability whatsoever with regard to the information contained in this study guide.

Distributed by the SCHOLAR Forum.

SCHOLAR Study Guide Higher Computing Science: Unit 1

Higher Computing Science Course Code: C816 76

ISBN 978-1-911057-28-4

Print Production and Fulfilment in UK by Print Trail [www.printtrail.com](http://www.printtrail.com)

## **Acknowledgements**

Thanks are due to the members of Heriot-Watt University's SCHOLAR team who planned and created these materials, and to the many colleagues who reviewed the content.

We would like to acknowledge the assistance of the education authorities, colleges, teachers and students who contributed to the SCHOLAR programme and who evaluated these materials.

Grateful acknowledgement is made for permission to use the following material in the SCHOLAR programme:

The Scottish Qualifications Authority for permission to use Past Papers assessments.

The Scottish Government for financial support.

The content of this Study Guide is aligned to the Scottish Qualifications Authority (SQA) curriculum.

All brand names, product names, logos and related devices are used for identification purposes only and are trademarks, registered trademarks or service marks of their respective holders.



---

# Contents

1	Development methodologies	1
1.1	Revision	3
1.2	An iterative software development process	4
1.3	Agile Development	15
1.4	Learning points	18
1.5	End of topic test	19
2	Analysis	21
2.1	Revision	23
2.2	Inputs, processes, outputs	23
2.3	Purpose, Scope, Boundaries and Functional Requirements	25
2.4	Analysing a program requirement	27
2.5	Learning points	28
2.6	End of topic test	29
3	Design	31
3.1	Revision	33
3.2	Introduction	34
3.3	Structure diagrams	34
3.4	Pseudocode	38
3.5	User interface design	41
3.6	Learning points	43
3.7	End of topic test	44
4	Implementation: Data types and structures	45
4.1	Revision	47
4.2	Data types and pseudocode	48
4.3	Simple data types	49
4.4	Identifying simple data types	50
4.5	Structured data types	51
4.6	Learning points	60
4.7	End of topic test	61
5	Implementation: Algorithm specification	63
5.1	Revision	65
5.2	Standard algorithms	66
5.3	Input validation	67
5.4	Finding the minimum or the maximum value in an array	70
5.5	Counting Occurrences	72
5.6	Linear search	74
5.7	Learning points	78

---

5.8	End of topic test . . . . .	78
6	Implementation: Computational constructs	81
6.1	Revision . . . . .	83
6.2	Introduction . . . . .	84
6.3	Variables and scope . . . . .	84
6.4	Pre-defined Functions . . . . .	85
6.5	Sub-programs . . . . .	89
6.6	User defined functions . . . . .	92
6.7	Parameters . . . . .	93
6.8	Sequential files . . . . .	97
6.9	CSV Files . . . . .	98
6.10	Learning points . . . . .	99
6.11	End of topic test . . . . .	100
7	Testing	103
7.1	Revision . . . . .	105
7.2	Test plans . . . . .	105
7.3	Debugging . . . . .	107
7.4	Debugging tools . . . . .	108
7.5	Learning points . . . . .	111
7.6	End of topic test . . . . .	112
8	Evaluation	115
8.1	Revision . . . . .	117
8.2	Software evaluation . . . . .	117
8.3	Learning points . . . . .	121
8.4	End of topic test . . . . .	122
9	End of unit 1 test	123
	Glossary	130
	Answers to questions and activities	132

---

# Topic 1

## Development methodologies

---

### Contents

1.1	Revision . . . . .	3
1.2	An iterative software development process . . . . .	4
1.2.1	Analysis . . . . .	5
1.2.2	Design . . . . .	6
1.2.3	Implementation . . . . .	8
1.2.4	Testing . . . . .	9
1.2.5	Documentation . . . . .	11
1.2.6	Evaluation . . . . .	12
1.2.7	Maintenance . . . . .	13
1.3	Agile Development . . . . .	15
1.3.1	Rapid application development . . . . .	15
1.3.2	Agile development . . . . .	16
1.3.3	Comparing methodologies . . . . .	17
1.4	Learning points . . . . .	18
1.5	End of topic test . . . . .	19

---

### Prerequisites

You should already know that:

- software development follows a defined process:
  - Analysis
  - Design
  - Implementation
  - Testing
  - Documentation
  - Evaluation

### Learning objective

By the end of this topic you will be able to:

- understand the iterative nature of the software development process;
- describe the seven stages in the traditional software development process: analysis; design; implementation; testing; documentation; evaluation; and maintenance;
- describe how agile methodologies are used to develop software;
- compare iterative and agile methodologies.



## 1.1 Revision

### Quiz: Revision

[Go online](#)

**Q1:** Which stage of the software development process is missing?

Analysis, Design, \_\_\_\_\_, Testing, Documentation, Evaluation.

- a) Fixing
- b) Implementation
- c) Re-analysis
- d) Specification

.....

**Q2:** During the design stage, which of the following would not be done?

- a) Wireframing a user interface.
- b) Creating a structure diagram of the program modules.
- c) Agreeing the functional requirements with the users.
- d) Deciding on variables and arrays required to store data.

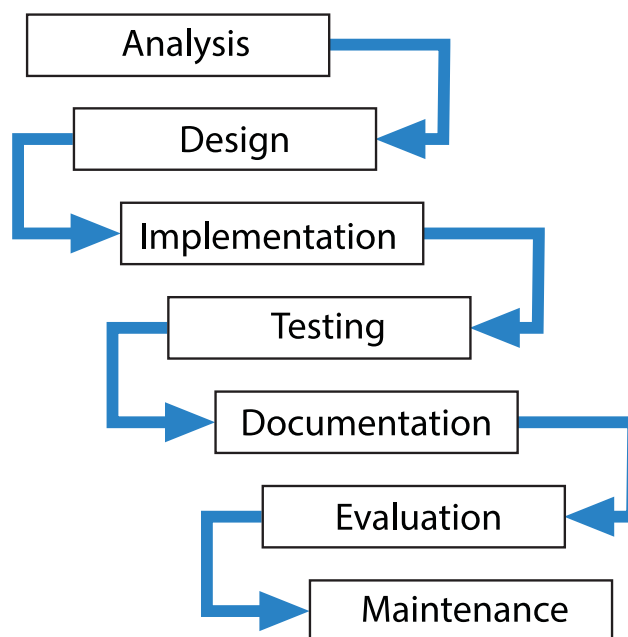
## 1.2 An iterative software development process

### Learning objective

By the end of this section you will be able to:

- understand the iterative nature of the software development process;
- describe the seven stages in the traditional software development process: analysis; design; implementation; testing; documentation; evaluation; and maintenance.

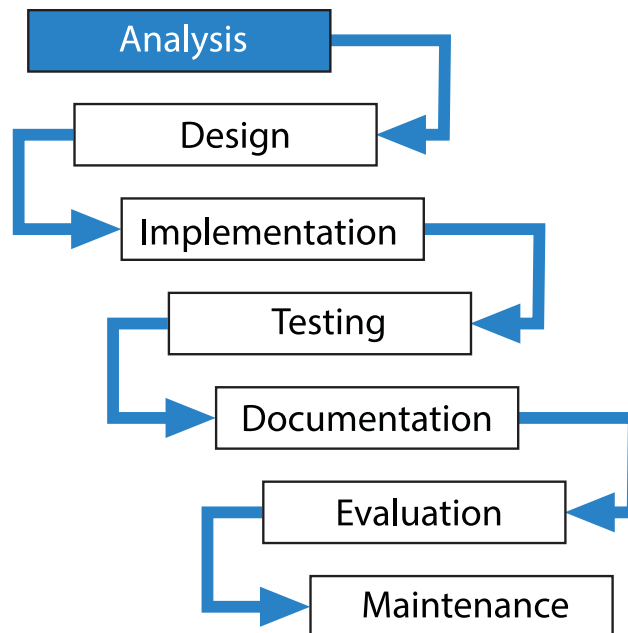
In this topic we are going to look at the seven stages in the software development process. These stages are:



The traditional software development process is often called the waterfall model because the development is seen as a series of developments flowing down from the analysis stage to the final evaluation stage.

This model has evolved as software developers have attempted to reduce the time and money spent creating and maintaining the increasingly complex applications they are being asked to create. Commercial software development is a process undertaken by a team of people who need to be able to work together in a structured and efficient way. The waterfall model involves constant revision and evaluation at every phase which makes it an iterative process. This ensures quality and efficiency in the final product. Large scale commercial software projects have traditionally followed this development process as far as possible in order to create successful product.

### 1.2.1 Analysis



The **analysis** stage of software development is the stage where an initial description of the problem to be solved is examined and turned into a precise description of exactly what the software will be able to do. Analysis of a problem is usually carried out by a Systems Analyst whose job is to take the description of the problem provided by the client and turn it into a software specification which can be used by the development team to create the completed application.

The **Systems Analyst** must be able to communicate effectively with the client in order to discover exactly what problem is that they want to solve, and also be able to communicate with the development team in order to accurately describe what needs to be produced. This is often more difficult than you might think for a number of reasons.

- The client may not be able to describe the problem they want solved accurately enough to convert directly into a clear description of a piece of software.
- The client may have unrealistic ideas of what is possible, or not be aware of what might be possible.
- In a large organisation there is often no one single person who knows exactly how every part of the system operates, or understands exactly what information needs to flow from one part of the organisation to another.

The Systems Analyst will collect as much information as they can about the organisation and the problem they want solved, because they need to know how the existing system works in order to design a solution which will work with the new one. As a result of their research they will create a **software specification** which accurately describes exactly what the software will be able to do, and will often also describe how long it will take to build and how much it will cost. This software specification is a **legally binding document** which can be used by either party to resolve possible disputes in the future. For this reason it is very important indeed that the document accurately describes what the client requires, and that they fully understand its contents.

Note that the software specification describes *what the software will be able to do*, not *how it does it*. That is the responsibility of the people who undertake the design stage.

Errors made or shortcuts taken at this stage in the software development process can have disastrous effects on subsequent stages.

### Quiz: Analysis

Go online

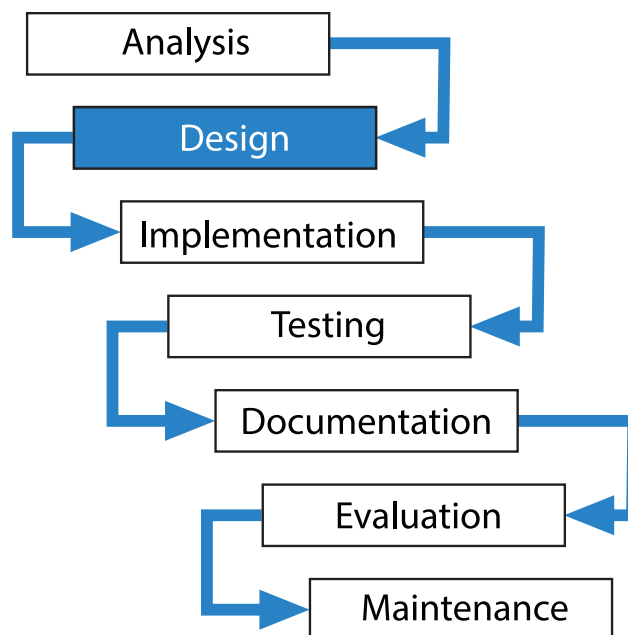


**Q3:** Why is the analysis stage of software development important?

.....

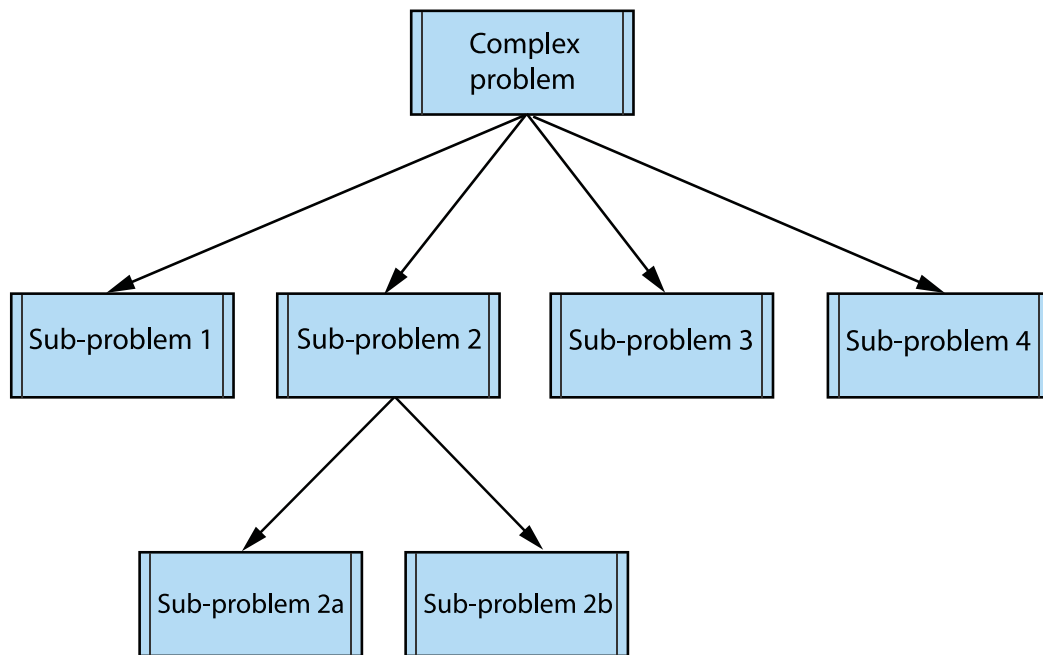
**Q4:** The Systems Analyst will create a software specification at the end of the Analysis stage of software development. What is a software specification?

### 1.2.2 Design



The **design** stage of software development is when the software specification created by the systems analyst is turned into a design which can be used by the team of programmers to write the code. The more time spent on this stage, the less time will be needed to be spent on the next one.

In theory any software problem can be broken down into smaller more easily managed problems. These problems can subsequently be further broken down into a set of simple steps. This process is often referred to as **Top Down Design and Stepwise Refinement**. Unfortunately things are not always as simple as this; as knowing how to break a problem down into smaller sub-problems takes practice. If it can be done successfully, a structure diagram will usually be created showing how the different sub-problems relate to each other.



When a problem is broken down into smaller sub-problems, the task becomes more manageable because each part can be worked on separately. This is called modular design. There are several advantages to this system:

- Different modules can be worked on simultaneously by separate programming teams.
- Modules can be tested independently.
- A well written module may be able to be re-used in another application.
- Modules can mirror the structure of the data being processed by the application. For instance a school management system may have a timetable module, a registration module, and an assessment module.

Once the data structures have been decided upon the flow of data around a system may be represented in a **data flow diagram**.

Once the structure of the program and its sub modules has been determined, the detailed logic of each component will be designed, using pseudocode.

If the pseudocode is written clearly and is thoroughly tested by working through the logic manually, then creating source code from it should be a relatively simple process.

### **Stepwise Refinement**

The process of designing the logic of each module is known as **stepwise refinement**. This is a process of breaking the module down into successively smaller steps, eventually resulting in a set of pseudocode instructions which can be converted into the chosen programming language.

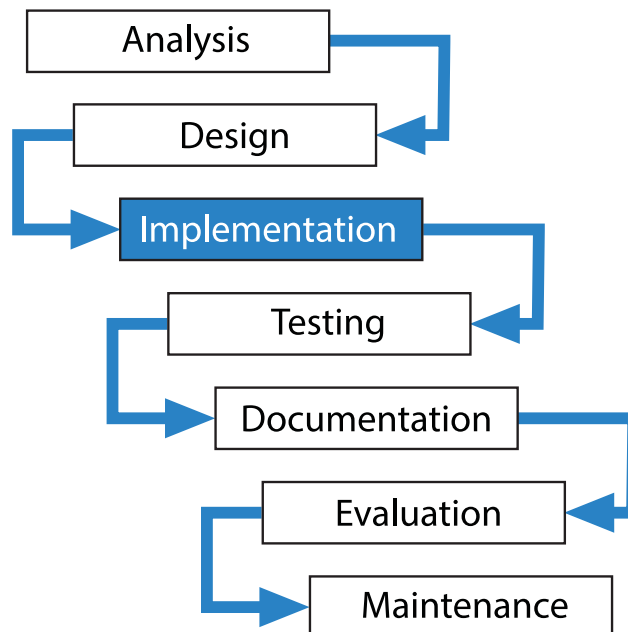
```

1  PROCEDURE module1()
2    <menu>
3    <step_2>
4    <step_3>
5    <step_4>
6  END PROCEDURE
7
8  PROCEDURE menu()
9
10   SEND "Menu: press H for help or C to continue"
11   RECEIVE userInput FROM (STRING) KEYBOARD)
12   WHILE userInput ≠ ["H"] AND userInput ≠ ["C"] DO
13     SEND "Please press H or C " TO DISPLAY
14     RECEIVE userInput FROM (STRING) KEYBOARD
15   END WHILE
16
17  END PROCEDURE

```

Each step may be further broken down into sub steps until the logic of the program is complete. You will learn more about techniques to design software in Topic 3, Design.

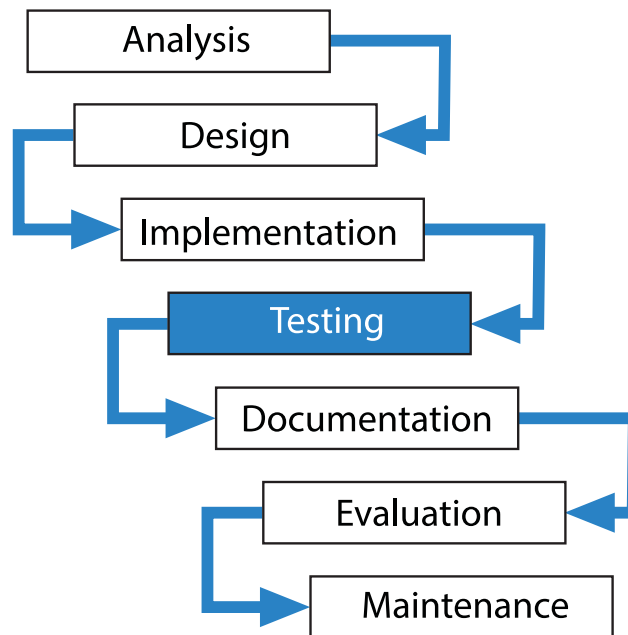
### 1.2.3 Implementation



Just as the design stage depends upon how well the analysis stage has been done, the **implementation** stage depends very much upon how clear the design is. If problems are spotted at the implementation stage, then this may well mean that the original design needs to be re-examined. This process of looking back to a previous stage to solve problems encountered in a subsequent one is what makes the software development process an iterative one.

If the design has been written in enough detail, the implementation stage should just be a matter of using the design to create the source code for the application.

### 1.2.4 Testing



Testing a piece of software has several purposes. It should check that:

- the software meets the specification;
- it is robust;
- it is reliable.

At the design stage, a set of test data will have been created which can be used to determine whether the software meets the specification or not, so that it is possible to see if the program does what it is supposed to do.

Modern programs are so complex that testing can never show that a program is correct in all circumstances. Even with extensive testing, it is almost certain that undetected errors exist.

**Testing can only demonstrate the presence of errors, it cannot demonstrate their absence.**

As far as possible, testing should be both **systematic**, which means testing in a logical order, and **comprehensive** which means testing every possible scenario.

#### Test data

When you are testing software, it should be tested with three types of data.

- **Normal data:** data that the program is expected to deal with.
- **Extreme data:** data that represents the values at the boundaries of the range of data it should accept. For instance if a program should only accept numbers between and including 1 and 100 then it should be tested with 1 and 100.
- **Exceptional data:** data that lie beyond the extremes of the program's normal operation; these should include a selection of what might be entered by accident or misunderstanding, so if a program should only accept whole numbers it should be tested with text input and decimal values. Exceptional data should also include data which is just **outside** the boundaries of the range of data it should accept.

**Activity: Testing**

Go online

**Q5:**

Match **three** of the following phrases to their correct descriptions:

1. data that is invalid;
2. data to test the extremes of a program's operations;
3. Abnormal data;
4. data that the program has been built to process;
5. data which lies beyond the extremes of normal operations.

Descriptions:

- Normal
- Extreme
- Exceptional

.....

**Q6:** Give examples of normal, extreme and exceptional test data for a program that should accept a numerical value for the months of the year.

Testing should be **systematic** and the results recorded so that time is not wasted repeating work done already, and the developers have a clear list of what has and what hasn't been tested. Test results should be documented in a clear list matching test data with results.

This kind of testing of a program within the organisation is called alpha testing.

Alpha testing will hopefully detect any logical errors in the code and if there are any discovered this will result in that part of the program being looked at again by the programming team and corrected. It will then have to be re-tested.

Alpha testing does not have to wait until an application is complete. It may be done on modules or on parts of an application while other parts are still being developed.

Once the software has been fully tested and corrected by the software developers, the next stage is to test it in the environment which it has been designed for. This is called beta testing.

Beta testing is important for a number of reasons:

- it is essential that the client is able to test the software and make sure that it meets the specification they agreed to at the analysis stage;
- although the programming team will have tested the software with appropriate test data, it is can be difficult for the programming team to test their work as a user who might make unpredictable mistakes rather than as a developer who is very familiar with the application they have been building;
- it is important that the people who are actually going to use the software are able to test it.

People involved in beta testing will send back error reports to the development team. An error report is about a malfunction of the program and should contain details of the circumstances which lead



to the malfunction. These error reports are used by the development team to find and correct the problem.

**Quiz: Testing (5 min)**

Go online



**Q7:** Which of the following statements are true of **alpha** testing of an application?

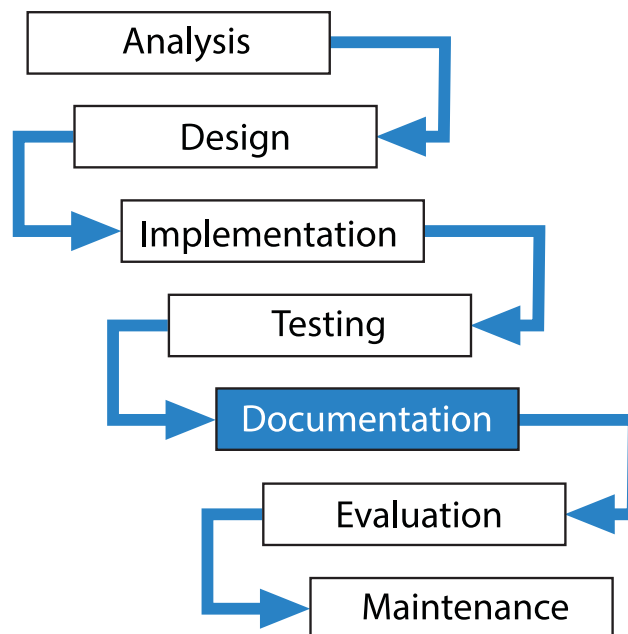
- a) Testing is done by the users.
- b) Testing is done by the programmers responsible for the application.
- c) Testing is done by specialist companies.
- d) Testing is done by the client.
- e) Testing may be done on parts of the application.

.....

**Q8:** Which one of the following statements describe **beta** testing?

- a) The testing is performed by the clients.
- b) The testing is more rigorous than alpha.
- c) The testing is for market research.
- d) The testing is performed by specialist companies.

**1.2.5 Documentation**



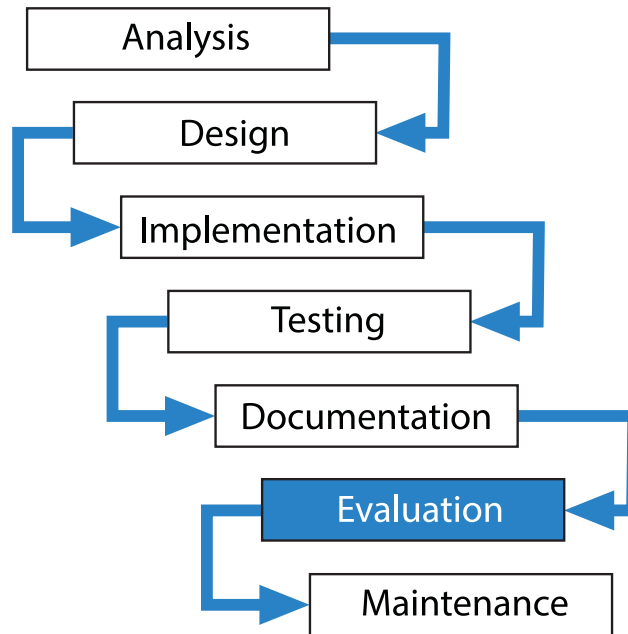
The **documentation** stage is when the guides to using and installing the software are produced. The documents created during the previous stages such as the software specification and test history are also collected together so that they can be referred to in case of changes or problems discovered at a later date.

The **user guide** for the software should include a comprehensive guide to the menu options, and how each one functions.

The **technical guide** for the software should include details of the minimum specification of

hardware required such as available RAM, processor clock speed, hard disk capacity and graphics card specification. It will also specify the platform and operating system versions which it is compatible with and any other software requirements or incompatibilities. The technical guide will give details of how to install the program, and if it is to function on a network, where to install it and how to licence it.

### 1.2.6 Evaluation

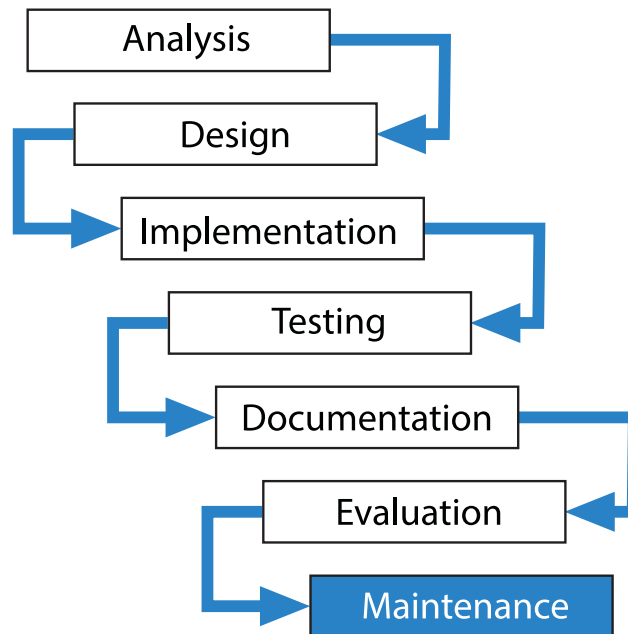


The evaluation stage is where the software is critically examined by both the client and the developer. The single most important criterion for evaluating software is whether it is fit for purpose i.e. does it match the software specification written at the analysis stage. It will also be evaluated against the following criteria:

- Robustness
- Maintainability
- Efficiency
- Usability

The evaluation is useful to the client because once it is complete, they can be sure that they have the software they need, and useful to the developer because if any problems are found at this stage it can save work later on. An evaluation can also help the developer improve their performance for future projects.

### 1.2.7 Maintenance



There are **three** types of software **maintenance**:

- **Corrective**
- **Adaptive**
- **Perfective**

#### **Corrective maintenance**

Is concerned with errors that were not detected during testing but which occur during actual use of the program. Users are encouraged to complete an error report, stating the inputs that seemed to cause the problem and any messages that the program might have displayed. The developer will be responsible for any costs incurred by this type of maintenance.

#### **Adaptive maintenance**

Is necessary when the program's environment changes. For example, a change of operating system could require changes in the program, or a new printer might call for a new printer driver to be added. A change of computer system will require the program to have its code adapted to run on to the new system. The cost of adaptive maintenance is usually borne by the client but there may be negotiation depending upon how predictable the change in circumstances was.

#### **Perfective maintenance**

Occurs when the client requests changes or improvements to the program which were not in the original software specification. This may be due to changes in the requirements or new legislation. Such maintenance can involve revision of the entire system and can be expensive. The cost of perfective maintenance is likely to be borne by the client.

**Activity: Maintenance**

Go online

**Q9:**

Match **three** of the following phrases to their correct descriptions:

1. only done under extreme circumstances;
2. errors removed that were initially undetected;
3. requirements incorrect;
4. occurs in response to requests to add new features;
5. needed when environment changes.

Descriptions:

- Corrective
- Adaptive
- Perfective

**Activity: Waterfall model**

Go online

**Q10:**

Place these terms in the correct order and match them to their correct definition:

1. Checking to see how well the software meets its specification
  2. Writing the source code
  3. Looking at the problem and collecting information
  4. Fixing problems and adapting the software to new circumstances
  5. Creating a structure diagram and pseudocode
  6. Trying to find ways in which the program will fail
  7. Creating a user guide and technical guide
- 
1. Documentation
  2. Design
  3. Analysis
  4. Implementation
  5. Maintenance
  6. Testing
  7. Evaluation

## 1.3 Agile Development

### Learning objective

By the end of this section you will be able to:

- describe how agile methodologies are used to develop software;
- compare iterative and agile methodologies.

### 1.3.1 Rapid application development

Although the waterfall model has traditionally been the one used for large scale software development projects, it has often been criticised as being too rigid and too slow a process, resulting in projects where the software specification had to be changed substantially during the lifetime of the project, or software became out of date before it was even complete. In theory the analysis stage should result in a software specification which can then be used throughout the rest of the project, but in practice this is often unrealistic.

**Rapid Application Development** means that the users should be involved at all stages of the development process and that changes to the design can be made at any time throughout the life of the project. It also means that the development process is faster and more flexible.

There are **four** stages to the Rapid Application Development model:



As you might guess from the name, the emphasis of Rapid Application Development is on creating software quickly and efficiently. It is however, still considered to be more appropriate to smaller software projects.

### **1.3.2 Agile development**

Agile software development is a product of the Rapid Application Development idea, with an emphasis on small scale developments, and with teams of people who have a flexible approach to change in requirements. Agile software development is seen as a more adaptable and responsive process compared to the rigid waterfall model. Agile development has been widely seen as being more suitable for certain types of environment using small teams of experts such as web development.

The benefits of the Agile method are:

- reduced development time;
- increased responsiveness to changing circumstances;
- more reduced costs due to the efficiency of using small groups of developers.

Agile development will make use of prototyping where working models of the proposed system are tried out and tested throughout the development process so that client feedback can be taken into account as early as possible. Developers and clients will use tools such as version management software and online ticket systems to keep track of issues and bugs and give feedback on progress.

A common feature of agile development is the frequent appearance of updates to the software, often given sequential version numbers.

There has been some criticism of the agile software development process as being too extreme a contrast to the tried and trusted waterfall model, or as just a management fad that simply describes existing good practices under new jargon, and wrongly emphasizes method over results. Agile development can also mean that it encourages the client to make changes to the specification throughout the development process rather than thinking clearly about what they require at the beginning.

### 1.3.3 Comparing methodologies

Both methodologies have strengths and weaknesses. The table below summarises some of the key differences.

#### Iterative Methodology (traditional)

##### Strengths:

- Rigid planning structure
- Good for large teams
- Helps to plan and track large software projects
- Clear agreement on outcomes at start of project

##### Weaknesses:

- Very rigid approach does not deal well with mid-project changes
- Can over-complicate simple projects
- Unidentified issues at the analysis stage can be time-consuming and costly to fix
- Little involvement of client after analysis.

#### Agile Methodology

##### Strengths:

- Copes well with little cumulative changes as the project progresses.
- Good for small-scale projects like most Apps
- Ongoing involvement of client allows changes to be agreed quickly
- Changes cause less delay or can be tackled in the next version.

##### Weaknesses:

- Works best with small teams
- Needs close version control and tracking of changes
- Can be difficult to determine the full scope of the project in the early stages
- Usually no legally binding agreement at the start

## 1.4 Learning points

### Summary

- The traditional waterfall model of software development consists of seven stages: analysis, design, implementation, testing, documentation, evaluation, and maintenance.
- Software development is an iterative process.
- In the analysis stage the client's project description is carefully examined and after discussion, a legally binding software specification is written.
- In the design stage, top down design and stepwise refinement is used to turn the software specification into structure diagrams, pseudocode and a data structures which the programming team can use.
- In the implementation stage the programming team use the design to create and debug the program code.
- In the testing stage the code is alpha tested using normal, extreme and exceptional test data, and then beta tested using clients or individuals outside the development team organization.
- In the documentation stage the user guide and technical guide are produced.
- In the evaluation stage the software is examined to see if it is reliable, robust, maintainable, efficient and user friendly.
- The maintenance stage is where problems are fixed, the software may be adapted to new circumstances and additional features may be added.
- Rapid Application Development (RAD) is an attempt to streamline the waterfall model by using prototyping and involving the client at more stages in the development process.
- Agile programming is a type of RAD suited to smaller projects. It is designed to be as flexible as possible where the specification may change throughout the development process resulting in reduced development time and costs.



## 1.5 End of topic test

### End of topic 1 test

[Go online](#)

**Q11:** Which one of these would not be found in the technical guide?

- a) Operating system required
  - b) Hardware requirements
  - c) Memory requirements
  - d) Tutorial
- .....

**Q12:** During the software development process, which one of the following is responsible for converting the design into actual program code?

- a) Programmers
  - b) Systems Analyst
  - c) Independent test group
  - d) Client
- .....

**Q13:** Which of these has the waterfall model stages in the right order?

- a) Analysis, Design, Implementation, Documentation, Testing, Evaluation, Maintenance
  - b) Analysis, Design, Implementation, Testing, Documentation, Evaluation, Maintenance
  - c) Analysis, Design, Evaluation, Implementation, Testing, Documentation, Maintenance
  - d) Analysis, Design, Implementation, Testing, Documentation, Maintenance, Evaluation
- .....

**Q14:** Which one of these is NOT an advantage of agile software development.

- a) Reduced development time
  - b) Responsiveness to changed circumstances
  - c) Reduced costs
  - d) Reduced time spent on analysis
- .....

**Q15:** Which of the following is a strength of Agile Development?

- a) High level of involvement of the client/customer.
- b) Lack of feedback from customers.
- c) Easy for large teams to work on an agile project.
- d) Clear scope and boundaries from the start of the project.



---

## Topic 2

# Analysis

---

### Contents

2.1	Revision . . . . .	23
2.2	Inputs, processes, outputs . . . . .	23
2.3	Purpose, Scope, Boundaries and Functional Requirements . . . . .	25
2.3.1	Purpose . . . . .	25
2.3.2	Scope and boundaries . . . . .	25
2.3.3	Functional requirements . . . . .	26
2.4	Analysing a program requirement . . . . .	27
2.5	Learning points . . . . .	28
2.6	End of topic test . . . . .	29

---

### Prerequisites

You should already know that:

- Most programs you will have written at National 5 followed a pattern of input-process-output.
- The overall function of a program is called its purpose.
- Individual elements of a program that are important to its operation are called functional requirements.

### Learning objective

By the end of this topic you will be able to:

- identify the inputs, processes and outputs of a program that you are asked to implement;
- describe what is meant by purpose, scope, boundaries and functional requirements in relation to software development;
- identify the purpose, scope, boundaries and functional requirements of a program that you are asked to implement.

## 2.1 Revision

**Quiz: Revision**

Go online



**Q1:** Which description correctly describes the term 'functional requirement'?

- a) Whether the interface allows the user to use function keys for shortcuts.
- b) A detailed list of what the finished program must do.
- c) A detailed list of what the finished program should not contain.
- d) A list of errors to check for during testing.

.....

**Q2:** Programs work following a pattern of:

- a) input —process —output
- b) output —input —process
- c) process —input —output
- d) input —output —process

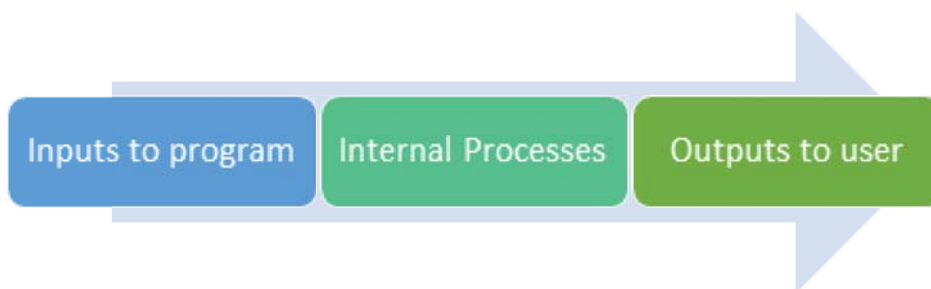
## 2.2 Inputs, processes, outputs

**Learning objective**

By the end of this section you will be able to:

- identify the inputs, processes and outputs of a program that you are asked to implement.

Most computer programs follow a simple process of inputs, processes, outputs. This is —essentially —what a computer is designed for after all!



**Inputs** can come from a variety of sources, for example:

- A user typing data into a text box on screen;
- A file containing the necessary data;
- Sensors from the hardware the program is running on (e.g. mapping software on a smartphone will make use of the position data from the GPS receiver).

A **Process** is any action that the program will need to carry out on the data to create new data. Examples include:

- Calculations (e.g. using arithmetic operators to compute an answer);
- Comparisons (e.g. checking if a value is greater than another in an if statement).

**Outputs** in traditional programming will often be displayed on screen for the program's user, but can also be given in many ways:

- Output data to a file;
- Activating an LED; (e.g a warning light);
- Operating a motor or piston (e.g. engaging the ABS braking system on a car).

### Exercise: Identifying inputs, processes and outputs (10 min)

Go online



Read the program description below and identify the inputs, processes and outputs required for the program to function.



"An app is being designed to calculate the Body-Mass Index (BMI) of an adult. BMI is calculated using the formula:  $\text{weight}(\text{kg}) \div \text{height}(\text{m})^2$ .

Results in the range less than 18.5 should give a message that the user is underweight, 18.5 to 25 should be ideal weight, 26 to 30 is overweight and greater than 30 should warn that they are obese."

Inputs	Processes	Outputs

## 2.3 Purpose, Scope, Boundaries and Functional Requirements

### Learning objective

By the end of this section you will be able to:

- describe what is meant by purpose, scope, boundaries and functional requirements in relation to software development.

In an earlier section, you learned about the Analysis stage of an iterative development process. The outcome of the analysis stage is a legally-binding document called the software specification.

As part of the software specification, four areas are analysed and agreed upon: the overall **purpose**, the **scope**, **boundaries**, and **functional requirements**.

### 2.3.1 Purpose

The **purpose** of a program is a general statement about what the program is required to do. It should clarify the main point(s) of the program between the developers and clients. At this stage the purpose does not need to be highly detailed —the scope, boundaries and functional requirements will give this —but it does have to be accurate: it will form part of the contract!

Consider the description from the activity in an earlier section:

*"An app is being designed to calculate the Body-Mass Index (BMI) of an adult. BMI is calculated using the formula:  $weight(kg) \div height(m)^2$ .*

*Results in the range less than 18.5 should give a message that the user is underweight, 18.5 to 25 should be ideal weight, 26 to 30 is overweight and greater than 30 should warn that they are obese."*

This is a good example of a purpose statement; it encompasses all the key things the program has to do, but is not bogged down in minute details.

### 2.3.2 Scope and boundaries

It is very important at the outset to establish clearly the scope and boundaries of the project. Scope and boundaries are opposite sides of the same "coin". Between them, they give a precise description of the extent of the project.

Here is a typical statement about scope and boundaries. You will find similar statements by searching the web. "The project scope states what will and will not be included as part of the project. Scope provides a common understanding of the project for all stakeholders by defining the project's overall boundaries."

One way of thinking about it is:

- the scope clarifies what the project must cover
- the boundaries clarify what the project will not cover.

For example, suppose your project was to develop an expert system giving students guidance on job opportunities which they should consider after graduating from University.

The scope of the project would be to create an expert system. Then it would be necessary to describe the range of jobs and degrees that would be included in the system, the level of information

that would be output by the system (does it suggest contact addresses as well as simply job types), the types of questions that the user will be asked. Does it cover all degrees, or is it only for students with Computing Science degrees, and so on ... All these things will define the **boundaries** of the system.

Sometimes it is also helpful to spell out exactly what will NOT be covered. So, for example, a clear statement could be made which states that the system will NOT cover advice on jobs for those with medical and veterinary degrees, or jobs overseas.

The scope and boundaries could also refer to technical issues. For example, they might state that the resultant system will run on any computer capable of running any version of Windows after Windows 7, but not on any other operating system.

The scope and boundaries can also cover other real-world issues like timescales and costs, which will have to be agreed as part of the development.

An example of scope and boundaries from the BMI program might be to more closely define what is meant by 'adult'. (This may require the client or developer to seek outside expert advice from a medical professional at this point.)

We could specifically say:

- "The program should work with adults who are aged above 20" (scope) or
- "the program should not work with children and adolescents 19 and under" (boundary).

In a later section, you will be asked to draw up further scope and boundary statements for this program.

### 2.3.3 Functional requirements

The scope and boundaries define clearly the extent of the project. It is like drawing an outline map of a country. Now you have to fill in some detail inside the outline. The best way to do this is to produce a list of .

Function requirements = what the product must do!

For example, if the project was to write a program to calculate the cost of sending a parcel, the list of functional requirements might include:

- attractive welcome screen
- all options available as clickable buttons on screen
- user input of destination, weight and dimensions of parcel
- user verification of all inputs
- output displayed on screen, and spoken through speakers
- all colours and fonts complying with latest guidance on accessibility
- and so on ...

Normally, functional requirements can be organised into inputs, processes and outputs for everything relating to the program operation. E.g. "Input: height. Should be a decimal number between 1.2 and 2.2. User will choose via a listbox in order to validate input."



Again, you will be asked to draw up further functional requirements in the next section.

## 2.4 Analysing a program requirement

### Learning objective

By the end of this section you will be able to:

- identify the purpose, scope, boundaries and functional requirements of a program that you are asked to implement.

### Exercise: Software specification (30 min)

Go online



Read the expanded information below about the piece of software being developed to calculate a user's BMI. You will be asked to report on the following items:

- Purpose of the program
- Scope
- Boundaries
- Functional Requirements.

You can also make use of reliable sources of information from the web to help draw up your report.

Your report should include 4 headings for purpose, scope, boundaries and functional requirements. The functional requirements should be grouped into inputs, processes and outputs.

Your tutor will evaluate your report at the end of the task.

#### Description:

*"An app is being designed to calculate the Body-Mass Index (BMI) of an adult. BMI is calculated using the formula:  $\text{weight}(\text{kg}) \div \text{height}(\text{m})^2$ .*

*It should work with heights between 1.2 —2.2 meters and weights from 40kg up to 120kg. Numbers outwith this range should be advised to speak to a medical professional.*

*Results in the range less than 18.5 should give a message that the user is underweight, 18.5 to 25 should be ideal weight, 26 to 30 is overweight and greater than 30 should warn that they are obese."*

*The client is eager to ensure that users are aware that BMI is a rough tool and that other factors can affect the result. They would like a splash-screen included that asks users to acknowledge this and give a link to a reliable UK-based website for further information.*

## 2.5 Learning points

### Summary

- Programs usually work via a system of input-process-output: input is received from the 'real world', the inputs are processed by the program and the results are outputted to the user.
- The purpose of a program is a clear and accurate description of what the program is to achieve.
- The scope is a collection of statements that explain what the program should be able to do or work with.
- The boundaries are the opposite of scope; what the program should not be able to do or work with.
- Functional requirements are clear statements explaining exactly what every input, process or output in the program does.

## 2.6 End of topic test

### End of topic 2 test

[Go online](#)

**Q3:** What is meant by the 'scope' of a program?

- a) The overall description of what a program is supposed to do.
- b) What data a program will not accept.
- c) What the program should be able to do.
- d) An exact description of what each input, process and output is.

.....

**Q4:** The boundaries of a program can be thought of as the 'opposite' of the scope?

- a) True
- b) False

.....

**Q5:** A program is being written to control a security door activated by a fingerprint. Which of the following would be a process?

- a) Reading the fingerprint data from the sensor.
- b) Checking the fingerprint data against a database of authorised users.
- c) Changing the red LED light to a green LED light on the fingerprint sensor when successfully read.
- d) Activating the electric motor to open the door.

.....

**Q6:** Functional requirements are:

- a) A set of precise statements covering each input, process and output.
- b) A brief description of who will use the program.
- c) The timescale and costs of a software project.
- d) What the program should not do.



---

## Topic 3

# Design

---

### Contents

3.1	Revision . . . . .	33
3.2	Introduction . . . . .	34
3.3	Structure diagrams . . . . .	34
3.3.1	Levels of design . . . . .	36
3.3.2	Data flow . . . . .	37
3.4	Pseudocode . . . . .	38
3.4.1	Example: Pseudocode . . . . .	38
3.4.2	Levels of design . . . . .	38
3.4.3	Data flow . . . . .	40
3.5	User interface design . . . . .	41
3.6	Learning points . . . . .	43
3.7	End of topic test . . . . .	44

---

**Prerequisites**

From your studies at National 5 you should already know:

- that **structure diagrams** and **flow charts** are graphical representations of the logic of a program;
- that **pseudocode** is an informal high-level description of how a computer program functions;
- how to use at least one of the above notations to design a program;
- that **wireframes** can be used to design the user interface of a program;
- that user inputs and the result of processes can be stored in **variables** and/or **arrays** within a program.

**Learning objective**

By the end of this topic you will be able to:

- explain how design notations can help the software development process;
- read and understand program designs that have been produced in both pseudocode and structure diagrams;
- design your own programs in both pseudocode and structure diagrams;
- include top level algorithm, refinements and data flow in your program designs;
- create wireframes of your user interfaces, showing inputs and outputs to/from your programs.

### 3.1 Revision

**Quiz: Revision**

Go online



First 2 questions refer to the following:

```

1 SET total TO 0
2 SET count TO 0
3 WHILE count < 10 DO
4     RECEIVE nextInput FROM (INTEGER) KEYBOARD
5     SET total TO total + nextInput
6     SET count TO count + 1
7 END WHILE
8 SEND total / 10 TO DISPLAY
    
```

**Q1:** The above is an example of?

- a) Pseudocode
- b) Source code
- c) Machine code
- d) High level language code

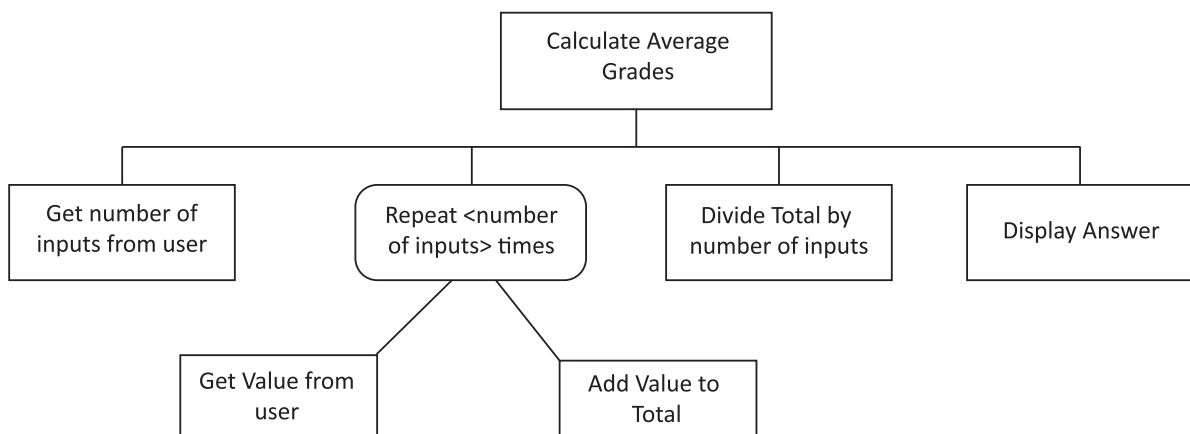
.....

**Q2:** How many numbers will be input into this program?

- a) 1
- b) 9
- c) 10
- d) 11

.....

**Q3:** What type of design notation is this?



- a) Pseudocode
- b) Structure Diagram
- c) Flow Chart
- d) Flow Diagram

## 3.2 Introduction

### Learning objective

By the end of this section you will be able to:

- explain how design notations can help the software development process;
- describe a variety of graphical program design notations including **structure diagrams**, **pseudocode** and **wireframes**.

A software specification describes what a program must do. The design stage of the software development process is where a set of documents is created that describe how the program will do it. These documents might describe the **structure of the program** in terms of different modules, the **pseudocode** between these modules and the **detailed logic** of the modules themselves. It makes sense to discuss the **user interface** at an early stage in the design process as well.

## 3.3 Structure diagrams

### Learning objective

By the end of this section you will be able to:

- read and understand a structure diagram;
- design a program using a structure diagram;
- include the top-level algorithm, refinements and data flow in a design.

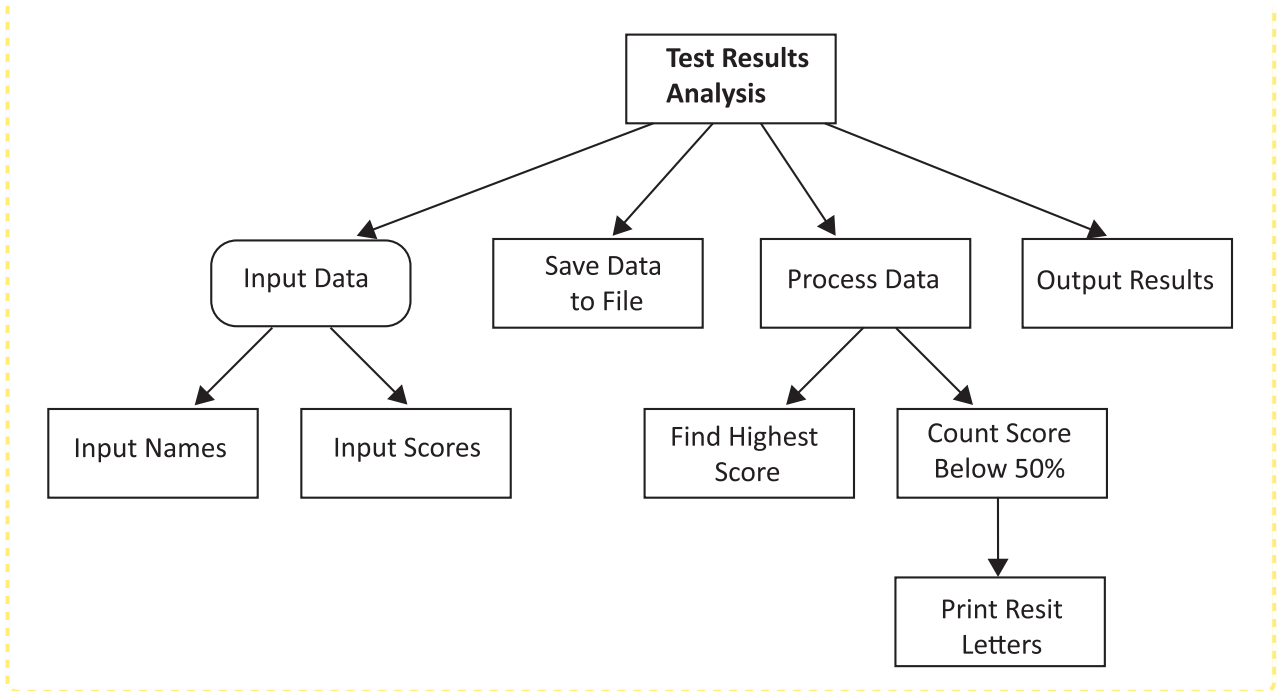
A **structure diagram** will be created as part of the top down analysis of the software specification. This allows the developers to break this complex problem description into a series of smaller sub-problem descriptions. These sub problems can be regarded as modules within the system and they themselves may be further divided into smaller (and hopefully simpler) sub problems.

### Example : Structure diagram

A program is required to take in a set of test results, save the data to file, calculate the highest score and how many failures there were. It should also print re-sit letters for all those candidates who failed.

This problem can be broken down into four main modules, two of which can be further broken down into sub tasks.









A structure diagram is organised to show the level or hierarchy of each sub task. The sequence of operations in the program is read from top to bottom, going left to right.

Once the structure of the program has been decided, the next step is to work out what data each module will need and what data it will pass on to the next module. This can be shown either by annotating the structure diagram with arrows representing data passing between each step.

You should already be familiar with some of the shapes used in a Structure Diagram from National 5:

Shape	Meaning
	Process
	Loop
	Selection
	Predefined function/procedure

### 3.3.1 Levels of design

In more complex programs, it is usual to work out the 'top-level' algorithm (main steps) of a program first, then continue to use structure diagram notation (or pseudocode) to refine each main step. Here is the above example again, this time with the top-level algorithm and each main step separated out.

Figure 3.1: Top-level Algorithm

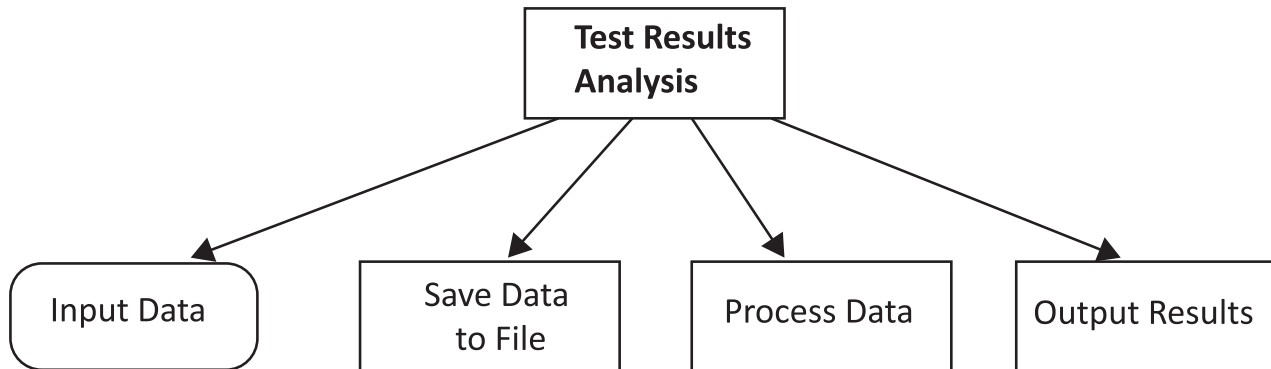


Figure 3.2: Refinement of 'Input Data' step

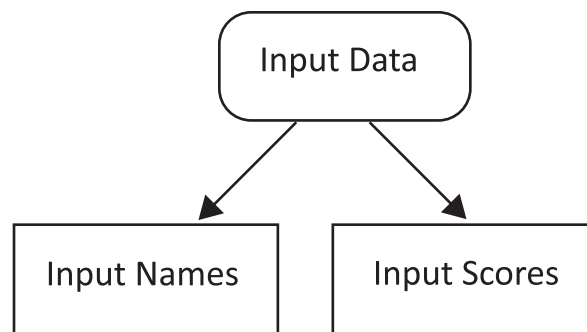
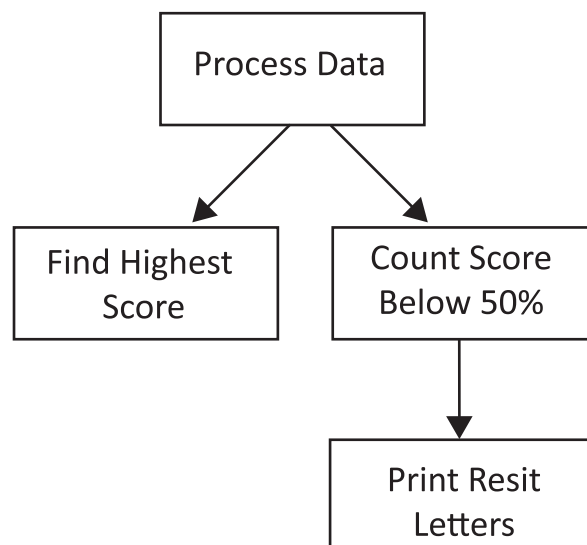


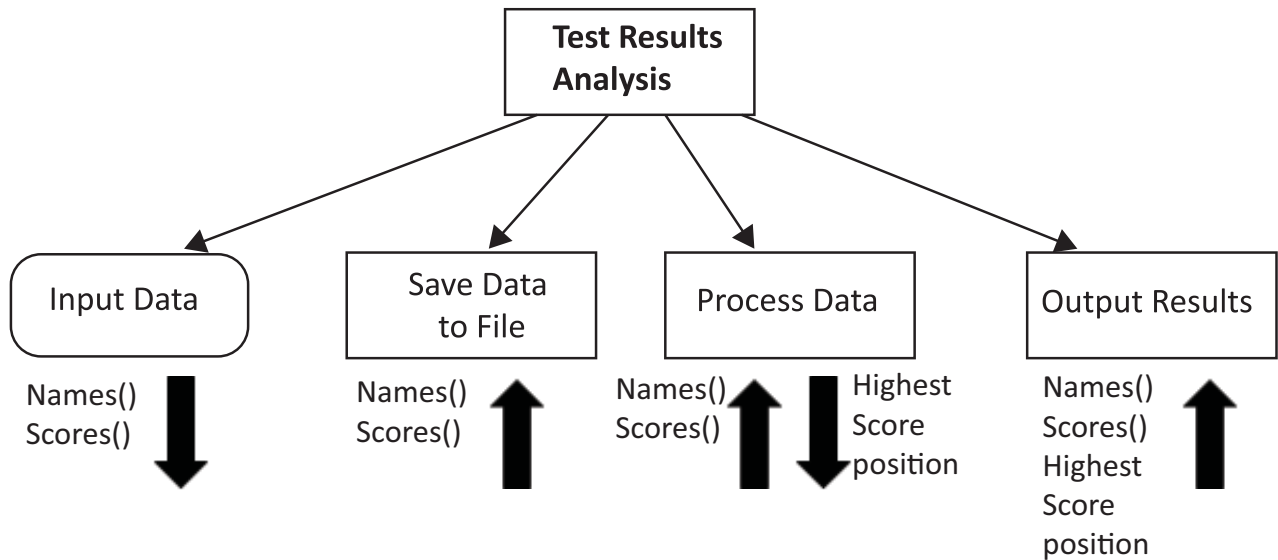
Figure 3.3: Refinement of 'Process Data' step



### 3.3.2 Data flow

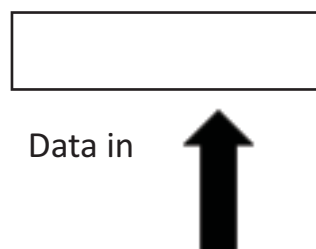
Figure 3.4 below has been annotated with data flow showing the inputs and outputs from the top-level algorithm.

Figure 3.4: Top-level Algorithm

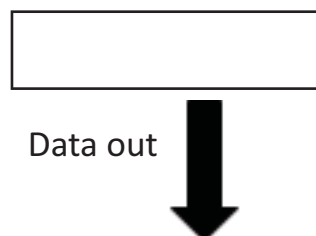


This shows the movements of items of data into and out of each part of the program. Note that these are internal movements, they are not items of data to/from the user themselves. Input and output from the user is normally shown in a **wireframe**.

Data going into each program module is shown by an arrow pointing *into* it:



Data going out of each program module is shown by an arrow pointing away from it:



As part of the course, you will learn to create modular programs by using subroutines and function in your own code. This will show you how to pass data between different parts of the program. This is covered in Topic 5.

## 3.4 Pseudocode

### Learning objective

By the end of this section you will be able to:

- read and understand pseudocode;
- design a program using pseudocode;
- include the top-level algorithm, refinements and data flow in your pseudocode.

Pseudocode is another commonly used design notation. This often closely follows the program logic and uses a hybrid language somewhere between English and a programming language; the focus is on the logic of what the program should do, not the exact syntax required to do this.

In the exam, SQA reference language is used to illustrate code: this is essentially a type of highly-specified pseudocode. Real pseudocode is usually much less formal than this! (*and —don't worry —no-one expects you to be able to write algorithms using the exact SQA reference language in the exam!*)

### 3.4.1 Example: Pseudocode

Pseudocode starts by working out the top-level algorithm and numbering each of the main steps. Using the program in section 3.3 as an example:

#### Test Results Analysis:

1. Input Data
2. Save Data to File
3. Process Data
4. Output Results

This is the top-level algorithm for the program —notice how it corresponds to the main steps in the structure diagram.

### 3.4.2 Levels of design

The top-level algorithm is obviously not very detailed. In reality we would want to refine this algorithm until each line of pseudocode roughly corresponds to a line of code in each program. To expand the pseudocode we follow the main step number with a period/full-stop, then a sub-step number.

Here are the refinements for the first 3 main steps:

**Input Data:**

```

1 SET counter TO 0
2 SET input_ok TO true
3
4 WHILE input_ok = true DO
5     RECEIVE names[counter] FROM (STRING) KEYBOARD
6     RECEIVE scores[counter] FROM (STRING) KEYBOARD
7     SEND "Another result? Yes/No" TO DISPLAY
8     RECEIVE response FROM (STRING) KEYBOARD
9     IF response = "No" THEN
10        SET input_ok TO false
11    END IF
12    SET counter TO counter + 1
13 END WHILE

```

**Save Data to File:**

```

1 CREATE "names.txt"
2 FOREACH name FROM names DO
3     SEND name TO "names.txt"
4 END FOREACH
5
6 CREATE "scores.txt"
7 FOREACH score FROM scores DO
8     SEND score TO "scores.txt"
9 END FOREACH

```

**Process Data:**

```

1 <find highest score>
2     <count scores below 50%>
3     <print resit letters>

```

Notice that steps 3.1 to 3.3 require further refinement. To do this we simply add another period/full-stop followed by sub-sub step numbers.

Here are the refinements for steps 3.1 to 3.3

**Find Highest Score:**

*# find highest score*

```

1 SET highest_score TO scores[0]
2
3 FOREACH score FROM scores DO
4     IF highest_score < score THEN
5         SET highest_score TO score
6     END IF
7 END FOREACH
8
9 SEND "The highest score was:" & highest_score TO DISPLAY

```

**Count Scores Below 50%:***# count scores below 50%*

```

1 SET total_failed TO 0
2 FOREACH score FROM scores DO
3     IF score < 50 THEN
4         SET total_failed TO total_failed + 1
5     END IF
6 END FOREACH
7 SEND "There were " & total_failed & " failed students" TO DISPLAY

```

**Print Re-sit letters:***# print re-sit letters*

```

1 SET counter TO 0
2 REPEAT
3     IF (score[counter] < 50) THEN
4         SEND "Dear " & names[counter] &
5             " Your resit is on " & now() + 14 TO <printer>
6     END IF
7 UNTIL names[counter] = ""
8 END REPEAT

```

**Activity: Pseudocode**

Go online



Can you write the pseudocode for the final step (4) of this program? The step should output a message to each student showing their own score, along with a statement of the highest score in the class. You can use a similar algorithm to step 3.3 for the student messages!

**3.4.3 Data flow**

Data flow in pseudocode is normally shown alongside the top-level algorithm beside each step. For example:

**Test Results Analysis**

1.	Input Data	<b>IN: n/a</b>	<b>OUT: names[], scores[]</b>
2.	Save Data to File	<b>IN: names[], scores[]</b>	<b>OUT: n/a</b>
3.	Process Data	<b>IN: names[], scores[]</b>	<b>OUT: highest_score</b>
4.	Output Results	<b>IN: names[], scores[], highest_score</b>	<b>OUT: n/a</b>

These input and output details should ultimately become parameters in the modular program that results from this. You will learn more about parameter passing in topic 5.

## 3.5 User interface design

### Learning objective

By the end of this section you will be able to:

- create wireframes of your user interfaces, showing inputs and outputs to/from your programs.

**Wireframes** are one of the techniques used for user interface design. The user interface of any software is the part which users experience and is therefore a crucial part of the design process. A wireframe is a visual guide that represents a website or program interface, and is normally created at an early stage in the development of an application to give the client and developers a clear idea of how the finished product will function and how users will interact with it.

Wireframes can be hand-drawn or computer drawn. Like pseudocode, there is no single standard for a wireframe design, but they will often have the following points in common:

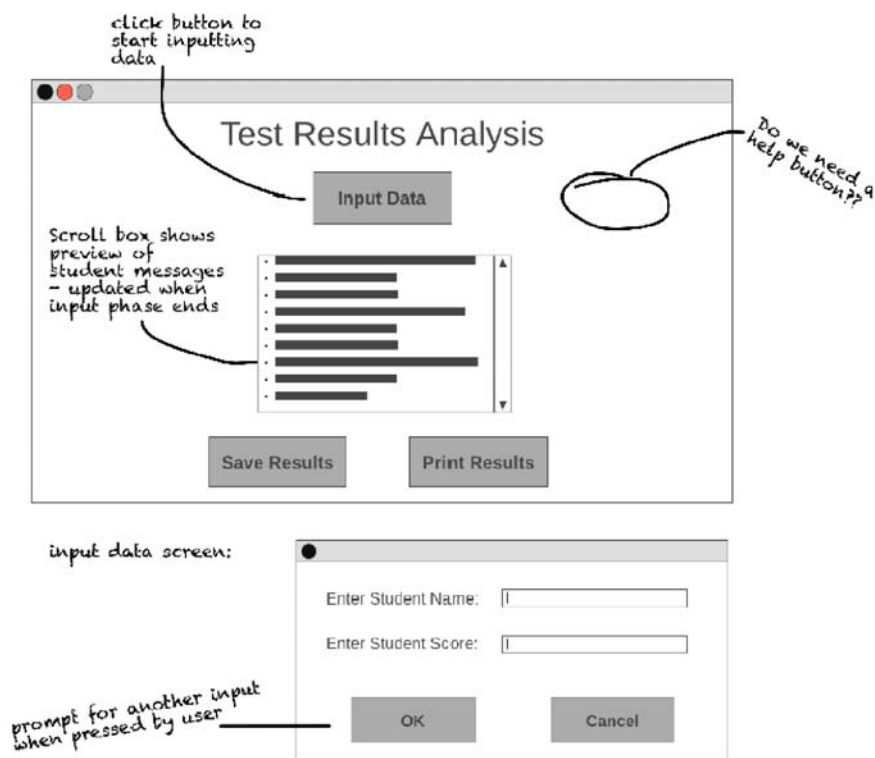
- A low level of 'artistic' detail —basic boxes and fonts are used to show the position of each element.
- Graphics are represented by stick-like sketches or just a box with a cross through it.
- Large areas of text (e.g. instructions) can be represented by wavy lines or dummy 'lorum ipsum' text. (See <https://www.lipsum.com/> for more!)
- Annotations describing actions, links, dependencies and behaviours can be added simply with arrows and text descriptions around the wireframe.

Wireframes should allow the client and developer to discuss possible user interface designs (and issues with them) without having to take time and effort to produce high-quality, realistic designs. The use of wireframing sits well with agile development methodologies, and once a layout is agreed upon the user interface is often built without any code, allowing the client to see it 'for real' on screen.

A number of programs are available to help create wireframes quickly. Two free options are:

- 'Pencil' (<https://pencil.evolus.vn/>) —installable on Windows, Mac and Linux.
- Wireframe (<https://wireframe.cc/>) —a simplistic, online app.

Here is an example wireframe for the Test Results Analysis program earlier.



### Exercise: Create a wireframe

Go online



Using suitable wireframing software (or pencil and paper!) create wireframe for the BMI calculator App you analysed in the last topic. Show your finished design to your tutor.



## 3.6 Learning points

### Summary

- A structure diagram is a graphical representation of a main program that has been divided into the top-level algorithm by a series of connected steps.
- Each top-level step can be further refined in a structure diagram through detailing further steps below.
- Data flow in a structure diagram can be shown by arrows annotated to each step.
- Pseudocode is a hybrid English/Code language used to detail the top-level algorithm and refine each into further steps.
- Steps in pseudocode are numbered using whole numbers. Refinements are numbered using the main step number, followed by a period then the sub-step number.
- Refinements often correspond directly with the source code of the resulting program.
- Data flow can be annotated to the steps in pseudocode by writing IN and OUT key words beside each step.
- Wireframes are a quick, informal sketch of a possible user interface.
- Wireframes are usually annotated with details such as interactivity, fonts, colours, links.

### 3.7 End of topic test

#### End of topic 3 test

[Go online](#)

**Q4:** Which one of these statements is true?

- a) It is not necessary to bother about the module names as these will change in the code.
  - b) The modules in a structure chart will become modules in the finished program.
  - c) Structure charts are not hierarchical.
  - d) A structure chart cannot show the data flow between modules.
- .....

**Q5:** Which one of these is **not** a graphical design notation?

- a) Wireframe
  - b) Structure diagram
  - c) Pseudocode
  - d) Data flow diagram
- .....

**Q6:** Which design notation would you use to design a user interface?

- a) Wireframe
  - b) Structure diagram
  - c) Pseudocode
  - d) Data flow diagram
- .....

**Q7:** Which design notation is the easiest to create source code from?

- a) Wireframe
  - b) Structure diagram
  - c) Pseudocode
  - d) Data flow diagram
- .....

**Q8:** There is only one correct way to write pseudocode.

- a) True
- b) False

---

## Topic 4

# Implementation: Data types and structures

---

### Contents

4.1	Revision . . . . .	47
4.2	Data types and pseudocode . . . . .	48
4.3	Simple data types . . . . .	49
4.4	Identifying simple data types . . . . .	50
4.5	Structured data types . . . . .	51
4.5.1	Arrays . . . . .	52
4.5.2	Strings . . . . .	52
4.5.3	Parallel arrays . . . . .	52
4.5.4	Records . . . . .	53
4.5.5	Handling records . . . . .	58
4.5.6	Identifying structured data types . . . . .	58
4.6	Learning points . . . . .	60
4.7	End of topic test . . . . .	61

---

**Prerequisites**

From your studies at National 5 you should already know:

- programming languages use a variety of simple data types including string, integer, real and boolean;
- a collection of values of the same data type can be stored using an array.

**Learning objective**

By the end of this topic you will be able to:

- explain the difference between a simple and a structured data type;
- understand the connection between data types and how computers store numbers and text;
- describe, use and give examples of structured data types:
  - Arrays
  - Parallel Arrays
  - Arrays of records

## 4.1 Revision

### Quiz: Revision

Go online



**Q1:** An integer is:

- a) a number greater than zero
  - b) a negative or positive number including zero with no decimal point
  - c) a negative or positive number including zero with a decimal point
  - d) a single digit number
- .....

**Q2:** A real number is:

- a) a number greater than zero
  - b) a negative or positive number including zero with no decimal point
  - c) a negative or positive number including zero with a decimal point
  - d) a single digit number
- .....

**Q3:** A Boolean is:

- a) a value which can be either true or false
  - b) a very large number
  - c) a variable which can only have two possible values
  - d) a complex data type
- .....

**Q4:** An array is:

- a) a collection of values
- b) a set of variables of the same type
- c) a structured data type storing values of the same type
- d) a list of values

## 4.2 Data types and pseudocode

### Learning objective

By the end of this topic you should be able to:

- understand that all programming languages store and manipulate data.

All programming languages manipulate data. In this topic we are going to look at how general purpose imperative programming languages store and manipulate data.

Throughout this unit we will be using SQA standard reference language to describe data types and control structures. This means that you should be able to convert the examples into whatever programming language your school or college is using.

Data types can be divided into two categories: **simple** and **structured** types.

The simple data types are:

- INTEGER
- REAL
- CHARACTER
- BOOLEAN

The structured data types are:

- ARRAY
- STRING
- RECORD

### 4.3 Simple data types

#### Learning objective

By the end of this topic you should be able to:

- understand the connection between simple data types and how computers store numbers and text.

We will be using the following simple data types:

- An INTEGER is a numerical value which has no decimal point. An INTEGER can be positive or negative including zero;
- A REAL is a numerical value which includes a decimal point;
- A CHARACTER is a single character from the keyboard or other input device;
- A BOOLEAN can have two values only: **true** or **false**.

These data types correspond to the various ways which computers store information at machine code level.

- INTEGERS are stored using **two's complement** notation;
- REAL numbers are stored using **floating point notation** which uses an exponent and a mantissa;
- CHARACTERS are stored as ASCII codes or if more than 128 are needed then they are stored using Unicode;
- A BOOLEAN can be stored using a single bit which is on or off.

Storing values using floating point notation is more memory and processor intensive than **two's complement**, and since there is always a trade-off between accuracy and range when storing values using floating point notation, it makes sense to store integer values as INTEGER rather than REAL.

All programming languages will have a have a maximum limit (positive and negative) for storing integers, and this limit is determined by the number of bits allocated to storing them.

If 32 bits were being used to store integers, then the range of possible values would be  $-2^{31}$  to  $2^{31} - 1$ .

If 64 bits were being used to store integers, then the range of possible values would be  $-2^{63}$  to  $2^{63} - 1$ . For this reason very large numbers are stored as REAL rather than INTEGER.

In maths and science, scientific notation is used to represent very large decimal numbers anyway. This is similar to the system of floating point used by computers.

**Activity: Simple data types (5 min)**

Go online



**Q5:** Decide what simple data type you would use to store the following values. Choose from:

- INTEGER
- REAL
- CHARACTER
- BOOLEAN

No.	Value	Simple data type
1	304	
2	45.78	
3	@	
4	-4	
5	5989.4	
6	-56.3	
7	!	
8	true	

## 4.4 Identifying simple data types

**Learning objective**

By the end of this topic you should be able to:

- identify the different simple data types used by your chosen programming language.

**Practical task: Simple data types (30 min)**

Look in the manuals, on the Internet or in the help documents for two of the programming languages in use in your school. List the simple data types available in your chosen languages.



## 4.5 Structured data types

### Learning objective

By the end of this section you will be able to:

- describe, use and give examples of structured data types:
  - Arrays
  - Parallel Arrays
  - Arrays of records

Most software applications require large amounts of data to be stored. If every single item of data had to be given a unique name then not only would this be very inconvenient, but accessing and manipulating these separate variables would be very complex.

For instance a set of numeric values could be stored by creating a set of variables:

```
1 SET number1 TO 23
2 SET number2 TO 16
3 SET number3 TO 3
4 SET number4 TO 77
5 SET number5 TO 23
```

They could be printed using this set of commands:

```
1 SEND number1 TO DISPLAY
2 SEND number2 TO DISPLAY
3 SEND number3 TO DISPLAY
4 SEND number4 TO DISPLAY
5 SEND number5 TO DISPLAY
```

This system becomes very cumbersome indeed if we are manipulating large quantities of data. If a number of items of the same type have to be stored, it makes sense to store them in a structure which can be referred to by a single identifier, and to be able to access these items using a control structure like a loop to process them sequentially. A structure like this is called an array.

Storing data such as numbers in a single structure makes printing and searching through the list much easier because the index can be used to identify each one in turn.

Array	23	16	3	77	23	1	11	9	45	39
Index	0	1	2	3	4	5	6	7	8	9

In this example we could set up an integer array storing 10 items to store these numbers:

```
VAR numbers [9]
```

So to print the contents of an array we could use the following code:

```
1 FOR counter FROM 0 TO 9 DO
2     SEND numbers[counter] TO DISPLAY
3 END FOR
```

We will be using the following structured data types:

ARRAY, STRING and RECORD.

### 4.5.1 Arrays

An ARRAY is an ordered sequence of simple data types, all of the same type.

[ 5, 9, 13] is an ARRAY storing three INTEGERS

["Fred","Sue","Jo","Anne"] is an ARRAY storing four STRINGS

Eg. [true, false, true, true] is an ARRAY storing four BOOLEANS

The arrays we will be using are one dimensional, ie. they are equivalent to an ordered list of items, identified by a single index. The index of an ARRAY or STRING starts at zero.

For example, the following two commands:

```
1 SET myNames TO ["Fred","Jim","Betty"]
2 SEND myNames[2] TO DISPLAY
```

Would create an ARRAY of three STRINGS and then print out the item: Betty

```
1 SET myNumbers TO [56, 7, 23, -12]
2 SEND myNumbers[0] TO DISPLAY
```

Would create an ARRAY of three INTEGERS and then print out the value: 56

### 4.5.2 Strings

A STRING is a special sort of ARRAY containing CHARACTERS.

Eg. "This is a message" is an example of a STRING.

STRINGS can be joined together or concatenated using the & symbol.

So the command:

```
SET newString TO "Hello " & "This is a message"
```

creates a new STRING with the value: "Hello This is a message".

Although strictly speaking a string is a complex data type, some programming languages treat a string as a simple data type rather than as an array of characters.

### 4.5.3 Parallel arrays

If we want to store information about the real world, we often need to store information of different types.

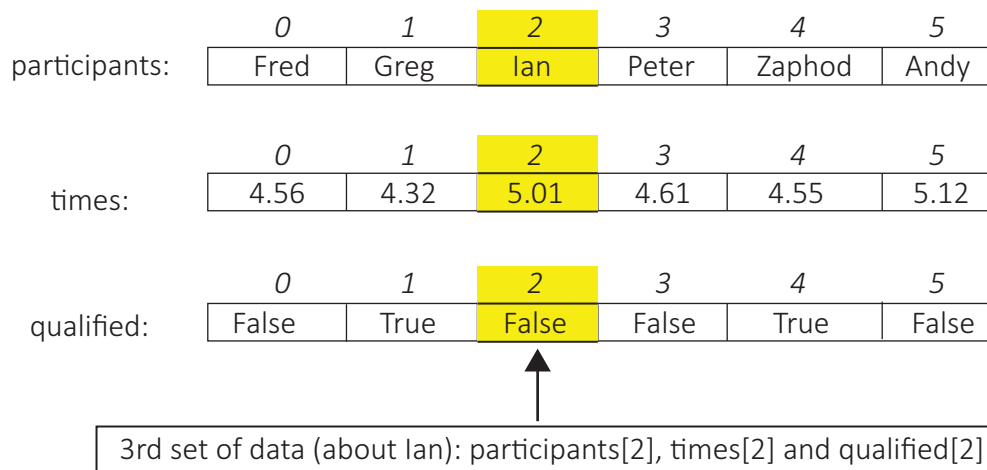
For example if we wanted to store the results of a race, we would need to store both the participants'

names as a `STRING` and their times as a `REAL` number. We might additionally want to store whether they qualified for the final or not as a `BOOLEAN` value.

We could store this information in 3 parallel arrays:

```
1 SET participants TO ["Fred", "Greg", "Ian", "Peter", "Zaphod", "Andy"]
2 SET times TO [4.56, 4.32, 5.01, 4.61, 4.55, 5.12]
3 SET qualified TO [false, true, false, false, true, false]
```

You can then use the arrays in 'parallel' —using the same value to **index** items from each array.



Here is an example of how you could output the information from the parallel arrays:

```
1 FOR counter FROM 0 TO 5 DO
2 SEND participant[counter] & "|" & times [counter] & "|" &
   qualified[counter] TO DISPLAY
3 END FOR
```

#### 4.5.4 Records

A `RECORD` can contain variables of different types, just as a record in a database can be made up of fields of different types. So a single record would be equivalent to a list of different types of item.

```
RECORD person IS {STRING forename, STRING surname, STRING address, INTEGER score}
```

For example, a record for a single individual could be:

```
SET person1 TO {Forename = "Fred", Surname = "Flintstone", Address = "Bedrock", score = 10}
```

A table in a database is equivalent to an `ARRAY` of `RECORDS`

A more intuitive way of storing the race information shown previously would be to store it as a set of records. In this case a single record would be the information about one participant —their name, time and qualifying status.

We firstly define the record structure in our language:

```
RECORD runner IS {STRING participant, REAL time, BOOLEAN qualified}
```

Then we can declare and **array of records**:

```
DECLARE race AS ARRAY OF runner INITIALLY []
```

Lastly, we can assign the relevant data to each item in the race array:

```

1 SET race[0] TO {participant = "Fred", time = 4.56, qualified = false}
2 SET race[1] TO {participant = "Greg", time = 4.32, qualified = true}
3 SET race[2] TO {participant = "Ian", time = 5.01, qualified = false}
4 SET race[3] TO {participant = "Peter", time = 4.61, qualified = false}
5 SET race[4] TO {participant = "Zaphod", time = 4.55, qualified = true}
6 SET race[5] TO {participant = "Andy", time = 5.12, qualified = false}

```

This loop will print the qualifiers:

```

1 FOR counter FROM 0 TO 5 DO
2
3 IF race[counter].qualified = true THEN
4     SEND race[counter].participant & " has qualified for the final"
5 END IF
6
7 END FOR

```

### Activity: Procedural language

Go online



**Q6:** Are the following Control Structures or Data Structures?

1. Arrays
2. Selection
3. Records
4. Iteration

### Practical task: Parallel arrays and records



Use your chosen programming language to create an array of records to store the race information and print out the qualifiers.

Another example could be if we wanted to store the characters in a kind of game. We would need to create a RECORD structure, then store the characters that used to be popular with "Dungeon and Dragon" enthusiasts as an ARRAY of RECORDS.

RECORD character IS{ string NAME, STRING weapon, INTEGER danger}

```

1 SET enemy[0] TO {name = "troll", weapon = " axe", danger = 3}
2 SET enemy[1] TO {name = "dwarf", weapon = " spell", danger = 3}
3 SET enemy[2] TO {name = "wizard", weapon = " staff", danger = 9}
4 SET enemy[3] TO {name = "ghost", weapon = " ectoplasm", danger = 2}

```

This loop will print out all the items in the array:

```
1 FOR counter FROM 0 TO 5 DO
2     SEND "Name: ", enemy[counter].name, "Weapon: ",
3         enemy[counter].weapon, "Danger level: " & enemy[counter].danger
4         TO DISPLAY
5 END FOR
```

Not all programming languages have a separate record structure, and use parallel arrays instead to store information which consists of a set of different data types.

### Activity: Data types 1 (5 min)

Go online



**Q7:** Decide what data type you would use to store the following values. Choose from

- INTEGER
- REAL
- CHARACTER
- BOOLEAN
- STRING

No.	Value	Data type
1	678	
2	Open Sesame!	
3	0	
4	-5.7	
5	4000	
6	TD5 7EG	
7	joe@companymail.com	

**Activity: Data types 2 (5 min)**

Go online

**Q8:** Decide what data type you would use to store the following values. Choose from

- INTEGER
- REAL
- CHARACTER
- BOOLEAN
- STRING

No.	Value	Data type
1	A UK telephone number	
2	The price of a pair of trainers	
3	Whether a character in a game has found a weapon or not	
4	The colour of a sprite	
5	The counter in a loop	
6	A URL	
7	A key-press	

**Activity: Structured data types (5 min)**

Go online

**Q9:** Decide what structured data type you would use to store the following. Choose from

- ARRAY of INTEGER
- ARRAY of REAL
- ARRAY of CHARACTER
- ARRAY of BOOLEAN
- ARRAY of STRING

No.	Value	Data type
1	A list of names	
2	A set of test scores out of 50	
3	The characters in a sentence	
4	The average temperatures during last month	
5	The last five Google searches you made	
6	Whether or not a class of pupils have passed an exam	

**Activity: Multiple data types (5 min)**

Go online



**Q10:** Decide what data types would be needed to create the following records, choose from:

- STRING, STRING, INTEGER
- STRING, INTEGER, BOOLEAN
- STRING, STRING, STRING,

No.	Records	Data types
1	Name, address and Scottish Candidate Number (SCN) for a list of pupils.	
2	Pupil ID, test score and pass/fail for a class	
3	Weapon name, ammunition type and damage value in a First Person Shooter game	

**Quiz: Pseudocode (15 min)**

Go online



Use your chosen programming language to work out what the result would be from the following pseudocode examples.

**Q11:** SET myNames TO ["Fred","Jim","Betty", "Justin","Greg"]  
SEND myNames[4] TO DISPLAY

- a) Fred
- b) Jim
- c) Betty
- d) Justin
- e) Greg

.....

**Q12:** SET myNames TO ["Fred","Jim","Betty", "Justin","Greg"]  
SEND myNames[1] TO DISPLAY

- a) Fred
- b) Jim
- c) Betty
- d) Justin
- e) Greg

.....

**Q13:** SET myVals TO [5, 12, 21, 35]  
SEND myVals[1] + myVals[3] TO DISPLAY

- a) 5
- b) 12
- c) 21
- d) 26
- e) 33
- f) 35
- g) 47
- h) 56

.....

**Q14:** SET mySentence TO "Hello World"  
SEND mySentence[6] TO DISPLAY

- a) H
- b) e
- c) l
- d) W
- e) o

#### 4.5.5 Handling records

##### Practical task: Handling records (60 min)



Find out the syntax your programming language uses to:

- Define a record structure
- Create an array containing 3 or more records
- Print out the contents of the array and the record.

#### 4.5.6 Identifying structured data types

##### Practical task: Structured data types (30 min)



Look in the manuals, on the Internet or in the help documents for two of the programming languages in use in your school. List the structured data types available in your chosen languages:



**Quiz: Identifying structured data types (5 min)**

Go online



Use your chosen programming language to work out what the result would be from the following pseudocode examples:

**Q15:**

```
1 RECORD monster IS {STRING name, BOOLEAN exists}
2 SET myMonster TO {name = "hydra", exists = false}
3 SEND myMonster.name to DISPLAY
```

- a) hydra
- b) exists
- c) myMonster
- d) false
- e) monster

.....

**Q16:**

```
1 TYPE monster IS RECORD {STRING name, BOOLEAN exists}
2 SET yourMonster TO {name = "crocodile", exists = true}
3 IF yourMonster.exists THEN send "Run!" TO DISPLAY
```

- a) crocodile
- b) Run!
- c) yourMonster
- d) monster
- e) exists

## 4.6 Learning points

### Summary

- All programming languages work with data, and that data can be held in a variety of ways depending on what type of data it is.
- Data types can be divided into two sorts: **simple** and **structured**.
- Simple data types are: INTEGER, REAL, CHARACTER and BOOLEAN.
- Simple data types correspond to the various ways which computers store information at machine code level: two's complement notation, floating point notation, ASCII code and as a single bit: 0 or 1.
- Structured data types are ARRAY and STRING (an ARRAY of CHARACTERS) and RECORD.
- Arrays, strings and records use an index to identify their contents. Indexes start at zero.
- You can declare an array of a record type that you have created.

## 4.7 End of topic test

### End of topic 4 test

[Go online](#)

**Q17:** From the data types listed above which would you use to store the following:

- INTEGER
  - REAL
  - CHARACTER
  - BOOLEAN
  - STRING
  - ARRAY of INTEGER
  - ARRAY of REAL
  - ARRAY of CHARACTER
  - ARRAY of BOOLEAN
  - ARRAY of STRING
  - RECORD
- a) The average of 5 INTEGERS?
- b) The visibility of a sprite in a game?
- c) The room descriptions in an adventure game.
- d) Time spent per day in seconds on the Internet over a month.
- e) Stock levels of products in a supermarket.
- f) The last 20 key-presses made while editing a document.
- g) A list of Email addresses.
- h) Whether a set of emails has been read or not.
- i) A set of room descriptions and contents in an adventure game.
- j) A set of pupil names and test scores.

.....

**Q18:**

Using SQA reference language, or a programming language you are familiar with, declare a record type for the following situation:

Anesha is writing a program to keep track of current smartphone specifications. She would like store:

- the make (e.g. 'Sumsang')
- model (e.g. 'Milky Way 12')
- storage capacity in GiB (e.g. 64)
- Fingerprint Scanner (e.g. TRUE)



---

## Topic 5

# Implementation: Algorithm specification

---

### Contents

5.1 Revision . . . . .	65
5.2 Standard algorithms . . . . .	66
5.3 Input validation . . . . .	67
5.4 Finding the minimum or the maximum value in an array . . . . .	70
5.5 Counting Occurrences . . . . .	72
5.6 Linear search . . . . .	74
5.7 Learning points . . . . .	78
5.8 End of topic test . . . . .	78

---

**Prerequisites**

From your studies at National 5 you should already know:

- an algorithm is a detailed sequence of steps which, when followed, will accomplish a task;
- an array is a data structure that stores a range of values of the same type in a single indexed structure;
- pseudocode is a method of describing a computer program in an informal English-like language;
- a fixed loop repeats program code a set number of times and a conditional loop repeats program code until a condition is met.
- Input Validation, Traversing an array and Running totals are examples of standard algorithms.

**Learning objective**

By the end of this topic you will be able to:

- recognise appropriate use of the following standard algorithms:
  - input validation;
  - find minimum/maximum;
  - count occurrences;
  - linear search.
- describe these standard algorithms in pseudocode and implement them in a high level programming language.

## 5.1 Revision

### Quiz: Revision

[Go online](#)

**Q1:** What complex data structure would you use to store a set of 20 student names?

- a) 20 STRING variables
- b) Array of CHARACTER
- c) ARRAY of STRING
- d) A file of STRING

.....

**Q2:** An index is?

- a) An ARRAY of INTEGER
- b) A list of numbers
- c) The position in an ARRAY
- d) A list of INTEGERS

.....

**Q3:** Which one of these is best used to describe the algorithm when designing software?

- a) Source code
- b) Pseudocode
- c) Binary code
- d) Machine code

.....

**Q4:** What kind of loop is this pseudocode an example of?

```
1 FOR counter FROM 0 TO 9 DO
2   SEND "Hello world" TO DISPLAY
3 END FOR
```

- a) A fixed loop
- b) A conditional loop
- c) A repeated loop
- d) A indexed loop

.....

**Q5:** What is the name of this standard algorithm?

```
1 For index FROM 0 to 5 DO
2 SET total TO total + scores[index]
3 END FOR
```

- a) Input Validation
- b) Traversing an array
- c) Running total
- d) Binary sort

## 5.2 Standard algorithms

### Learning objective

By the end of this section you will be able to:

- recognise appropriate use of the following standard algorithms:
  - input validation;
  - find minimum/maximum;
  - count occurrences;
  - linear search.

There are certain **algorithms** that appear in program after program. These are called **standard algorithms**.

The first one we are going to look at is **input validation**. You will already have implemented this at National 5. This is the task of making sure that the data input by the user is acceptable e.g. in a suitable format and within the upper and lower limits of the data required by the software, so that the program is both robust and reliable.

The other three algorithms we are going to examine all operate on lists of values: finding the maximum or minimum value, searching for a value, and counting the occurrences of a value. A common data structure used to store a list in a program is an **array**.



## 5.3 Input validation

### Learning objective

By the end of this section you will be able to:

- describe the Input validation algorithm in pseudocode and implement it in a high level programming language.

Input validation can be achieved in a number of ways: you can restrict the data the user can input by only presenting them with a limited set of possibilities such as a drop box or pull down menu, or you can accept their input but reject any which does not meet the restrictions imposed by the software and ask them to input it again. This second approach is the one we are going to study.

In this algorithm we are going to assume that the input from the user is going to come from the keyboard, and that the user will be asked for a value repeatedly until they provide one which meets the requirements of the program.

This means that we have to use a conditional loop to check to see whether the data is valid or not.

This algorithm uses a WHILE ... DO ... END WHILE conditional loop.

The user inputs the value then the **conditional loop** checks in case it is an invalid entry and asks for the value again.

```
1  PROCEDURE inputValidation()  
2  
3      RECEIVE userInput FROM (INTEGER) KEYBOARD  
4  
5      WHILE userInput < lowerLimit OR userInput > upperLimit DO  
6  
7          SEND "Input must be between "& lowerLimit & " and " & upperLimit  
            TO DISPLAY  
8          RECEIVE userInput FROM (INTEGER) KEYBOARD  
9  
10     END WHILE  
11  
12 END PROCEDURE
```

We could use a REPEAT ... UNTIL loop to do the same job.

The user inputs the value then the IF ... THEN ... END IF control structure checks to see if it is an invalid entry. Note that this algorithm is less efficient than the previous one because the input is being checked twice.

```

1  PROCEDURE inputValidation()
2
3      REPEAT
4
5          RECEIVE userInput FROM (INTEGER) KEYBOARD
6
7          IF userInput < lowerLimit OR userInput > upperLimit THEN
8              SEND "Input must be between " & lowerLimit & " and "
9              & upperLimit TO DISPLAY
10         END IF
11
12     UNTIL userInput >= lowerLimit AND userInput <= upperLimit
13
14 END PROCEDURE

```

### Practical task: Algorithms 1 (10 min)



Implement one of the `inputValidation` algorithms in your chosen programming language to ask the user for a number between 1 and 100.

Input validation for strings follows the same pattern:

```

1  PROCEDURE inputValidation()
2
3      RECEIVE userInput FROM (STRING) KEYBOARD
4
5      WHILE userInput ≠ ["Y"] AND userInput ≠ ["N"] DO
6
7          SEND "Input must be Y or N " TO DISPLAY
8          RECEIVE userInput FROM (STRING) KEYBOARD
9
10     END WHILE
11
12 END PROCEDURE

```

### Practical task: Algorithms 2 (10 min)



Write an input routine in your programming language which only accepts Y or N, but accepts them in upper or lower case.

Sometimes you may wish to limit the length of an input string.

This version of the input validation algorithm uses the length function to check the length of the `userInput` string against the `lengthLimit` value.

```
1 PROCEDURE inputValidation()  
2  
3 RECEIVE userInput FROM (STRING) KEYBOARD  
4  
5 WHILE (length(userInput) > lengthLimit) DO  
6  
7     SEND "Input must be less than " & lengthLimit & " Characters" TO  
8     DISPLAY  
9     RECEIVE userInput FROM (STRING) KEYBOARD  
10  
11 END WHILE  
12 END PROCEDURE
```

### Using a Boolean flag

In this version of the algorithm the **boolean** variable `validinput` is initialised to `false`, and the conditional loop only terminates when it has been set to `true`. This version of the algorithm is useful if you want to check a number of different conditions in the input string.

```
1 PROCEDURE inputValidation()  
2  
3 SET validInput TO false  
4  
5 REPEAT  
6  
7     RECEIVE userInput FROM (STRING) KEYBOARD  
8  
9     IF (length(userInput) < lengthLimit) THEN  
10        SET validInput TO TRUE  
11    ELSE  
12        SEND "Input must be less than " & lengthLimit & " characters" TO  
13        DISPLAY  
14    END IF  
15 UNTIL validInput = true  
16  
17 END PROCEDURE
```

### Practical task: Algorithms 3 (30 min)



You have been asked to write an input validation routine in your programming language to input a telephone number which can only contain the characters 0,1,2,3,4,5,6,7,8,9 and must be exactly 12 characters long.

(Note: if your programming language does not store a string as an array but stores them as a simple data type, you will have to use a string function to check each digit in turn.)

## 5.4 Finding the minimum or the maximum value in an array

### Learning objective

By the end of this section you will be able to:

- describe the Find Max and Min algorithm in pseudocode and implement it in a high level programming language.

To understand this algorithm, you have to remember that the processor can only do one thing at a time. The contents of the array to be examined can only be looked at individually. The *Finding the Maximum* algorithm uses a variable which will store the largest value in the array and at the beginning of the algorithm and makes it equal to the first item in the array. The rest of the array is then checked through one by one using an **fixed loop** comparing the contents with this variable and updating its value whenever a larger item is discovered.

### Activity: Find the maximum value in an array

Go online



Imagine a row of cards numbered from 0 to 9, each has a numeric value shown in the following table:

ID	Value
0	42
1	13
2	56
3	20
4	34
5	74
6	29
7	105
8	149
9	64

<b>Maximum value</b>

Each time you select a card, the value of that card is displayed as the maximum value IF it is greater than the value before. For example, card 4, followed by card 5 would display the value "74".

**Q6:** If you check the value of cards 0 through to 4 what would be the maximum value at this stage?

.....

**Q7:** If you check the value of cards 0 through to 5 what would be the maximum value at this stage?

.....

**Q8:** If you check the value of cards 0 through to 7 what would be the maximum value at this stage?

.....

**Q9:** If you select card 5, then 0, then 6, what value would be the maximum value?

.....

**Q10:** If you check the value of cards 0 through to 9 what would be the maximum value at this stage?

### Practical task: Finding the maximum



Assuming we have an array of 10 values: numbers[9] OF INTEGER

This algorithm sets maximumValue to the first item in the array then compares it to every item in the rest of the array. If one of these items is higher than the maximum, then it becomes the new maximum.

```

1  PROCEDURE  findMax()
2
3  SET maximumValue TO numbers[0]
4  FOR counter FROM 1 TO 9 DO
5    IF maximumValue < numbers[counter] THEN
6      SET maximumValue TO numbers[counter]
7    END IF
8  END FOR
9  SEND "The largest value was "& maximumValue TO DISPLAY
10
11 END PROCEDURE

```

Implement this algorithm in your programming language to find the maximum value in an array of 10 numbers.

The Finding the Minimum algorithm is very similar. Make the variable which will store the smallest value equal to the first item in the array then check through the rest of the array comparing each value in turn, swapping it if a smaller one is found.

```

1  PROCEDURE  findMin()
2
3  SET minimumValue TO numbers[0]
4  FOR counter FROM 1 TO 9 DO
5    IF minimumValue > numbers[counter] THEN
6      SET minimumValue TO numbers[counter]
7    END IF
8  END FOR
9  SEND "The smallest value was " & minimumValue TO DISPLAY
10
11 END PROCEDURE

```

If you want to know where in the array the value was found (ie. The index of the maximum or minimum value), as before you would use a fixed loop with a counter. The counter is then used to set the value of the foundAt variable whenever the minimumValue variable is updated.

```
1 PROCEDURE findMin()
2
3 SET foundAt TO 0
4 SET minimumValue TO numbers[0]
5
6 FOR index FROM 1 TO 9 DO
7   IF minimumValue > numbers[index] THEN
8     SET minimumValue TO numbers[index]
9     SET foundAt TO index
10  END IF
11 END FOR
12 SEND "The smallest value was "& minimumValue & " at position "
13     & foundAt & " in the list" TO DISPLAY
14
15 END PROCEDURE
```

### Practical task: Find winner (20 min)



Implement the findMin algorithm in your programming language to find the name of the winner of a race when the results are stored in two parallel arrays: names[9] and times[9].

## 5.5 Counting Occurrences

### Learning objective

By the end of this section you will be able to:

- describe the Counting Occurrences algorithm in pseudocode and implement it in a high level programming language.

The Counting Occurrences algorithm also uses an fixed loop to process an array. A variable that stores the number of times a particular value occurs is set to zero at the beginning of the procedure. It is then incremented every time the search value is identified in the array.

**Activity: Counting Occurrences**

Go online



Imagine a row of cards numbered from 0 to 9, each has a value shown in the following table:

ID	Value
0	A
1	B
2	C
3	D
4	B
5	A
6	A
7	B
8	E
9	C

Item to find	Occurrences

Each card has a value between A and E. Only one card can be revealed at a time in sequential order from 0 to 9.

The **Item to find** box is where you enter the value you wish to find.

What would be the output in the **Occurrences** box if the item to be found were:

**Q11:** A?

.....

**Q12:** B?

.....

**Q13:** C?

.....

**Q14:** D?

.....

**Q15:** E?

**Practical task: Counting Occurrences (15 min)**

Assuming we have an array of 10 values: numbers[9] OF INTEGER

This algorithm sets numberFound to the first item in the array then compares it to every item in the rest of the array.

```

1  PROCEDURE countOccurrences()
2
3  RECEIVE itemToFind FROM (INTEGER) KEYBOARD
4
5  SET numberFound TO 0
6
7  FOR EACH number FROM numbers DO
8    IF number = itemToFind THEN
9      SET numberFound TO numberFound + 1
10   END IF
11  END FOR EACH
12
13  SEND "There were " & numberFound & "occurrences of " & itemToFind
14     &
15     " in the list" TO DISPLAY
16  END PROCEDURE

```

Implement this algorithm in your programming language to find the number of times the character "e" occurs in a string typed at the keyboard.

(Note: if your programming language does not store a string as an array but stores them as a simple data type, you will have to use a string function to check each character in turn.)

**Practical task: Counting Occurrences 2 (10 min)**

Adapt your program to ask for the character you wish to search for as well as the phrase to be searched. Use input validation to make sure that the user only inputs a single character.

## 5.6 Linear search

**Learning objective**

By the end of this section you will be able to:

- describe the Linear Search algorithm in pseudocode and implement it in a high level programming language.

The linear search algorithm is used to find an item in a list. In this algorithm, a conditional loop is used to compare each item to the search term; the loop terminates when the search term is found.



A boolean variable is set to false at the beginning, then set to true when the item is found. A counter is used to keep track of where the item was found and the algorithm stops when either the first occurrence of the item has been found and the boolean variable has been set to true or the end of the array has been reached.

**Activity: Linear Search**

Go online



Imagine a row of cards numbered from 0 to 9, each has a value shown in the following table:

Index	Value
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J

Item to find	Found	Found at

Each card has a value between A and J. Only one card can be revealed at a time in sequential order from 0 to 9.

The **Item to find** box is where you enter the value you wish to find.

The **Found** box value is either TRUE or FALSE with a initial value of FALSE.

The **Found at** box value is the index value when your item is found.

**Q16:** If the item to find is C, what would be the **Found at** value be?

.....

**Q17:** If the item to find is H, what would be the **Found at** value be?

.....

**Q18:** If the item to find is D, and the **Found at** value is 8, what would the **Found** value be?

.....

**Q19:** If the item to find is F, and the **Found at** value is 5, what would the **Found** value be?

Assuming we have an array of 10 values: numbers[9] OF INTEGER

This algorithm uses a boolean variable `found` which is initially set to false. It uses a counter starting at 0 to check if the item searched for is present at that index position in the array.

```

1  PROCEDURE linearSearch()
2
3  RECEIVE itemToFind FROM (INTEGER) KEYBOARD
4
5  SET found TO false
6  SET arraySize TO highestIndex
7  SET counter TO 0
8
9  REPEAT
10     SET found TO numbers[counter] = itemToFind
11     SET counter TO counter + 1
12 UNTIL counter > arraySize OR found = true THEN
13
14 IF found = true THEN
15     SEND itemToFind & " found at position" & (counter - 1) TO DISPLAY
16 ELSE
17     SEND "Item not found" TO DISPLAY
18 END IF
19
20 END PROCEDURE

```

Note: In this algorithm the Boolean variable `found` is used to check whether the item has been found or not. The line `SET found TO numbers[counter] = itemToFind` could have been implemented using an `IF... THEN` structure...

```

1
2 IF numbers[counter] = itemToFind THEN
3     SET found TO true
4 END IF

```

... but the first version is more concise.

### Practical task: Linear Search (15 min)



Implement the algorithm `linearSearch` in your programming language to search for a name in a string array `names[9]`.

### Practical task: Linear Search 1 (60 min)



Create a program in your programming language which does the following:

- Fills an array with random integers between 1 and 10.
- Prints the array contents to screen.
- Finds and displays the maximum and minimum values in the array.

- Asks the user for an integer between 1 and 10 using input validation and displays how many times it occurs in the array.
- Asks the user for an integer between 1 and 10 using input validation and displays where in the array that number first occurs and indicates when it is not present.

### Linear search using a fixed loop

One limitation of the linear search algorithm which uses a conditional loop is the fact that it will only return the position of the first occurrence of the search item. If we use a fixed loop, then we can report the position of every instance in the numbers array.

```
1 PROCEDURE linearSearch2()  
2  
3   RECEIVE itemToFind FROM (INTEGER) KEYBOARD  
4   DECLARE found AS BOOLEAN INITIALLY false  
5  
6   FOR counter FROM 0 TO highestIndex DO  
7     IF numbers[counter] = itemToFind THEN  
8       SEND itemToFind & " found at position " & counter TO DISPLAY  
9       SET found TO true  
10    END IF  
11  END FOR  
12  IF found = false THEN  
13    SEND "Item not found" TO DISPLAY  
14  END IF  
15  
16 END PROCEDURE
```

### Practical task: Linear Search 2 (15 min)



Implement the algorithm `linearSearch2` in the programming language of your choice.

## 5.7 Learning points

### Summary

- Input validation is used to ensure that software is robust by repeatedly asking the user for input data and rejecting invalid data until the data meets the restrictions imposed by the software;
- Finding the maximum or minimum, counting occurrences and linear search all operate on arrays;
- Finding the maximum or minimum sets an initial value to the first item in the array then compares it to the remaining items;
- Counting occurrences sets a total to zero at the beginning and increments it as items are found to match the search item;
- The linear search sets a boolean variable to false initially and uses a conditional loop to set it to true when the item is found. The loop terminates when the item is found or the end of the array is reached.

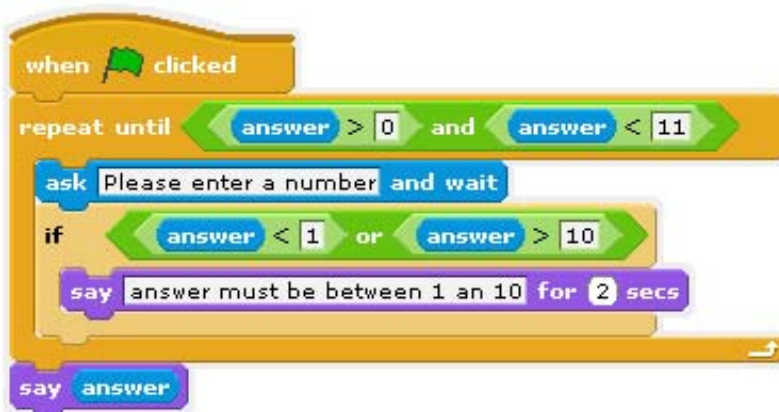
## 5.8 End of topic test

End of topic 5 test (10 min)

Go online



Q20:



This is an example of:

- Counting occurrences
- Input validation
- Linear search
- Finding the maximum

.....

**Q21:**

```
1 Private Sub Command1_Click()
2   Dim numbers(10) As Integer
3   Dim counter as Integer
4
5   For counter = 0 To 10
6     numbers(counter) = Int(Rnd * 10) + 1
7   Next counter
8
9   value = numbers(0)
10  For counter = 1 To 10
11    If Value < numbers(counter) Then value = numbers(counter)
12  Next counter
13
14  Print value
15 End Sub
```

This is an example of:

- a) Counting occurrences
  - b) Input validation
  - c) Linear search
  - d) Finding the maximum
- .....

**Q22:** The linear search algorithm uses:

- a) A fixed loop and a boolean variable
  - b) A conditional loop and a boolean variable
  - c) A maximum value and a boolean variable
  - d) A minimum value and a boolean variable
- .....

**Q23:** The counting occurrences algorithm uses:

- a) A fixed loop
  - b) A conditional loop and a boolean variable
  - c) A conditional loop
  - d) A minimum value and a boolean variable
- .....

**Q24:**

```
1 Private Sub Command1_Click()  
2   Dim numbers(10) As Integer  
3   Dim counter as Integer  
4  
5   For Counter = 0 To 10  
6     numbers(Counter) = Int(Rnd * 10) + 1  
7   Next Counter  
8  
9   total = 0  
10  value = 5  
11  For counter = 0 To 10  
12    If value = numbers(counter) Then  
13      total = total + 1  
14    End If  
15  Next counter  
16  Print total  
17 End Sub
```

This is an example of:

- a) Counting occurrences
- b) Input validation
- c) Linear search
- d) Finding the maximum

---

## Topic 6

# Implementation: Computational constructs

---

### Contents

6.1	Revision . . . . .	83
6.2	Introduction . . . . .	84
6.3	Variables and scope . . . . .	84
6.4	Pre-defined Functions . . . . .	85
6.4.1	Convert Real Numbers to Integer Numbers . . . . .	86
6.4.2	Modulus division . . . . .	86
6.4.3	Create substrings . . . . .	87
6.4.4	Convert characters to/from ASCII values . . . . .	88
6.5	Sub-programs . . . . .	89
6.6	User defined functions . . . . .	92
6.7	Parameters . . . . .	93
6.8	Sequential files . . . . .	97
6.9	CSV Files . . . . .	98
6.10	Learning points . . . . .	99
6.11	End of topic test . . . . .	100

---

**Prerequisites**

From your studies at National 5 you should already know how to:

- assign a value to a variable;
- use arithmetic and logical operators;
- use fixed and conditional loops;
- use simple and complex conditional statements;
- use pre-defined functions.

**Learning objective**

By the end of this topic you will be able to:

- understand how to use pre-defined functions to create substrings, convert characters to/from ASCII values, carry out modulus division and convert real numbers to integers;
- create and use sub-programs in your programming language;
- create your own user-defined functions;
- understand the concept of parameter passing and the difference between actual and formal parameters;
- use your chosen programming language to transfer data to and from text and CSV files.



## 6.1 Revision

### Quiz: Revision

[Go online](#)

**Q1:** What is this is an example of?

```
1 SET startPosition TO 1
```

- a) Assignment
  - b) Definition
  - c) Declaration
  - d) Optimisation
- .....

**Q2:** What is validItem an example of?

```
1 SET userInput TO validItem
```

- a) A function
  - b) A procedure
  - c) A user defined function
  - d) A definition
- .....

**Q3:** What is this is an example of?

```
1 IF age <=21 THEN checkId
```

- a) A complex conditional
  - b) A simple conditional
  - c) A conditional loop
  - d) An unconditional
- .....

**Q4:** What is this is an example of?

```
1 IF inputNumber >= 1 AND inputNumber <= 10 THEN
2   SEND "Number OK" TO DISPLAY
3 ELSE
4   SEND "Invalid entry" TO DISPLAY
5 END IF
```

- a) A simple conditional
  - b) A conditional loop
  - c) A complex conditional
  - d) An unconditional
- .....

**Q5:** When the program segment in question 4 is running, what would be the result if the value of `inputNumber` was 1?

.....

**Q6:** When the program segment in question 4 is running, what would be the result if the value of `inputNumber` was 11?

.....

**Q7:** What is this an example of?

```
1 FOR counter FROM 0 TO 9 DO
2   SET numbers[counter] TO Rand(100)
3 END FOR
```

- a) A fixed loop
- b) A conditional loop
- c) A conditional statement
- d) An unconditional loop

## 6.2 Introduction

A computational construct is a system of data representation and control structures used to solve problems using a computer through a programming language. What we are doing with any computer program is **storing and manipulating information**. Computational constructs are the features of a high level language which have been designed to make this task easier.

Although there are only three control structures: **sequence**, **selection** and **iteration**, to perform all programming tasks, code can be made more understandable if these control structures can be combined to make more powerful computational constructs.

This unit will look at a variety of constructs which are used in modern programming languages, and how they make the task of writing solutions to problems in a high level programming language easier.

## 6.3 Variables and scope

### Learning objective

By the end of this section you will be able to:

- understand the difference between the scope of a global variable and a local variable.

When a variable is created in a program this is called **variable declaration**. When a variable is declared, most languages allow it to be declared as being of a particular type and structure,

depending on the kind of data it is required to hold. Once a variable has been declared, it can be given a value. The value it has can then be used or changed (varied) during the running of the program.

Modern programming environments enable the programmer to create sub-programs within their main program. These sub-programs will correspond to the tasks which have been identified in the top down development process when the initial problem has been broken down into smaller sub-problems. Making these sub-programs as self-contained as possible is a good idea because if they contain variables which can change a value elsewhere in the code, it is often difficult to predict what the effects of this will be. This also improves the **modularity** of the code, so that the sub-programs can be tested independently, and also improves the portability of the code allowing the sub-programs to be used elsewhere without alteration.

A sub-program can only be self-contained if the variables declared and used within it are **local variables**. A local variable is one which only has a value within the sub-program where it is being used. This is often referred to as the **scope** of a variable. In most programming environments, the default for a variable is to be local to the sub-programs where they have been declared. Since a local variable only has a value inside its own sub-program, the variable name can be used again elsewhere without the danger of having an unexpected effect elsewhere.

A **global variable** has a wider scope - it exists and can be altered throughout the entire program. This means that if its value is changed inside a sub-procedure, that value will remain changed and will affect its value wherever in the program it is used, possibly unintentionally. A sub-procedure which uses a global variable will not be self contained, and is not going to be able to be used in a different context where that global variable does not exist. For these reasons, global variables are best avoided whenever possible.

## 6.4 Pre-defined Functions

### Learning objective

By the end of this section you will be able to:

- understand how to use pre-defined functions to:
  - convert real numbers to integers;
  - carry out modulus division;
  - create substrings;
  - convert characters to/from ASCII values;

As you may recall from National 5, programming languages come with a range of pre-defined functions built into them.

A function is a block of code that —when executed —will return a value to the piece of code that it was called from.

Here is an example of a function you will have used at National 5:

```
1 DECLARE myNumber AS REAL INITIALLY 3.4
2 DECLARE myRounded AS INTEGER INITIALLY 0
3 SET myRounded TO round(myNumber,0)
```

This would result in the variable `myRounded` storing "3"

### 6.4.1 Convert Real Numbers to Integer Numbers

Most languages contain a function to convert a real number into an integer. Here is an example using the `int()` function:

```
1 SET value TO 3.4
2 SET newValue TO int(value)
3 SEND newValue TO DISPLAY
```

In this case, the result will be "3". Different languages will generate an answer in different ways; some will simply discard anything after the decimal point while others will round the number. You should check with your tutor (or try it out yourself) to see what your language does!

### 6.4.2 Modulus division

Modulus division is a type of division where two integer numbers are divided and the remainder is returned.

```
1 SET result
2 TO 13 MOD 3
```

Result would be 1.

N.B. Languages usually treat modulus division more like an arithmetic operator than a function, but the result is the same. In Python for example:

```
1 Result
2 = 13 % 3
```

Result would be 1.

**Modulus division activity (20 min)**

Below is an algorithm that uses modulus division to convert a number of minutes entered by a user into hours and minutes.

Implement this in the programming language you are using in class.

```

1 DECLARE minutes AS INTEGER INITIALLY 0
2 DECLARE hours AS INTEGER INITIALLY 0
3 DECLARE remainingMinutes AS INTEGER INITIALLY 0
4
5 GET minutes FROM (INTEGER)KEYBOARD
6
7 SET hours TO int(minutes / 60)
8 SET remainingMinutes TO minutes MOD 60
9
10 SEND minutes & " minutes is " & hours & "hours and " &
11 remainingMinutes & "minutes" TO DISPLAY

```

**6.4.3 Create substrings**

In languages where strings are simple data structures, examples of common string functions are:

- left(string, n ) returns the first n characters of **string**
- right(string, n) returns the last n characters of **string**
- mid(string, r, n) returns n characters of **string** starting at r
- length(string) returns the number of characters in a **string** variable

```

1 SET myString TO "elephant"
2 Set newString TO left(myString, 3)
3 SEND newString TO DISPLAY

```

Result would be *ele*

```

1 SET myString TO "elephant"
2 Set newString TO right(myString, 3)
3 SEND newString TO DISPLAY

```

Result would be *ant*

```

1 SET myString TO "elephant"
2 Set newString TO mid(myString, 3, 5)
3 SEND newString TO DISPLAY

```

Result would be *phant*

```

1 SET myString TO "elephant"
2 SEND length(myString) TO DISPLAY

```

Result would be 8

#### 6.4.4 Convert characters to/from ASCII values

Functions also exist in most languages to convert individual characters to their ASCII character code and back again.

- `asc(char)` returns the ASCII code, e.g. `asc("A")` returns 65
- `chr(char)` returns the letter from an ASCII code, e.g. `chr(33)` returns "!"

This can be useful if you want to carry out 'calculations' on the characters, for example during encryption. (or, if you add/subtract 32 you can shift case!).

#### Convert characters to/from ASCII values activity



Below is an algorithm that uses the above functions to create a simple 'Caesar Cypher' encoder.

Implement this in the programming language you are using in class.

```

1 DECLARE shift AS INTEGER INITIALLY 1
2 DECLARE asciiValue AS INTEGER INITIALLY 0
3 GET character FROM (STRING)KEYBOARD
4 SET asciiValue TO asc(character)
5 #check it's not 'Z'!
6 IF asciiValue <= 89 THEN
7     SET asciiValue TO asciiValue + shift
8 ELSE
9     SET asciiValue TO 65
10 END IF
11 SEND chr(asciiValue) TO DISPLAY

```

## 6.5 Sub-programs

### Learning objective

By the end of this section you will be able to:

- create and use sub-programs in your programming language.

Sub-programs are named blocks of code which can be run from within another part of the program. When a sub-program is used like this we say that it is "called". Because they can be called from any part of the program they can be used over again if needed. For instance, an input validation sub-program may be called several times asking for different inputs. This makes your program easier to understand and makes writing code more efficient.

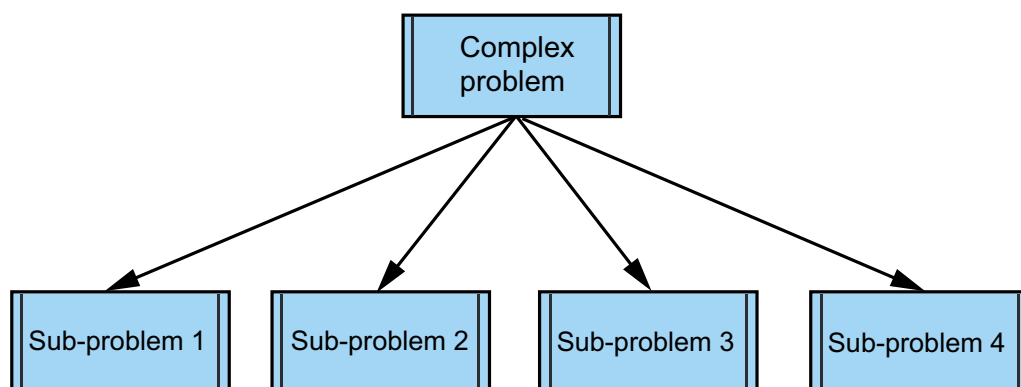
Sub-programs are often called **procedures** (which execute a set of commands), or **functions** (which return a value). In the case of object-oriented programming languages (such as Java) they are called methods. Breaking your program down into sub-programs is a good idea because it makes your code **modular**, readable and therefore more maintainable.

The structure of a program should follow the top down analysis which was originally used to break the problem down into smaller sub-tasks.

*<main program>*

```
1 SubProgram1()
2   SubProgram2()
3   SubProgram3()
4   SubProgram4()
```

*<end of main program> <SubProgram1> <SubProgram2> <SubProgram3> <SubProgram4>*



For example the exercise in the 'Algorithm specification' topic might have the following structure:

Note:

- In this example we are using a global variable: numbers[9] of INTEGER.
- Rand() is a function which returns a random number, so Rand(100) returns a random number between 1 and 100.
- GetValidInteger() is a user defined function which returns an integer value between 1 and 100 when the function is called.

```
1 PROCEDURE mainProgram
2   fillArray()
3   printArray()
4   findMinimum()
5   findMaximum()
6   countOccurrences()
7   linearSearch()
8 END PROCEDURE
9
10 PROCEDURE fillArray()
11   FOR counter FROM 0 TO 9 DO
12     SET numbers[counter] TO Rand(100)
13   END FOR
14 END PROCEDURE
15
16 PROCEDURE printArray()
17   FOR counter FROM 0 TO 9 DO
18     SEND numbers[counter] TO DISPLAY
19   END FOR
20 END PROCEDURE
```

```
1 PROCEDURE findMaximum()
2   SET maximumValue TO numbers[0]
3   FOR counter FROM 1 TO 9 DO
4     IF maximumValue < numbers[counter] THEN
5       SET maximumValue TO numbers[counter]
6     END IF
7   END FOR
8   SEND "The largest value was " & maximumValue TO DISPLAY
9 END PROCEDURE
10
11 PROCEDURE findMinimum()
12   SET minimumValue TO numbers[0]
13   FOR counter FROM 1 TO 9 DO
14     IF minimumValue > numbers[counter] THEN
15       SET minimumValue TO numbers[counter]
16     END IF
17   END FOR
18   SEND "The smallest value was " & minimumValue TO DISPLAY
19 END PROCEDURE
20
21 PROCEDURE countOccurrences()
22   SET itemToFind TO GetValidInteger()
23   SET numberFound TO 0
24   FOR EACH number FROM numbers DO
25     IF number = itemToFind THEN
26       SET numberFound TO numberFound + 1
27     END IF
28   END FOREACH
29   SEND "There were " & numberFound & "occurrences of " & itemToFind
30     & " in the list" TO DISPLAY
31 END PROCEDURE
32
```



```
33 FUNCTION getValidInteger() RETURNS INTEGER
34   RECEIVE userInput FROM (INTEGER) KEYBOARD
35   WHILE userInput < 1 OR userInput > 100 DO
36     SEND "Input must be between 1 and 100 " TO DISPLAY
37     RECEIVE userInput FROM (INTEGER) KEYBOARD
38   END WHILE
39   RETURN userInput
40 END FUNCTION
```

```
1  PROCEDURE linearSearch()
2   SET itemToFind TO getValidInteger()
3   SET found TO false
4   SET arraySize TO highestIndex
5   SET counter TO 0
6
7   REPEAT
8     SET found TO numbers[counter] = itemToFind
9     SET counter TO counter + 1
10  UNTIL found OR counter > arraySize
11
12  IF found THEN
13    SEND itemToFind &" found at position" & counter - 1 TO DISPLAY
14  ELSE
15    SEND "Item not found" TO DISPLAY
16  END IF
17 END PROCEDURE
```

### Practical task: Linear search (60 min)



Edit your programming language solution to the exercise at the end of the 'Algorithm specification' topic to reflect the pseudocode structure above.

The exercise was to write a program in your programming language which does the following:

- Fills an array with random integers between 1 and 10.
- Prints the array contents to screen.
- Finds and displays the maximum and minimum values in the array.
- Asks the user for an integer between 1 and 10 using input validation and displays how many times it occurs in the array.
- Asks the user for an integer between 1 and 10 using input validation and displays where in the array that number first occurs and indicates when it is not present.

### Methods

A method in an object-oriented language is a function that is defined inside a class.

In our example we would create the class `mainProgram` with the `getValidInteger` function and other methods defined within it.

Object-oriented programming languages often use the syntax:

```
1 object.functionName()
```

So our `getValidInteger` function would be called like this

```
1 MainProgram.getValidInteger()
```

## 6.6 User defined functions

### Learning objective

By the end of this section you will be able to:

- create your own user-defined functions.

Where a sub-program performs a sequence of commands (and relies on parameters for both input and output of data, which you will learn about in section 6.7), functions are a little bit different: a function uses **parameters** for input of data but will only send back ('return') one value. This characteristic allows a function to be used as part of a larger expression, as the function will be evaluated and replaced by a value.

Because of this characteristic, functions need to be declared with a data type in most languages, e.g.:

```
1 FUNCTION addTwoNumbers() RETURNS INTEGER
2     <code for function>
3     RETURN answer
4 END FUNCTION
```

The function can then be used as if it were just any other part of an expression. Let's suppose the `addTwoNumbers()` function asks the user for two numbers then returns the sum of those numbers. We could use it in any other calculation like so:

```
1 DECLARE mean AS REAL INITIALLY 0.0
2 SET mean TO addTwoNumbers() / 2
```

This will call the function; the function will return an integer when complete and then the division will take place.

In the previous section we used our own user-defined function `getValidInteger()` to return a value between 1 and 100.

```
1 FUNCTION getValidInteger() RETURNS INTEGER
2
3 RECEIVE userInput FROM (INTEGER) KEYBOARD
4
5 WHILE userInput < 1 OR userInput > 100 DO
6     SEND "Input must be between 1 and 100 " TO DISPLAY
7     RECEIVE userInput FROM (INTEGER) KEYBOARD
8 END WHILE
9
10 RETURN userInput
11
12 END FUNCTION
```

### Practical task: User-defined functions (30 min)



Create your own user-defined functions to return the following:

```
1 FUNCTION newRandom() RETURNS INTEGER
```

which returns a random number between 50 and 100.

```
1 FUNCTION userName() RETURNS STRING
```

which returns a string with a maximum length of 10 characters.

## 6.7 Parameters

### Learning objective

By the end of this section you will be able to:

- understand the concept of parameter passing and the difference between actual and formal parameters.

A more flexible solution to the input validation problem in the previous section would be to use **parameters** to set the range of numbers we wanted to return from the `getValidInteger` function when we called it. This would mean that the function could be called with different values (or variables) depending on what range of number we wanted to restrict it to.

**Formal parameters** are the parameters within brackets in the declaration of a function or procedure. A function or procedure without formal parameters will still have a set of empty brackets after the name in its declaration. Many programming languages will require these formal parameters to be given a data type in the declaration as well as a name. In this example the `getValidInteger` function is declared with two formal parameters, `lowerLimit` and `upperLimit`.

Sub-programs are defined with **formal** parameters and called with **actual** parameters.

```

1 FUNCTION getValidInteger(lowerLimit, upperLimit) RETURNS INTEGER
2
3   RECEIVE userInput FROM (INTEGER) KEYBOARD
4
5   WHILE userInput < lowerLimit OR userInput > upperLimit DO
6     SEND "Input must be between "& lowerLimit &" and "& upperLimit TO
7     DISPLAY
8     RECEIVE userInput FROM (INTEGER) KEYBOARD
9   END WHILE
10  RETURN userInput
11
12 END FUNCTION

```

This function can now be used in any program to return a valid number within the range provided. We could call this function with actual parameters, 1 and 50 to return a number between 1 and 50:

```

1 numberToUse = getValidInteger(1,50)

```

or we could call it with the actual parameters 1 and inputRange - a variable which has a value assigned elsewhere in the program:

```

1 RECEIVE inputRange FROM (INTEGER) KEYBOARD
2 SET numberToUse TO getValidInteger(1, inputRange)

```

This call would return a value between 1 and whatever was stored/held in the variable inputRange.

### Practical task: Parameters 1 (15 min)



Edit the code for the previous exercise to use the user-defined `getValidInteger` function with parameters.

### Practical task: Parameters 2 (15 min)



Create your own user-defined functions to return the following:

```

1 FUNCTION double(value) RETURNS INTEGER

```

which returns a number which is double the number passed into it.

```

1 FUNCTION userName(forename, yearOfBirth) RETURNS STRING

```

which returns a string which concatenates the forename and birth year passed into it.

Procedures can have formal parameters as well. For example we could declare the `printArray` sub-procedure with the formal integer array, `numbers`. Again many programming languages will require that formal parameters are given a data type as well as a name.

```
1 PROCEDURE printArray(numbers)
2
3   FOR counter FROM 0 TO 9 DO
4     SEND numbers[counter] TO DISPLAY
5   END FOR
6
7 END PROCEDURE
```

This would mean that the `numbers` array could be declared as a local variable within the main program, and then passed as an actual parameter to each one of the sub-programs.

The code for the `countOccurrences` sub-program would now be:

```
1 PROCEDURE countOccurrences(numbers)
2
3   SET itemToFind TO getValidInteger( 1, 100)
4   SET numberFound TO 0
5
6   FOREACH number FROM numbers DO
7     IF number = itemToFind THEN
8       SET numberFound TO numberFound + 1
9     END IF
10  END FOREACH
11
12  SEND "There were " & numberFound & "occurrences of " & itemToFind
13      & " in the list" TO DISPLAY
14
15 END PROCEDURE
```

If we now rewrite our original program using parameter passing throughout, we get:

```
1 PROCEDURE main()
2
3   fillArray(numbers)
4   printArray(numbers)
5   findMinimum(numbers)
6   findMaximum(numbers)
7   countOccurrences(numbers)
8   linearSearch(numbers)
9
10 END PROCEDURE
11
12 PROCEDURE fillArray(numbers)
13
14   FOR counter FROM 0 TO 9 DO
15     SET numbers[counter] TO Rand(100)
16   END FOR
17
18 END PROCEDURE
```

```
19
20 PROCEDURE printArray(numbers)
21
22   FOR counter FROM 0 TO 9 DO
23     SEND numbers[counter] TO DISPLAY
24   END FOR
25
26 END PROCEDURE
27
28 FUNCTION getValidInteger(lowerLimit, upperLimit) RETURNS INTEGER
29
30   RECEIVE userInput FROM (INTEGER) KEYBOARD
31   WHILE userInput < lowerLimit OR userInput > upperLimit DO
32     SEND "Input must be between "& lowerLimit &" and "& upperLimit TO
33       DISPLAY
34     RECEIVE userInput FROM (INTEGER) KEYBOARD
35   END WHILE
36   RETURN userInput
37 END FUNCTION
```

```
1 PROCEDURE countOccurrences(numbers)
2
3   SET itemToFind TO getValidInteger(1,100)
4   SET numberFound TO 0
5   FOR EACH number FROM numbers DO
6     IF number = itemToFind THEN
7       SET numberFound TO numberFound + 1
8     END IF
9   END FOREACH
10  SEND "There were " & numberFound & "occurrences of " & itemToFind
11    & " in the list" TO DISPLAY
12
13 END PROCEDURE
14
15
16 PROCEDURE linearSearch(numbers)
17
18   SET itemToFind TO getValidInteger(1,100)
19   SET found TO false
20   SET arraySize TO highestIndex
21   SET counter TO 0
22   REPEAT
23     SET found TO numbers[counter] = itemToFind
24     SET counter TO counter + 1
25   UNTIL found OR counter > arraySize
26   IF found THEN
27     SEND itemToFind & " found at position" & (counter-1) TO DISPLAY
28   ELSE
29     SEND "Item not found" TO DISPLAY
30   END IF
31
32 END PROCEDURE
```

Now all the variables used in the program are local to the main sub-program and so there are no global variables. The sub-programs are modular and each one could be used again with a different array as their actual parameter - in elsewhere in this or another program. In the same way that the `getValidInteger` function could be used again with different actual parameters to return a value within a different range.

### Practical task: Parameters 3 (30 min)



Create a procedure which takes two parallel arrays as parameters, a set of 10 names and a set of 10 scores and prints out the highest scoring name.

## 6.8 Sequential files

### Learning objective

By the end of this section you will be able to:

- use your chosen programming language to transfer data to and from sequential files.

As far as the computer is concerned, data can be input from a keyboard or a file and can be output to a display or file. If a file does not already exist, it may have to be created with a specific command, or your programming language may create it as part of the `OPEN` command.

```
1  PROCEDURE getData(numbers)
2    <open file "mydata.txt">
3    FOR counter FROM 0 TO 9 DO
4      RECEIVE numbers[counter] FROM (INTEGER) "mydata.txt"
5    END FOR
6    <close file "mydata.txt">
7  END PROCEDURE
8
9  PROCEDURE saveData(numbers)
10   CREATE "newdata.txt"
11   <open file "newdata.txt">
12   FOR counter FROM 0 TO 9 DO
13     SEND numbers[counter] TO "newdata.txt"
14   END FOR
15   <close file "newdata.txt">
16  END PROCEDURE
```

### Practical task: Sequential files



Adapt the parallel arrays exercise to read the scores array from a file. Use a text editor such as Notepad to create the file.

## 6.9 CSV Files

A Comma-Separated Value file, or CSV file for short, is a standard file format for exchanging organised data between programs such as databases and spreadsheets.

As implied by the name, they are simply text files with each data item (similar to a field in a database) separated by a comma value, and each row (record) separated by a line break.

Here is an example of a CSV file contents.

CSV files can be opened and read in the same way as a normal text file. In-built procedures (or methods in an object-oriented language) are available to automatically split the file by line and by comma, allowing you to quickly fill parallel arrays for an array of records with the contents of the CSV file.

Large amounts of data are available from official sources that can be used by researchers and academics. This data is often supplied in CSV files, allowing it to be imported to software packages or processed by a program that has been written specifically to analyse the data. Some examples include:

- <http://statistics.gov.scot>
- <https://www.opendata.nhs.scot/>
- <https://data.gov.uk/>

When you are happy with your ability to read CSV files into the programming language you are using, you may wish to explore some of these sources and try to write programs that apply the standard algorithms you have learned to the data sets.

### Practical Task: CSV Files



1	Bob	76
2	Mary	34
3	Fred	46
4	Sean	65
5	Kayleigh	40
6	Sarah	61
7	John	79
8	Rubina	38
9	Abby	71
10	Stephen	47
11	Chloe	63
12	Kasper	80
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		
28		
29		
30		
31		
32		
33		
34		
35		
36		
37		
38		
39		
40		
41		
42		
43		
44		
45		
46		
47		
48		
49		
50		

Using the programming language in your school or college, find out how to input this to a set of parallel arrays—or an array of records—and search the list for the highest score.



## 6.10 Learning points

### Summary

- A **computational construct** is a combination of control structures which can be used to make solving programming problems more intuitive.
- The scope of a variable describes where it can be accessed from.
- **Global** variables have scope throughout a program, **local** variables can only be accessed from within their own sub-procedure.
- If possible, global variables should be avoided and all variables in a program should be local.
- Breaking a problem down into smaller sub-problems enables **modular** code to be created where each sub-problem is coded as a separate procedure.
- A user-defined function is a sub-program which returns a value, and is defined as being of the data type corresponding to that value.
- A method in an object-oriented language is a function that is defined inside a class.
- Subprograms are defined with **formal** parameters and called with **actual** parameters.
- Most Languages have pre-defined functions that will allow you to create substrings from longer strings, and convert characters to/from ASCII to do manipulate them mathematically.
- Pre-defined functions also exist to carry out modular division and convert a real number to an integer number.
- Sequential files are treated in the same way as other input and output devices, but with specific commands for opening and closing.
- Comma Separated Value (CSV) files are a standard file format for exchanging structured information. Most languages have in-built procedures for reading their contents into arrays.

## 6.11 End of topic test

End of topic 6 test (10 min)

Go online



**Q8:** What is this code an example of?

```

1 SET items TO 0
2
3 FOR EACH number FROM numbers DO
4   IF number = item THEN
5     SET total TO total + 1
6   END IF
7 END FOR EACH

```

- a) Find the maximum
  - b) Find the minimum
  - c) Counting occurrences
  - d) Linear search
- .....

**Q9:** This function `getValidInteger()` returns what?

```

1 FUNCTION getValidInteger() RETURNS STRING
2
3 RECEIVE userInput FROM (STRING) KEYBOARD
4 WHILE length(userInput > 10 DO
5   SEND "Input must be less than 10 characters ") TO DISPLAY
6   RECEIVE userInput FROM (STRING) KEYBOARD
7 END WHILE
8 RETURN userInput
9
10 END FUNCTION

```

- a) Real value
  - b) Integer value
  - c) Boolean value
  - d) String value
- .....

**Q10:** This line of code is in a program to add the name "Fred" to an array of STRINGS:

```

1 addNameToList("Fred", names)

```

"Fred" and names are?

- a) Formal parameters
  - b) Actual parameters
  - c) Real parameters
  - d) Reference parameters
- .....

**Q11:** In this procedure definition, *name* and *listOfNames* are?

```
1 PROCEDURE addNameToList (name , listOfNames)
```

- a) Formal parameters
- b) Actual parameters
- c) Real parameters
- d) Reference parameters

.....

**Q12:** What string would display as a result of this code?

```
1 SET myString TO "ORANGES AND LEMONS"
2 Set newString TO left(9,myString)
3 SEND newString TO DISPLAY
```

- a) ORANGES A
- b) ND LEMONS
- c) ORANGE
- d) AND LEMON

.....

**Q13:** What length would display as a result of this code?

```
1 SET myString TO "ORANGES AND LEMONS"
2 SEND length(myString) TO DISPLAY
```

- a) 10
- b) 16
- c) 18
- d) 20

.....

**Q14:** A formal parameter whose value may be changed by the procedure where it is defined is?

- a) A reference parameter
- b) A value parameter
- c) An actual parameter
- d) A real parameter

.....

**Q15:** . A formal parameter whose value can NOT be changed by the procedure where it is defined is?

- a) A reference parameter
- b) A value parameter
- c) An actual parameter
- d) A real parameter

.....

**Q16:** The line of code SET answer TO 17 MOD 4 would result in answer containing which value?

- a) 17
- b) 4
- c) 1
- d) 0

---

## Topic 7

# Testing

---

### Contents

7.1	Revision . . . . .	105
7.2	Test plans . . . . .	105
7.3	Debugging . . . . .	107
7.4	Debugging tools . . . . .	108
7.4.1	Dry runs . . . . .	108
7.4.2	Trace tables . . . . .	108
7.4.3	Trace tools . . . . .	110
7.4.4	Breakpoints . . . . .	110
7.4.5	Watchpoints . . . . .	111
7.5	Learning points . . . . .	111
7.6	End of topic test . . . . .	112

---

### Prerequisites

From your studies at National 5 you should already know:

- why we should use normal, extreme and exceptional test data;
- that using internal commentary, meaningful identifiers, and indentation aids code readability.

### Learning objective

By the end of this topic you will be able to:

- construct a test plan;
- describe:
  - comprehensive testing;
  - systematic testing;
- explain the difference between syntax, execution and logic errors;
- understand how dry runs, trace tables, trace tools and breakpoints are used in the debugging process.

## 7.1 Revision

### Quiz: Revision

Go online



**Q1:** Which set of test data would be the best one to use to test an input routine asking for numbers between 1 and 100?

- a) 0, 1,10,50, 99,100, 101, X
- b) 1 5 20 40, 50, 60, 90 100
- c) 5,9,2,60,80 100, 101, A
- d) 0, 1,5, 46, 67, 84, 90, 93

.....

**Q2:** Which identifier would be the best one to use for an array of integers storing test scores?

- a) score
- b) scores
- c) s
- d) values

.....

**Q3:** Which lines of code in this example should be indented to make it more readable?

```

1 Line 1  INTEGER FUNCTION getvalidItem()
2 Line 2  RECEIVE userInput FROM (INTEGER) KEYBOARD
3 Line 3  WHILE userInput < 1  OR userInput > 100 DO
4 Line 4  SEND ["Input must be between 1 and 100 "] TO DISPLAY
5 Line 5  RECEIVE userInput FROM (INTEGER) KEYBOARD
6 Line 6  END WHILE
7 Line 7  RETURN userInput
8 Line 8  END FUNCTION
    
```

.....

**Q4:** What is internal commentary?

## 7.2 Test plans

### Learning objective

By the end of this section you will be able to:

- construct a test plan;
- describe comprehensive testing;
- describe systematic testing.

As we have seen in the Development Methodologies topic, testing can only demonstrate the presence of errors, it cannot demonstrate their absence. For this reason, testing should be both **systematic** and **comprehensive**.

**Systematic** testing is where tests are done in a way which is planned, and which can be documented as a result. **Comprehensive** testing is when every aspect of the software is tested.

A **test plan** is a set of test data which has been created in order to systematically and comprehensively test the software which the client has requested in order to ensure that it meets the original specification when delivered. Much of the test plan will be created during the design stage of the software development process, because by this stage it should be known what the inputs will be and outputs should be, and also what the user interface looks like. The test plan and its results are part of the documentation which will accompany the testing stage of the software development process when it is complete.

A test plan will include:

1. **The software specification against which the results of the tests will be evaluated.**

The **software specification** produced at the end of the analysis stage of the software development process should detail the functional requirements, scope and boundaries. It is a legally binding document which protects both client and developer. The design of the test plan must take this document into account.

2. **A schedule for the testing process.**

The testing schedule is necessary for the same reason as every other part of the software development process needs to be scheduled in order to deliver the project on time.

3. **Details of what is and what is not to be tested.**

**Exhaustive testing** - where every possible input and permutations of input to a program are tested - is not possible. Even a simple input validation routine could theoretically need to be tested with every possible valid number, and the possibilities run into millions once you have several different inputs which could be applied in any order. The tests selected should be ones which are practical within the time available. There will always be external circumstances which cannot be tested until the software is in the hands of the client or the user base. This is where **acceptance testing (beta testing)** is important.

4. **The test data and the expected results.**

A test plan will include **normal**, **extreme** and **exceptional** test data; the results expected from inputting this data to the program and whether the result passes or fails the test.

This is a set of test data for a sub-program which should allow the user to input a whole number between and including 1 and 100.

	Data	Expected Result	Actual Result	Pass / Fail
<b>Normal</b>	46, 62, 80	accept		
<b>Extreme</b>	1, 100	accept		
<b>Exceptional</b>	0, 101, -5, 15.6, A, %	reject		

*NB. It is important that values outside the input range, but on the boundary of acceptable data are included in the exceptional test data.*



### 5. Documentation of the testing process.

The testing process needs to be documented so that if problems are encountered at a later date, the test results can be checked and duplication of work avoided.

This kind of testing of a program by the developers is called **alpha testing**.

## 7.3 Debugging

### Learning objective

By the end of this section you will be able to:

- explain the difference between syntax, execution and logic errors.

**Debugging** is the process of finding and correcting errors in code.

Some errors in code will be discovered during the implementation stage. However some will only be identified at the testing stage which means that the implementation stage needs to be re-visited to correct them.

Errors likely to be spotted at the implementation stage are syntax errors and execution (runtime) errors. A **syntax error** is one which can be spotted by a translator: by a compiler when the source code is translated into machine code, or by an interpreter while the code is being entered by the programmer. A syntax error could be a misspelling of a keyword, or a mistake in the structure of a program like a missing END WHILE in a WHILE loop or a missing END IF in an IF condition.

An **execution error** is one which happens when the program is run, causing it to stop running (crash). Examples include division by zero or trying to access an array index that's beyond the range of that array. These types of error are not identified by the compiler or the interpreter, but appear when the program is run.

Logical errors, sometimes called **semantic errors**, are ones where the code is grammatically correct as far as the interpreter or compiler is concerned, but does not do what the programmer intended. These types of error may be spotted during the implementation stage, but may also be spotted during the testing stage.

### Quiz: Debugging

Go online



**Q5:** Which of these are **not** syntax errors?

- Missing semi colon
- Division by zero
- IF without END IF
- Out of memory
- WHILE without DO

.....

**Q6:** There is an error in this pseudocode. What kind of error is it?

```
1 FOR counter FROM 1 TO 20 DO
2     SET total TO 0
3     RECEIVE userInput FROM (INTEGER)KEYBOARD
4     SET total TO total + userInput
5 END FOR
6 SET average TO total / 20
7 SEND "The average of these numbers was " & average TO DISPLAY
```

## 7.4 Debugging tools

### Learning objective

By the end of this section you will be able to:

- understand how dry runs, trace tables, trace tools and breakpoints are used in the debugging process.

Debugging is made much easier if source code is well documented, and uses meaningful variable names and indentation. Modularity makes debugging easier since sub programs can be tested independently, especially if they are self contained and do not use global variables.

Syntax errors will be highlighted by the interpreter or compiler while code is being written or compiled, but logical errors can only be found by running a program and watching its operation. There are a number of techniques which can be used to monitor the values of variables at different points in the code execution to aid this process.

### 7.4.1 Dry runs

A dry run is simply a manual run-through the pseudocode or source code of the program, usually taking notes of the values of variables at various points in the process while doing so. In effect the person doing the dry run is taking the place of the computer in order to check that the code is doing what they expect it to do. Keeping track of the values of variables at different stages of the code execution is complicated so normally the tester would use a table, either on paper or on computer to help.

### 7.4.2 Trace tables

A trace table is similar to the table that would be used during a dry run, but is often used to test an algorithm for a specific sub program when the tester wants to check the result of a number of different values of a variable. A trace table and its results are an important element in the documentation of the testing process.

**Algorithm:**

```

1 SET total TO 0
2 FOR counter FROM 1 TO 5 DO
3     RECEIVE userInput FROM (INTEGER)KEYBOARD
4     SET total TO total + userInput
5 END FOR
6 SET average TO total / 5
7 SEND "The average of these numbers was "& average TO DISPLAY

```

**Trace table:**

Total	Counter	userInput	average
0	1	3	0
3	2	7	0
10	3	4	0
14	4	11	0
25	5	11	5

**Output:** The average of these numbers was 5.

**Algorithm:**

```

1 PROCEDURE linearSearch(numbers,)
2
3     SET itemToFind TO 10
4     SET found TO false
5     SET arraySize TO 4
6     SET counter TO 0
7
8     REPEAT
9     IF number[counter] = itemToFind THEN
10        SET found TO true
11    END IF
12        SET counter TO counter + 1
13    UNTIL found OR counter > arraySize
14
15    IF found THEN
16        SEND itemToFind &" found at position" & counter - 1 TO DISPLAY
17    ELSE
18        SEND "Item not found" TO DISPLAY
19    END IF
20
21 END PROCEDURE

```

**Trace table:**

itemToFind	found	arraysize	counter
10	false	4	0
10	false	4	1
10	false	4	2
10	true	4	3

**Output:** 10 was found at position 3 in the list.

**Note:** arrays are indexed from zero.



### Practical task: Trace tables (10 min)

Create a trace table for this algorithm:

```

1 SET numbers TO [3, 15, 4, 7, 8]
2
3 PROCEDURE findMaximum(numbers)
4
5     SET maximumValue TO numbers[0]
6     FOR counter FROM 1 TO 4 DO
7         IF maximumValue < numbers[counter] THEN
8             SET maximumValue TO numbers[counter]
9         END IF
10    END FOR
11    SEND "The largest value was "& maximumValue TO DISPLAY
12
13 END PROCEDURE

```

### 7.4.3 Trace tools

Some programming environments have **trace** facilities as a debugging feature. Tracing tools let the programmer see which lines of code are being executed and what variables are changing their value while the program is running.

A **watch** takes a variable and displays its value as the program progresses. The programmer steps through the code, one statement at a time, and the value of the variable being traced is displayed on the watch screen which can be set to stop when reaches a particular value.

Some trace tools also allow investigation of actual memory locations and, in particular, the contents of the **stack**.

Programs that contain a large number of procedures use the stack to store all their procedure calls during program execution. By examining such data, any errors occurring in the order of procedure or function calling from the main program can be checked and corrected.

### 7.4.4 Breakpoints

Some programming environments will enable the programmer to set a breakpoint.

Setting a breakpoint in a program sets a point in the source code where the program will stop execution, at which point the values of variables at this point can be examined.

Breakpoints can be set to stop execution at a particular point in code.

Once the program has stopped, the values of the variables in use can be examined, or written to a file for study later.

### 7.4.5 Watchpoints

Whereas a breakpoint is a specific place in a program where you want it to stop, a watchpoint is where you set a program to stop when a variable has a specific value or when a particular event such as a keypress, data entry or a menu selection has occurred. When using either a breakpoint or a watchpoint, the purpose is to track the flow of data or the changes in values of variables in order to debug the code.

## 7.5 Learning points

### Summary

- Testing can only demonstrate the presence of errors - it cannot demonstrate their absence.
- A test plan is a set of test data which should systematically and comprehensively test the software to ensure that it meets the original specification.
- A test plan will include normal, extreme and exceptional test data.
- A syntax error is where the "grammatical" rules of the language have been broken. It is normally detected spotted by a compiler or interpreter.
- A logic error is one where the code is grammatically correct but does not do what the programmer intended.
- An execution error is one which causes the program to stop (crash) when it is run.
- A dry run is a manual run through pseudocode or the source code of the program.
- A breakpoint is a set point in a program where it will stop execution so that the values of variables can be examined.
- A watchpoint is when a program is set to halt when a variable has reached a specific value.
- Test and their results should be fully documented so that if subsequent problems are discovered, they can be checked and duplication of work avoided.

## 7.6 End of topic test

End of topic 7 test (10 min)

Go online



**Q7:** Which of these are syntax errors?

- Missing semi colon
- Division by zero
- IF without END IF
- Out of memory
- WHILE without DO

.....

**Q8:** Which of these are execution errors?

- Missing semi colon
- Division by zero
- IF without END IF
- Out of memory
- WHILE without DO

.....

**Q9:** Fill in the missing values in this trace table:

minimumValue	counter	numbers[counter]
	1	
	2	
	3	
	4	

```

1 SET numbers TO [17, 15, 4, 7, 8]
2
3 PROCEDURE findMinimum(numbers)
4
5   SET minimumValue TO numbers[0]
6   FOR counter FROM 1 TO 5 DO
7     IF minimumValue > numbers[counter] THEN
8       SET minimumValue TO numbers[counter]
9     END IF
10  END FOR
11  SEND "The smallest value was "& minimumValue TO DISPLAY
12
13 END PROCEDURE

```

.....

**Q10:** A computer program is designed to accept input values between 0 and 99 as whole numbers. If the value 99 was entered this would be an example of?

- a) Exceptional data
- b) Normal data
- c) Invalid data
- d) Extreme data

.....

**Q11:** Alpha testing is carried out by end users of a program.

- a) True
- b) False

.....

**Q12:** Beta testing is carried out by end users of a program.

- a) True
- b) False

.....

**Q13:** Fill in the missing values in this trace table:

value	display	counter
1	no display	0
		1
		2
		3
		4
		5
		6

```

1 PROCEDURE sequence()
2   SET value TO 1
3   SET counter TO 0
4   REPEAT UNTIL counter > 5
5     SET value TO value + counter
6     SEND value TO DISPLAY
7     SET counter TO counter + 1
8   END REPEAT
9 END PROCEDURE
    
```

.....

**Q14:** Which of these source code characteristics do NOT help in debugging a program?

- a) Internal documentation
  - b) Modular code
  - c) Global variables
  - d) Meaningful variable names
- .....

**Q15:** Why is it important to document any tests performed during the software development process?

- a) To avoid duplication of tests if errors are discovered later on.
  - b) To let the client know that the software has been tested for errors.
  - c) To help ensure that the testing is both comprehensive and systematic.
  - d) To make sure that tests are not done unnecessarily.
- .....

**Q16:** Which of these is not a debugging tool provided with a software development environment?

- a) Breakpoint
- b) Watchpoint
- c) Trace
- d) Dry run



---

## Topic 8

# Evaluation

---

### Contents

8.1	Revision . . . . .	117
8.2	Software evaluation . . . . .	117
8.2.1	Fitness for purpose . . . . .	117
8.2.2	Efficiency . . . . .	118
8.2.3	Usability . . . . .	119
8.2.4	Maintainability . . . . .	120
8.2.5	Robustness . . . . .	120
8.3	Learning points . . . . .	121
8.4	End of topic test . . . . .	122

---

### Prerequisites

From your studies at National 5, you should already know that:

- software can be evaluated in terms of fitness for purpose, efficiency and robustness.
- source code can be evaluated for its readability. Using internal commentary, meaningful identifiers, indentation and white space all affect this.

### Learning objective

By the end of this topic you will be able to:

- evaluate software you or others have developed based on:
  - fitness for purpose;
  - efficient use of coding constructs;
  - usability;
  - maintainability;
  - robustness.

## 8.1 Revision

### Quiz: Revision

[Go online](#)

**Q1:** Which of the following improves the readability of code? You may choose more than 1 answer.

- a) Indentation
  - b) Internal commentary
  - c) Test data
  - d) Wireframes
- .....

**Q2:** Robustness means:

- a) How quickly a piece of software can carry out its actions.
- b) Whether users like the user interface.
- c) Whether a program has been tested.
- d) How well a program can cope with invalid data.

## 8.2 Software evaluation

This topic contains sub-topics:

- Fitness for purpose
- Efficiency
- Usability
- Maintainability
- Robustness

### 8.2.1 Fitness for purpose

Fitness for Purpose is making sure that the program meets all the requirements that were agreed in the Software Specification drawn up during the analysis of the problem, or meets the clients' needs if using an Agile methodology.

The program should be evaluated against the functional requirements, scope and boundaries to ensure that all agreed elements of the program have been implemented and function as expected by clients/users.

## 8.2.2 Efficiency

### Efficiency

Software is considered to be efficient if it avoids using resources unnecessarily. Resources may be processor time, RAM, hard disk space, or Internet bandwidth. There may be a trade-off between programmer time and efficiency. The increased processor speed and memory capacity of modern machines can encourage saving valuable programmer time but at the expense of creating less efficient software. This can be seen with newer versions of operating systems, which often perform more slowly than their predecessor on the same hardware.

### Coding Constructs

Careful consideration of how to implement a program is important to make it efficient. Some examples are:

- Choosing when to use fixed and conditional loops. For example, a linear search that should terminate when it finds a value should use a conditional loop—a fixed loop would continue to search the array even when the value is found.
- Re-using functions or Sub Procedures: This means the code is physically shorter and less space for declared variables or arrays needs to be reserved in RAM.
- Careful use of conditional statements: consider the following two examples of code:

#### Example 1

```
1 SET grade TO "F"
2 IF score >= 50 AND score < 60
   THEN
3   grade = "C"
4 END IF
5 IF score >= 60 AND score < 70
   THEN
6   grade = "B"
7 END IF
8 IF score >= 70 THEN
9   grade = "A"
10 END IF
11 SEND grade TO DISPLAY
```

#### Example 2

```
1 SET grade to "F"
2 IF score >= 50 THEN
3   grade="C"
4   IF score >= 60 THEN
5     grade="B"
6     IF score >= 70 THEN
7       grade="A"
8     END IF
9   END IF
10 END IF
11 SEND grade TO DISPLAY
```

In example 1, the program will have to evaluate all three expressions in the three IF statements, even if the first one is true and can therefore only result in a 'C'. The program is reliable in the sense that it provides the correct output but is inefficient.

Example 2 uses **nested if** statements, and the expressions are simpler (1 operator to evaluate instead of 3). The first IF statement will be evaluated regardless, but if the score is less than 50 there will be no further evaluation. Similarly, if the score is greater than 50 but less than 60 then only the first and second IF statement have been evaluated.

## Memory and Variable Types

You need to think carefully about what type of data is being stored and ensure you choose the best option. Here are typical storage requirements for variables declared in the programming language Visual Basic 2015:

Variable Type	Storage Requirements
Integer	32 bits
Short Integer	16 bits
String	16 bits per character, maximum 4GB! (2 billion characters).
Real Number —Single Precision	32 bits
Real Number —Double Precision	64 bits
Boolean	1 bit

If you were only wanting to represent small numbers, between -32,768 and +32,767 then it would make more efficient use of memory to use a short integer than a normal integer. This is a simplistic example, but large programs with multiple arrays, records and parameters need to be careful to make good use of memory.

### Efficiency activity (10 min)

Go online



Using the web, try to find out the storage requirements for common data types in a programming language you have used in class.

## 8.2.3 Usability

The user interface of a software product is the part which has most influence on the reaction of its users. A user interface should be:

- Customisable
- Appropriate
- Consistent
- Provide protection from error
- Accessible

A user interface should be **customisable** so that users can alter the way they use the software to their own preferences.

A user interface must be **appropriate** to the expertise of the user expected to be using the software. Ideally it will provide a number of different levels of interface depending on the expertise of the user. A word processor for instance will provide a number of different ways of performing the same function, menu options, shortcut keys, and toolbar icons.

A user interface should be **consistent** so that users find similar functions grouped together under menus, dialog boxes with commonly used options like OK and Cancel in the same place.

A user interface should **provide protection from error** so that critical events such as deleting data give sufficient warning to the user before they are completed.

A user interface should be **accessible** so that its design does not impede those users with disabilities. This might mean making the interface compatible with a text reader, providing customisable colour settings for a high contrast display or the provision of optional large size icons or toolbars.

### 8.2.4 Maintainability

Software is **maintainable** if it can be easily changed and adapted. This is why **readability** and **modularity** are so important in software design. The person maintaining it may not be the same person as the one who wrote it. Even the original author may find their code difficult to understand at a later date if it has not been written clearly. Modularity makes a program easier to maintain because the separate functions can be tested and changed without causing unexpected consequences with other parts of the program. It is also easier to locate (and therefore correct) errors in a modular program.

### 8.2.5 Robustness

Software is robust if it is able to cope with mistakes that users might make or unexpected conditions that might occur. These should not lead to wrong results or cause the program to hang. For examples an unexpected condition, could be something going wrong with a printer, (it jams, or it runs out of paper) a disc drive not being available for writing, because it simply isn't there, or the user entering a number when asked for a letter. Put simply, a robust program is one which should never crash.

How can you guard against incorrect input?

- Include error-checking code such as an input validation algorithm (like you learned at National 5) for anything typed in by the user.
- Use built-in functions to convert data types before processing. For example, if you are expecting an integer but it is possible for the user to enter a real number, use the `int()` function on the user's input.
- Design the user interface so that the user can select items from a drop-down list or menu rather than typing in values. (This is commonplace on many websites where a calendar is used to ensure a valid date is entered.)

You must also ensure your software is thoroughly tested using exceptional test data prior to release.

**Activity: Evaluation terminology**

Go online



**Q3:** Match each term to the correct description:

- Usability
- Maintainable
- Efficient
- Fit for purpose
- Robust

Term	Description
	Ability of a program to keep running even when external errors occur.
	The program's interface is clear and can be operated by the intended users.
	Whether the program wastes memory or processor time.
	Has the program been designed to easily altered by another programmer.
	Does the program fulfil all the requirements of the specification.

### 8.3 Learning points

#### Summary

- Evaluating Software on Fitness for Purpose requires checking against the software specification to ensure all agreed elements have been implemented and are functioning correctly.
- Efficient programs make sensible use of data constructs, data types and design algorithms that run with the least amount of code required to be executed.
- Loops and Conditional statements require careful consideration as they have a big impact on the amount of processing time required to run a program.
- Usability of a program can be improved by providing help readily, using a consistent and clear layout, allowing customisation and giving prompts or feedback when errors occur.
- Program code can be made more maintainable by both improving the readability of code and through the use of modular programming (making use of procedures and functions).
- Programs should be evaluated for robustness to ensure they can cope with incorrect input and give feedback to the user without crashing.

## 8.4 End of topic test

End of topic 8 test (10 min)

Go online



**Q4:** Checking that a program contains all the features requested by the developer is called:

- a) Fitness for purpose
  - b) Usability
  - c) Robustness
  - d) Efficiency
- .....

**Q5:** Making sure a program makes good use of computer resources such as RAM and processor time is called:

- a) Fitness for purpose
  - b) Usability
  - c) Robustness
  - d) Efficiency
- .....

**Q6:** Users perceive that a new piece of software is difficult to find features on the user interface. Which type of evaluation should have identified this?

- a) Fitness for purpose
  - b) Usability
  - c) Robustness
  - d) Efficiency
- .....

**Q7:** Evaluating program code to ensure it is readable and modular is called?

- a) Fitness for purpose
- b) Usability
- c) Robustness
- d) Maintainability



## **Topic 9**

# **End of unit 1 test**

---

## End of unit 1 test

Go online



**Q1:** Which of the following characteristics are true of Agile software development?

1. Responsiveness to changed circumstances.
  2. Increased costs.
  3. Reduced time spent on analysis.
  4. Reduced development time.
- .....

**Q2:** Which of the following is true about traditional iterative development methodologies?

- a) The client is heavily involved at all stages.
  - b) They work best on small-scale projects like smartphone apps.
  - c) They are suited to large-scale team-based software development.
  - d) They offer the same advantages of agile development.
- .....

**Q3:** Which of the following is **not** a consideration during analysis?

- a) Purpose
  - b) Scope
  - c) Functional requirements
  - d) High Level Language choice
- .....

**Q4:** Which of these is **not** a design notation?

- a) Structure diagram
  - b) Data flow diagram
  - c) Source code
  - d) Pseudocode
- .....

**Q5:** Top Down Design is:

- a) creating pseudocode from the structure diagram and data flow diagram.
  - b) breaking a large and complex problem into smaller, more manageable sub-problems.
  - c) writing source code.
  - d) creating a wireframe interface design.
- .....

**Q6:** Which design notation would you use to design a user interface?

- a) Wireframe
- b) Structure diagram
- c) Pseudocode
- d) Data flow diagram

.....  
**Q7:** Which data structure would be best suited to store a set of test marks for a class of 20 pupils?

- a) 20 STRING variables
- b) A STRING array of 20
- c) An INTEGER array of 20
- d) A REAL array of 20

.....  
**Q8:** What type of variable should be used to identify the index of an array?

- a) STRING
- b) REAL
- c) BOOLEAN
- d) INTEGER

.....  
**Q9:** The competitor's names and times in a race are stored in two arrays. Which data types will be used for the arrays?

- a) STRING array and INTEGER array
- b) REAL array and INTEGER array
- c) STRING array and REAL array
- d) STRING array and BOOLEAN array

.....  
**Q10:** Parallel arrays are:

- a) two arrays containing linked data with the same index values.
- b) two arrays with different index values containing different data.
- c) arrays with identical information content.
- d) arrays which are part of the same procedure.

.....  
**Q11:** Runners in a race have the following information stored about them: name, nationality, previous personal best time, and lane number. What is the best way of storing this data?

- a) A set of 4 variables for each runner
  - b) 4 separate arrays
  - c) A single record structure
  - d) A single variable for each runner
- .....

**Q12:** Race time data is used to calculate the number of qualifiers who have performed better than the average race time. Which of these algorithms will be used?

- a) Input validation
  - b) Counting occurrences
  - c) Linear search
  - d) Finding the maximum
- .....

**Q13:** Why does the linear search algorithm need a Boolean variable?

- a) To count the number of items found
  - b) To terminate the loop when the item is found.
  - c) To store where the item is found.
  - d) To terminate the loop when the end of the array is reached.
- .....

**Q14:** Which standard algorithm is being used in this pseudocode segment?

```
1 RECEIVE item FROM (INTEGER) KEYBOARD
2 SET total TO 0
3 FOR EACH number FROM numbers DO
4     IF number = item THEN
5         SET total TO total + 1
6     END IF
7 END FOREACH
8 SEND total TO DISPLAY
```

- a) Counting occurrences
  - b) Input validation
  - c) Finding the Maximum
  - d) Linear search
- .....

**Q15:** Which standard algorithm would you use to find the name of the winner?

- a) Counting occurrences
  - b) Finding the Minimum
  - c) Finding the Maximum
  - d) Linear search
- .....

**Q16:** Which standard algorithm would you use to find out how many qualifiers there were?

- a) Counting occurrences
  - b) Finding the Minimum
  - c) Finding the Maximum
  - d) Linear search
- .....

**Q17:** Which standard algorithm would you use to find the time of a specific contestant?

- a) Counting occurrences
- b) Finding the Minimum
- c) Finding the Maximum
- d) Linear search

.....

**Q18:** What would be the output from this pseudocode example?

```
1 SET myVals TO [1, 12, 13, 35]
2 SEND myVals[0] + myVals[3] TO DISPLAY
```

- a) 1
- b) 14
- c) 47
- d) 36

.....

**Q19:** In this procedure definition, name and times are:

```
1 PROCEDURE findWinner(name, times)
```

- a) formal parameters
- b) actual parameters
- c) real parameters
- d) reference parameters

.....

**Q20:** In this function call the parameters 1 and 100 are:

```
1 Userinput = validNumber( 1, 100)
```

- a) formal parameters
- b) actual parameters
- c) real parameters
- d) reference parameters

.....

**Q21:** Which of these statements are **true**?

- 1. A function returns a value.
- 2. A function can be called with formal parameters.
- 3. A function can be user-defined.
- 4. A function is the same a procedure.

.....

**Q22:** Which of these are syntax errors?

1. Missing semi colon
2. Division by zero
3. IF without END IF
4. Overflow error
5. Out of memory
6. WHILE without DO

.....

**Q23:** Which of the following pre-defined functions would return an integer value?

- Int()
- Asc()
- Mod()
- Left()

.....

**Q24:** In the following algorithm to write a score to a file, what command is missing?

1. CREATE "highscore.txt"
2. SEND highest\_score TO "highscore.txt"
3. \_\_\_\_\_ "highscore.txt"

- a) CLOSE
- b) OPEN
- c) APPEND
- d) FINISH

.....

**Q25:** Which of these is not a debugging technique?

- a) Dry-run
- b) Break-run
- c) Trace Table
- d) Watchpoint

.....

**Q26:**

A programming environment allows the programmer to turn on the following display:

Counter(INT)	Score(INT)
1	14
2	23
3	45
4	45

Which debugging technique is this an example of?

- a) Dry-run
- b) Break-run
- c) Trace Table
- d) Watchpoint

.....

**Q27:** A program is being evaluated to see if it correctly deals with errors when the user enters incorrect data. This is referred to as:

- a) Robustness
- b) Reliability
- c) Fitness for Purpose
- d) Efficiency

.....

**Q28:** When evaluating a program in terms of its Fitness for Purpose, what documentation should be used?

- a) Internal commentary from the programmers.
- b) Functional Requirements from the software specification.
- c) Feedback from beta testers.
- d) Pseudocode from the designers.

## Glossary

### Actual parameter

the parameters which are used when a procedure or function is called.

### Algorithm

a detailed sequence of steps which, when followed, will accomplish a task.

### Boolean

a value which can only be true or false.

### Conditional loop

a control construct which allows a block of code to be repeated until a condition is met, often depending on user input.

### Execution error

an error which only manifests itself when a program is run rather than when its source code is translated.

### Fixed loop

a loop which repeats a set number of times.

### Formal parameter

the parameters in the definition of a procedure or function.

### Function

a function is a sub-program which returns a value.

### Global variable

a global variable is one which has scope throughout the entire program where it occurs.

### Increment

incrementing a variable means to increase its value by a fixed amount (usually 1).

### Local variable

a local variable is one which only has scope within the sub-procedure it has been declared in.

### Modularity

a program is said to be modular if it is composed of sub-programs which can be tested independently.

### Module library

a self contained pre-written and pre-tested blocks of code which can be re-used in other programs.

### Procedure

a sub-program which can be called from within it's main program.

### Scope

the scope of a variable is the range of sub-programs where it has a value.



**Semantic error**

a logical error in a program when the code is grammatically correct, but does not do what it is intended to do.

**Stack**

a dynamic data structure much used by software applications and the computer for storing temporary data. Data can only be accessed via the top of the stack.

**Syntax error**

an error in a program where the code is grammatically incorrect and cannot be translated into machine code.

**Value parameter**

a value parameter in a function or procedure definition is one which does not change. It is a copy of the value being passed into the sub-program. The original outside the sub-program does not change.

**Variable declaration**

when a variable is defined for the first time giving it a name and a data type.

**Watch**

where the programmer identifies a variable whose value can be displayed in a separate window while a program is running in order to help with de-bugging.

## Answers to questions and activities

### Topic 1: Development methodologies

#### Quiz: Revision (page 3)

**Q1:** b) Implementation

**Q2:** c) Agreeing the functional requirements with the users.

#### Quiz: Analysis (page 6)

**Q3:** The analysis stage is important because unless the initial problem description is clearly stated and the software specification agreed upon, then subsequent stages will suffer from delays and difficulties due to the need to re analyse the task and rewrite the software specification.

**Q4:** The software specification describes what the software to be created must be able to do.

#### Activity: Testing (page 10)

**Q5:** 2) Extreme, 4) Normal, and 5) Exceptional

**Q6:**

Normal: 2, 4, 5

Extreme: 1, 7

Exceptional: 0, 8, @, 67

#### Quiz: Testing (page 11)

**Q7:**

b) Testing is done by the programmers responsible for the application.

e) Testing may be done on parts of the application.

**Q8:** a) The testing is performed by the clients.

#### Activity: Maintenance (page 14)

**Q9:** 2) Corrective, 4) Perfective, and 5) Adaptive

**Activity: Waterfall model (page 14)****Q10:**

Analysis:	Looking at the problem and collecting information
Design:	Creating a structure diagram and pseudocode
Implementation:	Writing the source code
Testing:	Trying to find ways in which the program will fail
Documentation:	Creating a user guide and technical guide
Evaluation:	Checking to see how well the software meets its specification
Maintenance:	Fixing problems and adapting the software to new circumstances

**End of topic 1 test (page 19)**

**Q11:** d) Tutorial

**Q12:** a) Programmers

**Q13:** b) Analysis, Design, Implementation, Testing, Documentation, Evaluation, Maintenance

**Q14:** d) Reduced time spent on analysis

**Q15:** a) High level of involvement of the client/customer.

**Topic 2: Analysis****Quiz: Revision (page 23)**

**Q1:** b) A detailed list of what the finished program must do.

**Q2:** a) input —process —output

**Exercise: Identifying inputs, processes and outputs (page 24)****Example answer**

Inputs	Processes	Outputs
<ul style="list-style-type: none"> <li>• Height in Meters</li> <li>• Weight in KG</li> </ul>	<ul style="list-style-type: none"> <li>• • Weight/height<sup>2</sup></li> <li>• Selection of 1 of 4 categories:               <ul style="list-style-type: none"> <li>◦ Underweight</li> <li>◦ Ideal</li> <li>◦ Overweight</li> <li>◦ Obese</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• "You are underweight"</li> <li>• "You are ideal weight"</li> <li>• "You are overweight"</li> <li>• "Warning: you are obese"</li> </ul>

**End of topic 2 test (page 29)**

**Q3:** b) What data a program will not accept.

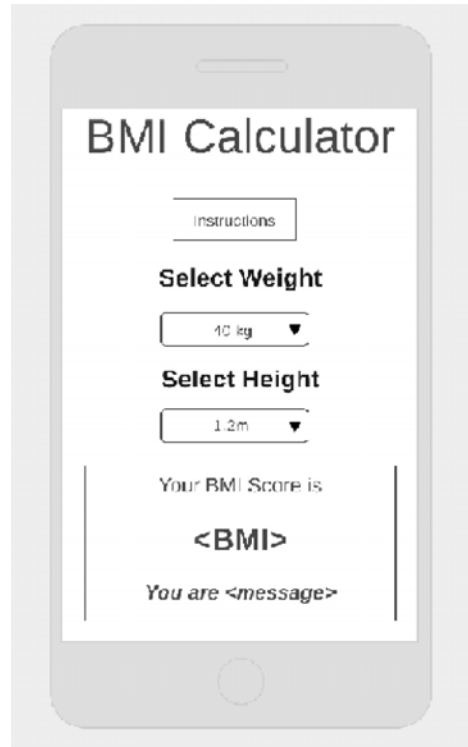
**Q4:** a) True

**Q5:** b) Checking the fingerprint data against a database of authorised users.

**Q6:** a) A set of precise statements covering each input, process and output.

**Topic 3: Design****Quiz: Revision (page 33)****Q1:** a) Pseudocode**Q2:** c) 10**Q3:** b) Structure Diagram**Activity: Pseudocode (page 40)****Example Answer:**

```
1 SET counter TO 0
2 REPEAT
3 SEND "Dear " & names[counter] & "You scored " & score[counter] & "% in
  the test" TO <printer>
4 UNTIL names[counter] = ""
5 SEND "The highest score in the class was " & highest_score & "% " TO
  <printer>
```

**Exercise: Create a wireframe (page 42)****Example answer****End of topic 3 test (page 44)**

**Q4:** b) The modules in a structure chart will become modules in the finished program.

**Q5:** c) Pseudocode

**Q6:** a) Wireframe

**Q7:** c) Pseudocode

**Q8:** b) False

**Topic 4: Implementation: Data types and structures****Quiz: Revision (page 47)**

**Q1:** b) a negative or positive number including zero with no decimal point

**Q2:** c) a negative or positive number including zero with a decimal point

**Q3:** a) a value which can be either true or false





**Q4:** c) a structured data type storing values of the same type

**Activity: Simple data types (page 50)**

**Q5:**

No.	Value	Simple data type
1	304	Integer
2	45.78	Real
3	@	Character
4	-4	Integer
5	5989.4	Real
6	-56.3	Real
7	!	Character
8	true	Boolean

**Practical task: Simple data types (page 50)****Example answer**

Data Type	Visual Basic 6	Snap! (formerly called BYOB)
Integer	Integer	
Real	Single, Double	
Character	String	
Boolean	Boolean	

**Activity: Procedural language (page 54)****Q6:**

Control Structures	Data Structures
Selection	Arrays
Iteration	Records

**Activity: Data types 1 (page 55)****Q7:**

No.	Value	Data type
1	678	INTEGER
2	Open Sesame!	STRING
3	0	CHARACTER
4	-5.7	REAL
5	4000	INTEGER
6	TD5 7EG	STRING
7	joe@companymail.com	STRING

**Activity: Data types 2 (page 56)****Q8:**

No.	Value	Data type
1	A UK telephone number	STRING *
2	The price of a pair of trainers	REAL
3	Whether a character in a game has found a weapon or not	BOOLEAN
4	The colour of a sprite	STRING
5	The counter in a loop	INTEGER
6	A URL	STRING
7	A key-press	CHARACTER

\* Telephone numbers can start with a leading zero which would be ignored if they were stored as an integer.



**Activity: Structured data types (page 56)****Q9:**

No.	Value	Data type
1	A list of names	ARRAY of STRING
2	A set of test scores out of 50	ARRAY of INTEGER
3	The characters in a sentence	ARRAY of CHARACTER
4	The average temperatures during last month	ARRAY of REAL
5	The last five Google searches you made	ARRAY of STRING
6	Whether or not a class of pupils have passed an exam	ARRAY of BOOLEAN

**Activity: Multiple data types (page 57)****Q10:**

No.	Records	Data types
1	Name, address and Scottish Candidate Number (SCN) for a list of pupils.	STRING, STRING, STRING
2	Pupil ID, test score and pass/fail for a class	STRING, INTEGER, BOOLEAN
3	Weapon name, ammunition type and damage value in a First Person Shooter game	STRING, STRING, INTEGER

**Quiz: Pseudocode (page 57)****Q11:** e) Greg**Q12:** b) Jim**Q13:** g) 47**Q14:** d) W**Practical task: Handling records (page 58)****Example answers****Visual Basic 6:**

Module code:

```

1 Type game_character
2     name As String
3     weapon As String
4     danger As Integer
5 End Type

```

Program code:

```

1 Dim enemy(3) As game_character
2 Private Sub fill_array()
3 enemy(0).name = "troll"
4 enemy(0).weapon = "axe"
5 enemy(0).danger = 3
6 enemy(1).name = "Dwarf"
7 enemy(1).weapon = "spell"
8 enemy(1).danger = 5
9 enemy(2).name = "wizard"
10 enemy(2).weapon = "staff"
11 enemy(2).danger = 9
12 enemy(3).name = "ghost"
13 enemy(3).weapon = "ectoplasm"
14 enemy(3).danger = 2
15 End Sub
16
17 Private Sub CmdDisplayRecord_Click()
18
19 fill_array
20 For Counter = 0 To 3
21     Picture1.Print "Name: " ;enemy(Counter).name
22     Picture1.Print " Weapon: ";enemy(Counter).weapon
23     Picture1.Print " Danger level:";enemy(Counter).danger
24 Next Counter
25 End Sub

```

## Python

```

1 #declare dictionary
2 enemy = {}
3
4 enemy[0] = ["troll","axe",3]
5 enemy[1] = ("dwarf","spell",5)
6 enemy[2] = ("wizard", "staff",9)
7 enemy[3] = ("ghost","ectoplasm",2)
8
9 for counter in enemy:
10     print ("Name:", enemy[counter][0])
11     print ("Weapon:", enemy[counter][1])
12     print ("Danger level:", enemy[counter][2])
13     print()

```

## Java




In Java a record is just an object which has instance variables but not instance methods.

```

1 Class enemy{
2     String name;
3     String weapon;
4     int danger;
5 }

```

**Practical task: Structured data types (page 58)****Example answer**

Data Type	Visual Basic 6	Snap! (formerly called BYOB)	Python
ARRAY	Array		list / array
STRING	String		list / string
RECORD	Record		dictionary

**Quiz: Identifying structured data types (page 59)**

**Q15:** a) hydra

**Q16:** b) Run!

**End of topic 4 test (page 61)**

**Q17:**

- a) REAL
- b) BOOLEAN
- c) ARRAY of STRING
- d) ARRAY of INTEGER
- e) ARRAY of INTEGER
- f) ARRAY of CHARACTER
- g) ARRAY of STRING
- h) ARRAY of BOOLEAN
- i) RECORD
- j) RECORD

**Q18:** Example answer:

```
1 RECORD smartphone IS {STRING make, STRING model, INTEGER capacity,
  BOOLEAN fp_scanner}
```

**Topic 5: Implementation: Algorithm specification****Quiz: Revision (page 65)****Q1:** c) ARRAY of STRING**Q2:** c) The position in an ARRAY**Q3:** b) Pseudocode**Q4:** a) A fixed loop**Q5:** c) Running total**Practical task: Algorithms 1 (page 68)****Possible algorithm for solution**

```
1 PROCEDURE inputValidation()  
2  
3   RECEIVE userInput FROM (INTEGER) KEYBOARD  
4  
5   WHILE userInput < 0 OR userInput > 100 DO  
6  
7     SEND "Input must be between 1 and 100 inclusive" TO DISPLAY  
8     RECEIVE userInput FROM (INTEGER) KEYBOARD  
9  
10  END WHILE  
11  
12 END PROCEDURE
```

**Practical task: Algorithms 2 (page 68)****Possible solution**

```
1 PROCEDURE inputValidation()  
2  
3   RECEIVE userInput FROM (STRING) KEYBOARD  
4  
5   WHILE userInput ≠ ["Y"] AND userInput ≠ ["N"] AND userInput ≠ ["y"]  
6     AND userInput ≠ ["n"] DO  
7  
8     SEND "Input must be Y or y or N or n" TO DISPLAY  
9     RECEIVE userInput FROM (STRING) KEYBOARD  
10  
11  END WHILE  
12  
13 END PROCEDURE
```

**Practical task: Algorithms 3 (page 69)****Possible solution**

```
1 PROCEDURE inputValidation()  
2  
3   validInput = false  
4   validLength = 12  
5  
6   REPEAT  
7  
8     RECEIVE userInput FROM (STRING) KEYBOARD  
9  
10    IF length(userInput) = validLength THEN  
11      validInput = true  
12    END IF  
13  
14    FOR EACH letter FROM userInput DO  
15      IF letter < 0 OR letter > 9  
16        THEN validInput = false  
17      END IF  
18    END FOR EACH  
19  
20    IF validInput = false THEN  
21      SEND "Input must contain exactly " & validLength & " digits" TO  
22        DISPLAY  
23    END IF  
24  UNTIL validInput = true  
25  
26 END PROCEDURE
```

**Possible solution using mid\$ function:**

```
1 PROCEDURE inputValidation()
2
3   SET validInput TO false
4   SET validLength TO 12
5
6   REPEAT
7
8     RECEIVE userInput FROM (STRING) KEYBOARD
9
10    IF length(userInput) = validLength THEN
11      SET validInput TO true
12    END IF
13
14    FOR counter FROM 1 TO validlength DO
15      IF mid$(userInput, counter, 1) < 0 OR mid$(userInput, counter, 1) >
16        9 THEN
17        SET validInput TO false
18      END IF
19    END FOR
20
21    IF validInput = false THEN
22      SEND "Input must contain exactly " & validLength & " digits" TO
23        DISPLAY
24    END IF
25
26  UNTIL validInput = true
27 END PROCEDURE
```

**Activity: Find the maximum value in an array (page 70)****Q6:** 56**Q7:** 74**Q8:** 105**Q9:** 74**Q10:** 149

**Practical task: Find winner (page 72)****Possible solution**

```
1 PROCEDURE findWinner()  
2  
3   SET foundAt TO 0  
4   SET bestTime TO times[0]  
5  
6   FOR index FROM 1 TO 9 DO  
7     IF bestTime > times[index] THEN  
8       SET bestTime TO times[index]  
9       SET foundAt TO index  
10    END IF  
11  END FOR  
12  SEND "The winner was "& names[foundAt] & "with a time of "& bestTime  
    TO DISPLAY  
13  
14 END PROCEDURE
```

**Activity: Counting Occurrences (page 73)****Q11: 3****Q12: 3****Q13: 2****Q14: 1****Q15: 1**

**Practical task: Counting Occurrences (page 74)****Possible solution**

```
1 PROCEDURE countOccurrences()  
2  
3 RECEIVE phrase FROM (STRING) KEYBOARD  
4  
5 SET numberFound TO 0  
6 SET itemToFind TO "e"  
7  
8 FOR EACH letter FROM phrase DO  
9   IF letter = itemToFind THEN  
10    SET numberFound TO numberFound + 1  
11  END IF  
12 END FOR EACH  
13  
14 SEND "There were " & numberFound & "occurrences of the letter e  
15     in the phrase you typed" TO DISPLAY  
16  
17 END PROCEDURE
```

**Possible solution using mid\$ function:**

```
1 PROCEDURE countOccurrences()  
2  
3 RECEIVE phrase FROM (STRING) KEYBOARD  
4  
5 SET numberFound TO 0  
6 SET itemToFind TO "e"  
7  
8 FOR counter FROM 1 TO length(phrase) DO  
9   IF mid$(phrase, counter, 1) = itemToFind THEN  
10    SET numberFound TO numberFound + 1  
11  END IF  
12 END FOR  
13  
14 SEND "There were " & numberFound & "occurrences of the letter e  
15     in the phrase you typed" TO DISPLAY  
16  
17 END PROCEDURE
```

**Activity: Linear Search (page 75)****Q16:** 2**Q17:** 7**Q18:** FALSE**Q19:** TRUE



**End of topic 5 test (page 78)**

**Q20:** b) Input validation

**Q21:** d) Finding the maximum

**Q22:** b) A conditional loop and a boolean variable

**Q23:** a) A fixed loop

**Q24:** a) Counting occurrences

**Topic 6: Implementation: Computational constructs****Quiz: Revision (page 83)****Q1:** a) Assignment**Q2:** c) A user defined function**Q3:** b) A simple conditional**Q4:** c) A complex conditional**Q5:** "Number OK"**Q6:** "Invalid Entry"**Q7:** a) A fixed loop**Practical task: User-defined functions (page 93)****Possible solutions:**

```
1 FUNCTION newRandom() RETURNS INTEGER
2
3     randomNumber = Rand(50) + 50
4
5     RETURN randomNumber
6
7 END FUNCTION
8
9
10 STRING FUNCTION userName() RETURNS STRING
11
12 RECEIVE userInput FROM (STRING) KEYBOARD
13
14 WHILE length(userInput) > 10
15     SEND "Input must be Less than 10 Characters" TO DISPLAY
16     RECEIVE userInput FROM (STRING) KEYBOARD
17 END WHILE
18
19 RETURN userInput
20
21 END FUNCTION
```

**Practical task: Parameters 2 (page 94)****Possible solution**

```
1 FUNCTION double(value) RETURNS INTEGER
2
3     value = value * 2
4     RETURN value
5
6 END FUNCTION
7
8
9 FUNCTION userName(foreName, yearOfBirth) RETURNS STRING
10
11     newString = foreName & yearOfBirth
12     RETURN newString
13
14 END FUNCTION
```

**Practical task: Parameters 3 (page 97)****Possible solution**

```
1 SET scores TO [4,12,6,23,6,34,2,6,17,2]
2 SET names TO ['Fred','Barney','Wilma','Betty','Jim','Sally','Joe',
3 'Zaphod','Greg','Jo']
4
5
6
7 PROCEDURE findWinner(names, scores)
8
9     SET maximumScore TO scores[0]
10    SET winner TO names[0]
11    FOR counter FROM 1 TO 9 DO
12        IF maximumScore < scores[counter] THEN
13            SET maximumScore TO scores[counter]
14            SET winner TO names[counter]
15        END IF
16    END FOR
17    SEND "The winner was " & winner TO DISPLAY
18
19 END PROCEDURE
```

**Practical task: Sequential files (page 97)****Possible solution**

```
1 SET names TO['Fred','Barney','Wilma','Betty','Jim','Sally',
2 'Joe','Zaphod','Greg','Jo']
3
4
5 PROCEDURE findWinner(names, scores)
6
7   GetData(scores)
8
9   SET maximumScore TO scores[0]
10  SET winner TO names[0]
11  FOR counter FROM 1 TO 9 DO
12    IF maximumScore < scores[counter] THEN
13      SET maximumScore TO scores[counter]
14      SET winner TO names[counter]
15    END IF
16  END FOR
17  SEND "The winner was " & winner TO DISPLAY
18
19 END PROCEDURE
20
21 PROCEDURE getData(scores)
22   <open file "scores.txt">
23   FOR counter FROM 0 TO 9 DO
24     RECEIVE scores[counter] FROM (INTEGER) "scores.txt"
25   END FOR
26   <close file "scores.txt">
27 END PROCEDURE
```

**End of topic 6 test (page 100)**

**Q8:** c) Counting occurrences

**Q9:** d) String value

**Q10:** b) Actual parameters

**Q11:** a) Formal parameters

**Q12:** a) ORANGES A

**Q13:** c) 18

**Q14:** a) A reference parameter

**Q15:** b) A value parameter

**Q16:** c) 1

**Topic 7: Testing****Quiz: Revision (page 105)****Q1:** a) 0, 1,10,50, 99,100, 101, X**Q2:** b) scores**Q3:**

```

1 Line 4 SEND ["Input must be between 1 and 100 "] TO DISPLAY
2 Line 5 RECEIVE userInput FROM (INTEGER) KEYBOARD

```

**Q4:** Short statements embedded in the source code describing how that part of the program functions in order to aid code readability.**Quiz: Debugging (page 107)****Q5:**

- Division by zero
- Out of memory

**Q6:** The error is a logic (semantic) error. because the total variable is set to zero every time the loop repeats. It should be set to zero before the loop starts.**Practical task: Trace tables (page 110)****Possible solutions**

maximumValue	counter	numbers[counter]
3	1	15
15	2	4
15	3	7
15	4	8

Output: The largest value was 15.

**End of topic 7 test (page 112)****Q7:**

- Missing semi colon
- IF without END IF
- WHILE without DO

**Q8:**

- Division by zero
- Overflow error

**Q9:**

minimumValue	counter	numbers[counter]
17	1	15
15	2	4
4	3	7
4	4	8

**Output:** The smallest value was 4.**Q10:** d) Extreme data**Q11:** b) False**Q12:** a) True**Q13:**

value	no display	counter
1		0
1	1	1
2	2	2
4	4	3
7	7	4
11	11	5
16	16	6

**Q14:** c) Global variables**Q15:**

- To avoid duplication of tests if errors are discovered later on answer.
- To help ensure that the testing is both comprehensive and systematic.

**Q16:** d) Dry run

**Topic 8: Evaluation****Quiz: Revision (page 117)**

**Q1:** a) Indentation and b) Internal commentary

**Q2:** d) How well a program can cope with invalid data.

**Activity: Evaluation terminology (page 121)**

**Q3:**

<b>Term</b>	<b>Description</b>
Robust	Ability of a program to keep running even when external errors occur.
Usability	The program's interface is clear and can be operated by the intended users.
Efficient	Whether the program wastes memory or processor time.
Maintainable	Has the program been designed to easily altered by another programmer.
Fit for purpose	Does the program fulfil all the requirements of the specification.

**End of topic 8 test (page 122)**

**Q4:** a) Fitness for purpose

**Q5:** d) Efficiency

**Q6:** b) Usability

**Q7:** d) Maintainability

**Topic 9: End of unit 1 test****End of unit 1 test (page 124)****Q1:**

1. Responsiveness to changed circumstances.
4. Reduced development time.

**Q2:** c) They are suited to large-scale team-based software development.

**Q3:** d) High Level Language choice

**Q4:** c) Source code

**Q5:** b) breaking a large and complex problem into smaller, more manageable sub-problems.

**Q6:** a) Wireframe

**Q7:** c) An INTEGER array of 20

**Q8:** d) INTEGER

**Q9:** c) STRING array and REAL array

**Q10:** a) two arrays containing linked data with the same index values.

**Q11:** c) A single record structure

**Q12:** b) Counting occurrences

**Q13:** b) To terminate the loop when the item is found.

**Q14:** a) Counting occurrences

**Q15:** b) Finding the Minimum

**Q16:** a) Counting occurrences

**Q17:** d) Linear search

**Q18:** d) 36

**Q19:** a) formal parameters

**Q20:** b) actual parameters

**Q21:**

1. A function returns a value.
3. A function can be user-defined.

**Q22:**

1. Missing semi colon
3. IF without END IF
6. WHILE without DO

**Q23:**



- Int()
- Asc()
- Mod()

**Q24:** a) CLOSE

**Q25:** b) Break-run

**Q26:** b) Break-run

**Q27:** a) Robustness

**Q28:** b) Functional Requirements from the software specification.