



Software Design and Development Stage 6

Software and Course Specifications
Higher School Certificate 2012

Original published version updated:

December 2010

October 2011

June 2013

© 1999 Copyright Board of Studies NSW for and on behalf of the Crown in right of the State of New South Wales.

This document contains Material prepared by the Board of Studies NSW for and on behalf of the State of New South Wales. The Material is protected by Crown copyright.

All rights reserved. No part of the Material may be reproduced in Australia or in any other country by any process, electronic or otherwise, in any material form or transmitted to any other person or stored electronically in any form without the prior written permission of the Board of Studies NSW, except as permitted by the Copyright Act 1968. School students in NSW and teachers in schools in NSW may copy reasonable portions of the Material for the purposes of bona fide research or study. Teachers in schools in NSW may make multiple copies, where appropriate, of sections of the HSC papers for classroom use under the provisions of the school's Copyright Agency Limited (CAL) licence.

When you access the Material you agree:

- to use the Material for information purposes only
- to reproduce a single copy for personal bona fide study use only and not to reproduce any major extract or the entire Material without the prior permission of the Board of Studies NSW
- to acknowledge that the Material is provided by the Board of Studies NSW
- not to make any charge for providing the Material or any part of the Material to another person or in any way make commercial use of the Material without the prior written consent of the Board of Studies NSW and payment of the appropriate copyright fee
- to include this copyright notice in any copy made
- not to modify the Material or any part of the Material without the express prior written permission of the Board of Studies NSW.

The Material may contain third-party copyright materials such as photos, diagrams, quotations, cartoons and artworks. These materials are protected by Australian and international copyright laws and may not be reproduced or transmitted in any format without the copyright owner's specific permission. Unauthorised reproduction, transmission or commercial use of such copyright materials may result in prosecution.

The Board of Studies has made all reasonable attempts to locate owners of third-party copyright material and invites anyone from whom permission has not been sought to contact the Copyright Officer, ph (02) 9367 8289, fax (02) 9279 1482.

Images on pages 85–87 appear with kind permission of Arthur Galletly and are edited for use here.

Published by
Board of Studies NSW
GPO Box 5300
Sydney NSW 2001
Australia

Tel: (02) 9367 8111
Fax: (02) 9367 8484
Internet: www.boardofstudies.nsw.edu.au

20110819

20130447

Contents

1	Foreword	4
2	Introduction	5
3	Syllabus references.....	6
4	General specifications	12
4.1	Systems modelling tools.....	12
4.2	Project management tools.....	21
4.3	Meta languages	23
5	Software specifications	25
5.1	Language specifications	25
5.2	Options.....	28
5.2.1	Option 1: Programming paradigms	28
5.2.2	Option 2: The interrelationship between software and hardware	31
6	Methods of algorithm description	34
6.1	Introduction	34
6.2	Overview of two methods.....	35
6.3	Programming structures.....	37
7	Standard algorithms.....	58
8	Advanced data structures	70
9	Searching and sorting algorithms	75
10	Documentation	85

1 Foreword

The HSC Software and Course Specifications for Software Design and Development contain information for the Higher School Certificate from 2012. This information is relevant to students studying the Preliminary course from 2011. This document is an integral part of the course content and needs to be referred to regularly in conjunction with the syllabus.

This HSC Software and Course Specifications document should be read in conjunction with:

- *Amended Software Design and Development Stage 6 Syllabus*
- Official Notices in *Board Bulletins*

The Board of Studies reserves the right to make changes to the Software and Course Specifications. As they are reviewed, the amendments will be published and notified in the Official Notices on the Board of Studies website www.boardofstudies.nsw.edu.au.

Curriculum advice may be obtained on:

Phone (02) 9367 8111

Fax (02) 9367 8476

Correspondence should be addressed to:

Board of Studies NSW
GPO Box 5300
Sydney
NSW 2001

2 Introduction

This document provides content and clarification of the depth of study required for concepts in the amended *Software Design and Development Stage 6 Syllabus*. The document should be read in conjunction with the amended *Software Design and Development Stage 6 Syllabus*.

The document is available on the board's website so that it can be regularly updated.

3 Syllabus references

This section defines and clarifies selected syllabus content which may change over time.

It is intended that specific content in this section will be reviewed and updated as the need arises to maintain currency of the syllabus.

Preliminary Course

8.1.1 Social and ethical issues

Students learn about:	Students learn to:	Syllabus page
Intellectual Property <ul style="list-style-type: none"> • use of software covered by a licence agreement, such as: <ul style="list-style-type: none"> – public domain – shareware – freeware – open source (GNU licence) – site licence – creative commons 		14

8.1.2 Hardware and software

Students learn about:	Students learn to:	Syllabus page
Hardware	<ul style="list-style-type: none"> • describe how data is captured, stored, manipulated or displayed on a variety of hardware devices. Select ONE device from each: <ul style="list-style-type: none"> – Input: <ul style="list-style-type: none"> – keyboard – mouse – scanner – radio frequency identification (RFID) – barcode reader – graphics tablet – microphone – Output: <ul style="list-style-type: none"> – laser printer – inkjet printer – cathode ray tube (CRT) – LCD display – plasma display – CD writer/burner – DVD writer/burner – Data projector – Speakers 	16

HSC Course

9.1.1 Social and ethical issues

Students learn about:	Students learn to:	Syllabus page
Software piracy and copyright <ul style="list-style-type: none"> current and emerging technologies used to combat software piracy such as: <ul style="list-style-type: none"> non-copyable data sheet use of serial numbers site licence, installation counter on a network encryption key registration code back-to-base authentication 		34
Legal implications <ul style="list-style-type: none"> national and international legal action resulting from software development. Identify issues raised in cases at both national and international level. such as: <ul style="list-style-type: none"> national: <ul style="list-style-type: none"> RACV vs Unisys Microsoft vs Netscape NSW T Card system international: <ul style="list-style-type: none"> search engines (eg Google vs national censorship laws) Metallica vs Napster 		35

9.1.2 Application of software development approaches

Students learn about:	Students learn to:	Syllabus page
Software development approaches <ul style="list-style-type: none"> trends in software development <ul style="list-style-type: none"> changing nature of applications <ul style="list-style-type: none"> web-based software learning objects widgets apps and applets Web 2.0 tools cloud computing mobile phone technology collaborative environments 		37

9.2.2 Planning and designing software solutions

[illegible]

9.2.3 Implementation of software solution

Students learn about:	Students learn to:	Syllabus page
Emerging technologies <ul style="list-style-type: none"> the effect of emerging hardware and software technologies on the development process, such as: <ul style="list-style-type: none"> iPhone Wii remote handheld communication devices scanning pen biometric devices multi-point surface software radio-frequency identification (RFID) social networking software 		47

9.2.4 Testing and evaluating software solutions

Students learn about:	Students learn to:	Syllabus page
Reporting on the testing process <ul style="list-style-type: none"> documentation of the test data and output produced. (see Section 10 for a sample testing report) 		49

9.2.5 Maintaining software solutions

Students learn about:	Students learn to:	Syllabus page
Documenting changes <ul style="list-style-type: none"> using CASE tools to monitor changes and versions (see Section 10 for a sample of source code documented to reflect changes) 		51

9.3 Developing a solution package

Students learn about:	Students learn to:	Syllabus page
Designing and developing a software solution to a complex problem	<ul style="list-style-type: none"> use a logbook to document the progress of their project. <p>Logbooks could include:</p> <ul style="list-style-type: none"> – email messages – spreadsheets – blogs – handwritten dated entries – electronic journal entries <p>Each entry should include:</p> <ul style="list-style-type: none"> – time and date – tasks achieved – difficulties and solutions – ideas and thoughts – reflection on progress – upcoming tasks – reference to resources used 	53

9.4 Options

9.4.1 Option 1 Programming Paradigms

Students learn about:	Students learn to:	Syllabus page
Logic paradigm	<ul style="list-style-type: none"> recognise representative fragments of code written using the logic paradigm (see Section 5.2.1 for specific examples of algorithm fragments for the logic paradigm) 	56
Object oriented paradigm	<ul style="list-style-type: none"> recognise representative fragments of code written using the object oriented paradigm (see Section 5.2.1 for specific examples of algorithm fragments for the object oriented paradigm) 	56

9.4.2 Option 2: The interrelationship between software and hardware

Students learn about:	Students learn to:	Syllabus page
Representation of data within the computer <ul style="list-style-type: none"> character representation, namely: <ul style="list-style-type: none"> ASCII Unicode (see Section 5.2.2) 		58
Programming of hardware devices <ul style="list-style-type: none"> processing an input data stream from sensors and other devices <ul style="list-style-type: none"> interpreting the data stream Devices such as: <ul style="list-style-type: none"> security cameras USB mouse biometric scanners generating output to an appropriate device <ul style="list-style-type: none"> required trailer information Devices such as: <ul style="list-style-type: none"> model helicopter model train/car/boat plotter cutter machines automated jar filling line with valves and conveyor belt scrolling display system for one line text display modem printer 		60

4 General specifications

4.1 Systems modelling tools

IPO diagrams

These diagrams are used to document a system by identifying the inputs into each major process, the general nature of these processes, and the outputs produced.

The IPO diagram is in the form of a table with 3 columns, one for Input, Process and Output. The following IPO diagram describes the voting system subsequently shown as a data flow diagram.

Input	Process	Output
VoterID	Check if on electoral roll Check if already voted Retrieve candidates from endorsed candidates file	'Not allowed to vote' 'Already voted' List of endorsed candidates
Vote, candidate name	Update voter's record with 'voted' flag = 1 Retrieve candidate record Increment count in candidates record and rewrite	Updated voter's record Updated candidates record 'Thank you for voting'
Candidates file	Read in each candidates record to an array of records Sort these records into descending order of votes	Report of results showing candidate name, number of votes

Context diagrams

Context diagrams are used to represent an overview of the system. The system is shown as a single process along with the inputs and outputs. The external entities are connected to the single process by data flow arrows. Each element represented is labelled. A context diagram does not show data stores or internal processes.

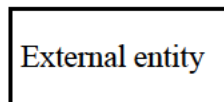
SYMBOLS USED



A circle represents a single process representing the entire system

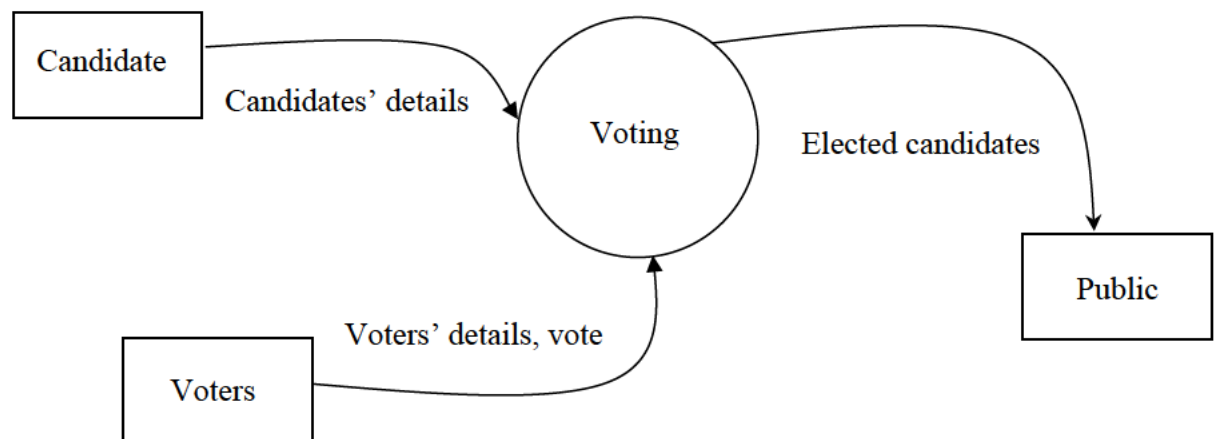


A curved arrow represents the flow of data between the single process and external entities.



A rectangle represents any person or organisation, source or sink that provides data to the system or receives data from the system.

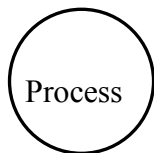
The example below is a context diagram that represents a voting system.



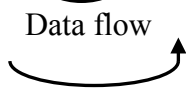
Data flow diagrams

Data flow diagrams represent a system as a number of processes that together form a single system. A data flow diagram is a refinement of a context diagram. Data flow diagrams therefore show a further level of detail not seen in the context diagram. Data flow diagrams identify the source of data, its flow between processes and its destination along with data generated by the system.

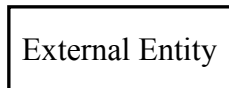
SYMBOLS USED



A circle represents a process. Processes use inputs to generate outputs.



A labelled, curved arrow represents the flow of data between processes, data stores and external entities.



A rectangle represents any person or organisation, source or sink that provides data to the system or receives data from the system.

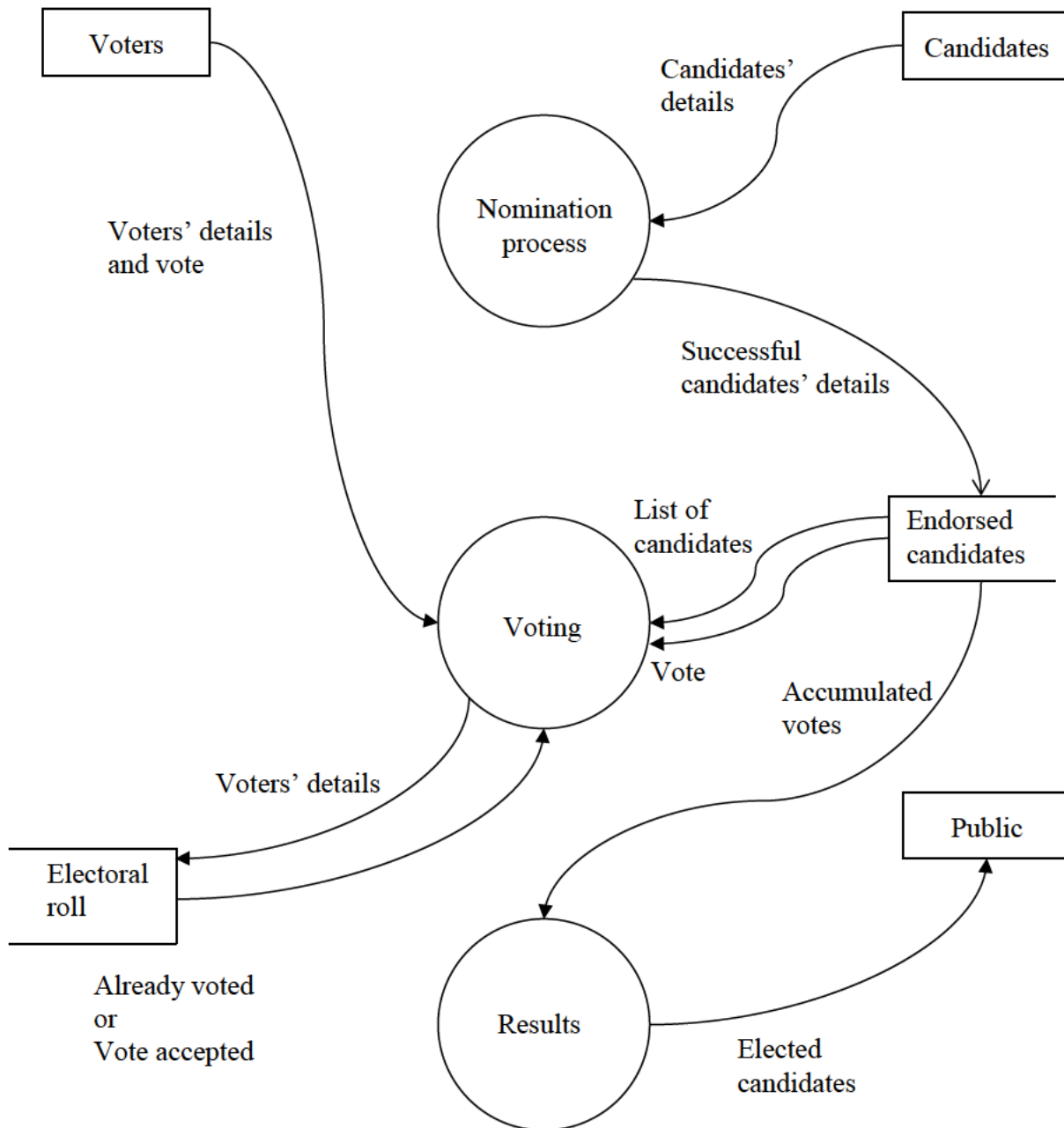


An open-ended rectangle represents a data store. It can be an electronic file or non-computer storage.



It is important that the labels used on each of these symbols are meaningful.

The example below models a voting system in more detail. Note that both the context diagram and the data flow diagram for the system must reflect the same external entities and data moving between them.



Structure charts

Structure charts represent a system by showing the separate modules or subroutines that comprise the system and their relationship to each other.

Rectangles are used to represent modules or subroutines, with lines used to show the connections between them. The chart is read from top to bottom, with component modules or subroutines on successively lower levels, indicating these modules or subroutines are called by the module or subroutine above. For all modules or subroutines called by a single module or subroutine, the diagram is read from left to right to show the order of execution.

SYMBOLS USED



Data movement between modules or subroutines (usually passed as parameters) is shown with the use of arrows.



A filled circle is used to indicate a flag or control variable.



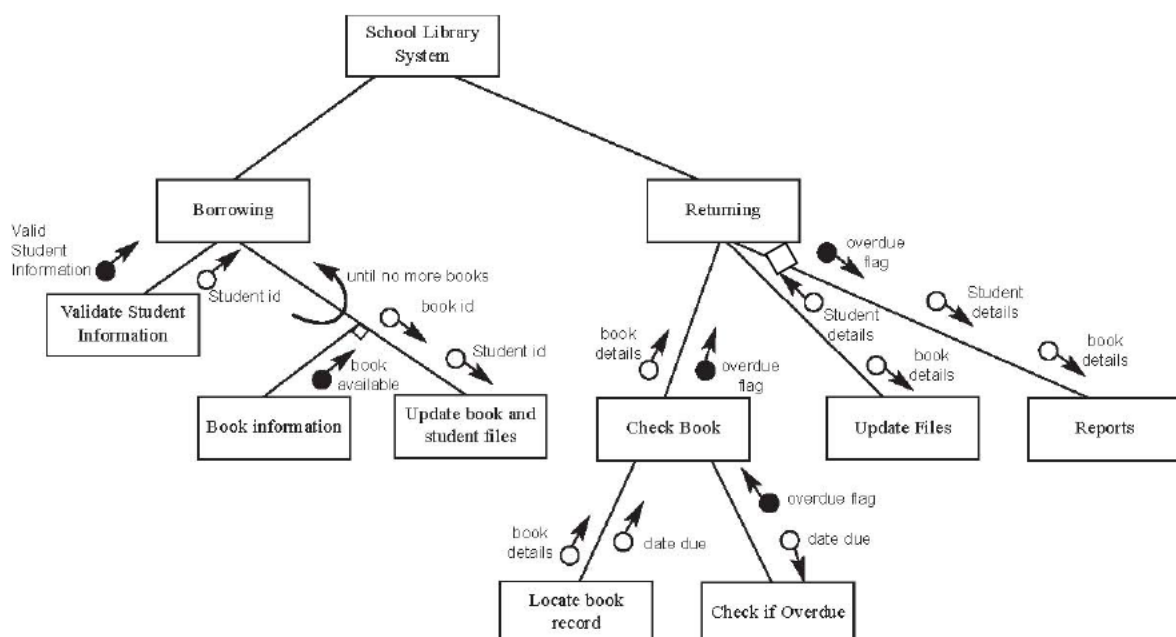
A decision (i.e. optional execution of modules or subroutines) is indicated by use of a small diamond at the intersection of the connecting lines between modules or subroutines that are called as the result of a binary or multi-way selection.

Alternatively, the diamond may appear on a single connecting line if calling that module or subroutine is optional. In the diagram shown, a report may not be required to be produced each time a book is returned.



Repetition (execution of a particular module or subroutine or set of modules or subroutines multiple times) is shown by a curved arrow.

The following example represents a library system.



System flowcharts

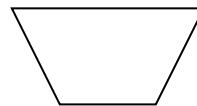
System flowcharts are a diagrammatic way of representing the system to show the flow of data, the separate modules comprising the system and the media used. Standard symbols include those used for representing major processes and physical devices that capture, store and display data. Many of these symbols have become outdated as a result of changes in technology.

Note that system flowcharts are distinctly different from program flowcharts, which are used to represent the logic in an algorithm. They do not use a start or end symbol, and are not intended to represent complex logic.

Standard symbols used in systems flowcharts



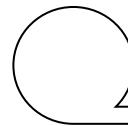
Input/output



Manual operation



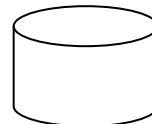
Paper document



Magnetic tape



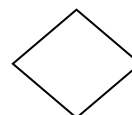
Online display



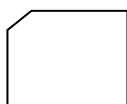
Disk drive



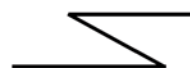
Online input



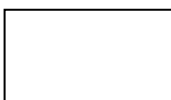
Decision



Punched card

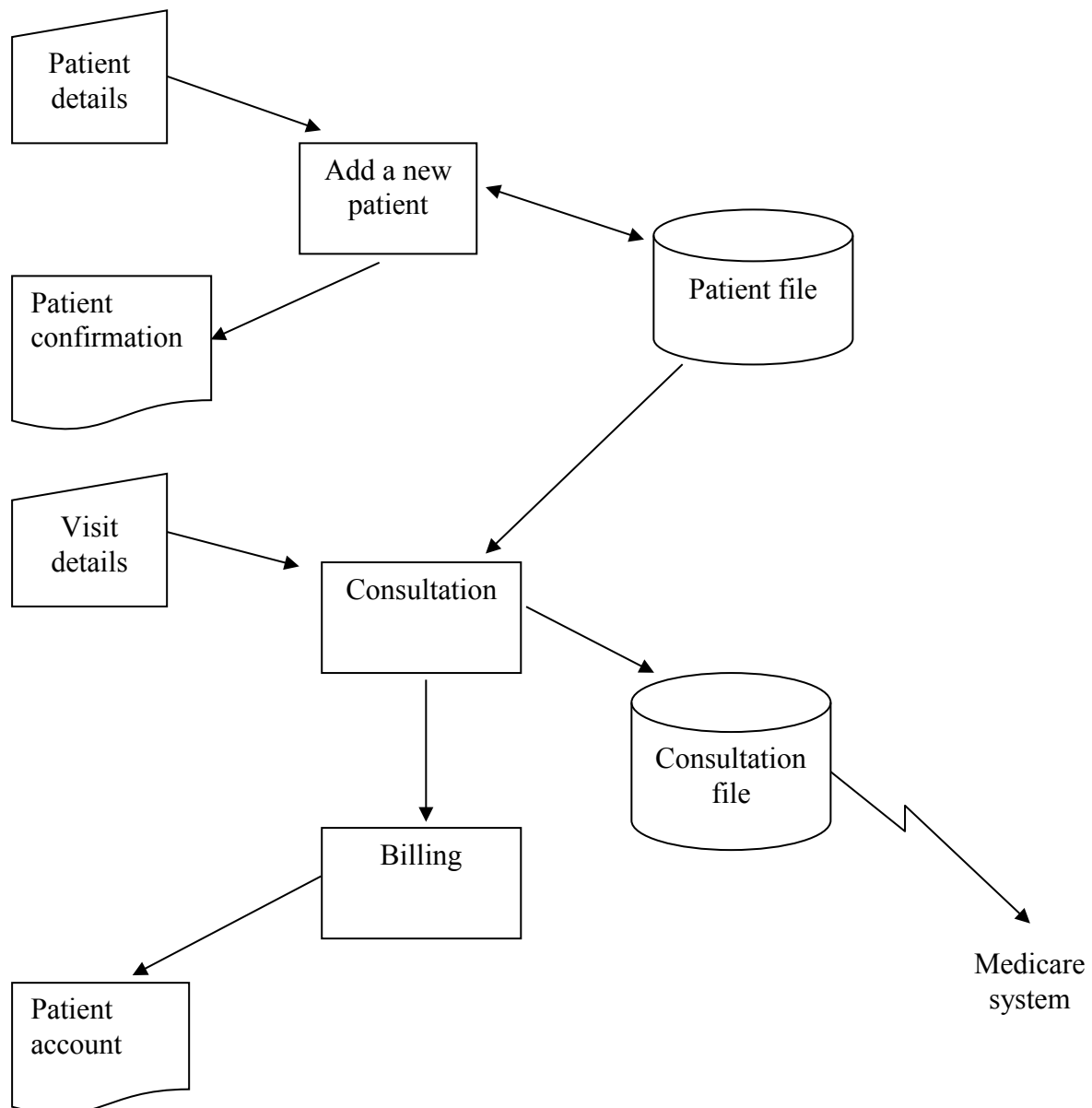


Telecommunications link



Process

The following systems flowchart represents part of the current system used by a local doctor in managing their patient information.



Data dictionary

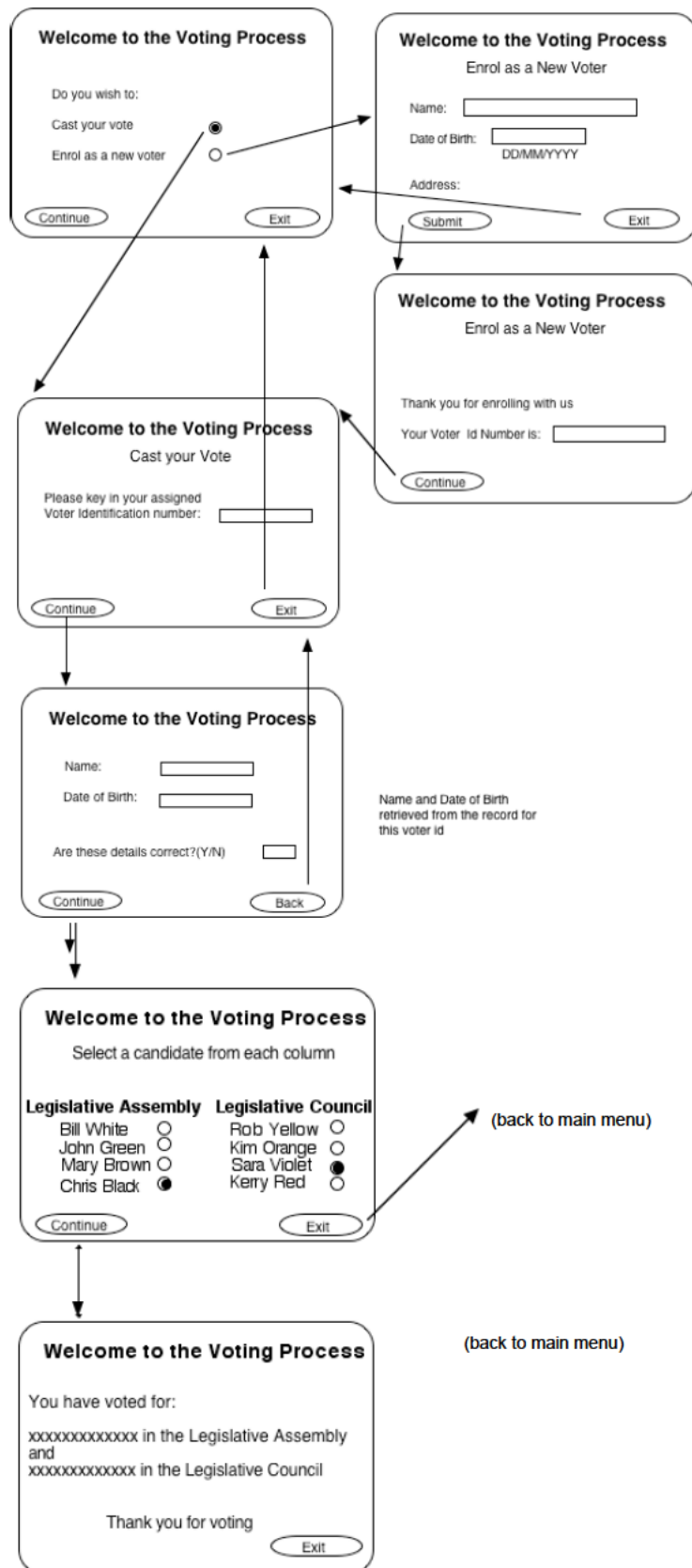
A data dictionary is a comprehensive description of each data item in a system. This commonly includes: variable name, size in bytes, number of characters as displayed on screen, data type, format including number of decimal places (if applicable) and a description of the purpose of each field together with an example.

Data item	Data type	Format	Number of bytes required for storage	Size for display	Description	Example	Validation
UserId	String	XXNNN	5	5	Uniquely identifies user. First letters of names followed by unique 3-digit identifier	PT173	
FirstName	String		15	15	Given name of employee	George	
Surname	String		25	25	Family name of employee	Wu	
DOB	Floating Point (date format)	YYYY/MM/DD	4	10	Birth date of employee	1953/10/05	Valid date less than today
TimesLate	Integer	NNN	2	3	Count of times late to work	47	Integer between 0 and 999
PayRate	Floating Point	\$NNN.NN	4	7	Hourly rate of pay	\$024.37	Greater than 20, less than 400
UnionMember	Boolean	X	1	1	Y or N	N	
Gender	Boolean	X	1	1	M or F	F	
Departments	Array (string)		20 * number of departments	20 * number of departments	Names of departments in organisation	Administration Finance Marketing	
TaxScale	Array of records						
UpperLimit	Floating Point	\$NN,NNN.NN	4	10	Upper limit of salary to which this tax rate applies	\$32,000.65	
TaxRate	Floating Point	NN.N%	4	5	% tax applied to salary	46.0%	
RetiredEmployee	Record		29		Data written to retirement file when employee retires		
UserId	String	XXNNN	5	5	Uniquely identifies user. First letters of names followed by unique 3-digit identifier	PT173	
TerminationDate	Floating Point (date format)	DD/MM/YY	4	8	Date of retirement	01/04/10	
Reason	String		20	20	Reason for retirement	Health	From a drop-down list
Retirees	File (Sequential)				Collection of all retired employee records for last 10 years		

Storyboard

A storyboard shows the various interfaces (screens) in a system as well as the links between them. The representation of each interface should be detailed enough for the reader to identify the purpose, contents and design elements. Areas used for input, output and navigation should be clearly identified and labelled. Any links shown between interfaces should originate from the navigational element that triggers the link.

This storyboard represents an online voting system. Elements of each screen are clearly identified and the links between screens are clearly shown.



4.2 Project management tools

Tools for system documentation

The software used by students (inside or outside the classroom) must allow the production and maintenance of:

- manuals, incorporating screenshots, table of contents, index and footnotes
- algorithms (flowcharts and/or pseudocode)
- system flowcharts (using specialist software such as Visio or SmartDraw)
- structure charts
- data flow diagrams (using specialist software such as Visio or SmartDraw)
- context diagrams (using specialist software such as Visio or SmartDraw)
- storyboards (using presentation software such as PowerPoint)
- data dictionary (using spreadsheet software or a table in a word processed document)
- IPO diagrams (using spreadsheet software or a table in a word processed document)

It is important that students practise using a range of these tools as they implement their smaller projects as well as their major project.

Gantt charts

A Gantt chart displays each of the component tasks on an estimated timeline. The estimated time required for each task and dependencies between related tasks should be clearly shown. The time scale should be clearly indicated with dates included and important milestones in the project clearly marked.

These charts should be produced using specialist software, such as Visio, MS Project or TurboProject, as it introduces students to advanced project management features.

A selection of appropriate charts can be found at:

www.total-quality-management-software.com/gantt-chart-examples.asp

www.4csys.com/images/gantt_chart_2.jpg

www.projectmanager.com/images/showcase/gantt-charts-1.jpg

www.hsc.csu.edu.au/sdd/core/package/solution_package/gantt_chart/projmgmtscreendump.gif

Note that the level of sophistication and the amount of detail included in these charts reflect the type and scope of the specific projects. The detail must be sufficient to allow effective time management of the project.

Students are expected to produce an initial Gantt chart before starting work on their project. It is important for students to regularly update their Gantt charts to reflect actual versus estimated times for tasks at regular intervals during the project's development.

Log books

Log books are used to document the progress of a project. Entries should include:

- date
- description of the progress (or lack thereof) made since the last entry
- tasks achieved
- descriptions of stumbling blocks or issues encountered and how they were managed
- details of possible approaches for upcoming tasks
- reflective comments
- reference to resources used.

Log books may be produced using spreadsheets, blogs, handwritten entries or electronic journal entries. Email messages to a fellow developer may be useful, as they contain time and date stamps.

A sample log book entry may look like:

April 22nd 2010 – problems with images

I am so pleased the coding for my mainline is finally done! This morning I spent some time importing the pictures for each screenshot. I had some trouble with it. The one problem I still have to fix is the transparency so I will try to import each in GIF format and make the background transparent. Hopefully it will work.

Note that the comment is reflective, describes what has been achieved and identifies a particular stumbling block and possible approaches to dealing with it.

4.3 Meta languages

Extended Backus-Naur Form (EBNF)

EBNF is metalanguage used to define the syntax of any programming language. The following symbols are used:

= 'is defined as'

| indicates a choice between alternatives

terminal symbol there is no need for further definition of this item. It may be a symbol or a reserved word (such as PRINT, IF...)

< > Used to specify a term that will be subsequently defined

[] indicate an optional part of a definition

{ } indicate a possible repetition (0 or more times)

() used to group elements together

Example Letter = A | B | C

Digit = 0 | 1 | 2 | 3 | 4 | 5

Identifier = <Letter> {<Letter> <Digit>}

Assignment Statement

LET <Identifier> = <Identifier>

Interpretation An identifier is defined to be a Letter followed by one or more Letters or Digits. *Letter* is a non-terminal symbol defined as A or B or C
Digit is a non-terminal symbol defined as 0 or 1 or 2 or 3 or 4

Using these definitions, a valid Identifier could therefore be any of the following:

B24A

C

ABC1

A valid assignment statement could be
LET C = B24A

Railroad diagrams

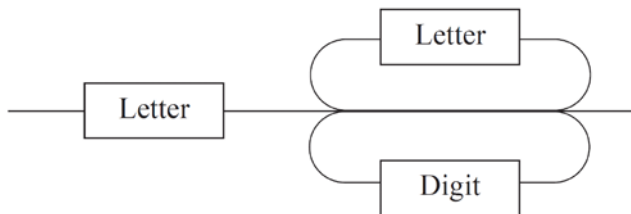
This is an alternative, graphical method used to define the syntax of a programming language.

Rectangles are used to enclose non-terminal symbols, that is, symbols that will be further defined. Circles or rounded rectangles are used to enclose terminal symbols. There is no need for further definition of such items. They may be a symbol or a reserved word.

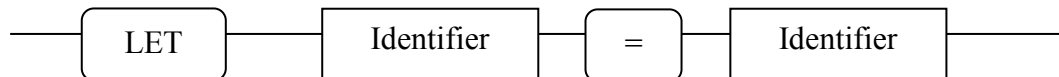
These elements are linked by paths to show all valid combinations. By starting at the left-hand side of the diagram and tracing any path in a forward direction to reach the right-hand side of the diagram, a syntactically correct construct is defined.

‘Railroad’ in this context means a branch in the diagram is valid if it is treated as a set of tracks in a railroad layout where a branch can only be followed in a forward direction.

Identifier



Assignment statement



5 Software specifications

5.1 Language specifications

The syllabus does not prescribe a single coding language for implementation of programs but advocates a range of high level languages.

Students are required to be proficient in a language which allows them to use concepts covered in the syllabus. These should include standard control structures, relevant data structures, a debugging tool and the use of a compiler and an interpreter. This allows students to practise on a regular basis a range of the concepts covered in the course.

Appropriate languages support structured programming concepts.

General language requirements

The programming language chosen should allow the students to:

- use meaningful identifiers
- use binary and multi-way selection
- use pre-test and post-test repetition and FOR / NEXT statements
- include comments (remarks) and appropriate indentation for control structures in the code to document the program
- use numeric (integer and real) and string data
- use logical and relational operators
- use record and array data types including multidimensional arrays and arrays of records
- use string handling operations to extract characters from a string
- use and create procedures (subprograms, subroutines) which may require parameters
- allow definition of functions that return a value
- input data from a sequential and a relative file
- write output to a sequential and relative file
- append records to a sequential file
- update specified records in a relative file
- include the use of a debugging facility such as single instruction stepping, trace and breakpoints.

Languages such as the following meet the requirements of this course:

- Pascal
- a structured version of BASIC
- Visual Basic
- C++
- Java
- Java Script
- Python
- ActionScripting for Flash
- Objective C.

Programming environments that generate code automatically (eg GameMaker) or webpage formatting languages (eg HTML, XML) do not meet these requirements.

Event-driven approach

The programming language(s) chosen for this approach should allow the students to:

- create scripts which make use of standard control structures
- create scripts which permit systems events such as mouse button presses and keystrokes to be handled
- implement a solution not limited to the simple linking of screens.

Languages such as the following are appropriate for this approach:

- Visual Basic
- HyperCard
- Flash
- Revolution
- Delphi
- REALBasic

Prototyping approach

The software package(s) chosen for this approach must allow the students to:

- create several linked pages, cards or screens
- create graphic and text elements on the pages, cards or screens, using drag and drop facilities
- accept input.

Software applications such as the following are appropriate for this approach:

- PowerPoint
- Visual Basic
- HyperCard
- Revolution
- Delphi
- Flash
- Access
- FileMaker Pro
- REALBasic.

Rapid Applications Development (RAD)

The software package(s) chosen for this approach should allow the students to:

- create graphic and text elements on the screens using drag and drop facilities
- accept input
- store and access data
- perform mathematical operations on the data
- perform sorting, searching and reporting operations on the data.

Software applications such as the following are appropriate for this approach:

Visual Basic, Excel, HyperCard, Revolution, Delphi, Access, FileMaker Pro, REALBasic.

Software used to simulate the fetch–execute cycle in the CPU

The software package(s) chosen should allow the students to simulate the processing of machine code instructions using animation to demonstrate the movement and processing of data in the CPU.

There are a number of excellent simulations available on the internet, such as the analytical engine www.course.com/downloads/computerscience/aeonline/6/3/index.html

5.2 Options

5.2.1 Option 1: Programming paradigms

Students should design and create simple solutions in both the logic and object oriented paradigms. The languages chosen should be representative of the paradigm and incorporate the concepts in the syllabus.

Logic paradigm

The following example uses a family database.

Some *facts* of the family database are:

female(X)	<i>meaning that X is a female.</i>
male(Y)	<i>meaning that Y is a male.</i>
parent(X, Y)	<i>meaning that X is the parent of Y.</i>

Some examples of facts for this database are:

female(karen)
female(rosemary)
female(sun yi)
female(mahdu)

male(sam)
male(steve)

parent(sam, karen)
parent(rosemary, karen)
parent(mahdu, sam)
parent(steve, sam)
parent(rosemary, sun yi)

Some *rules* for the family database are:

grandparent(X, Y) :- parent(X, Z), parent(Z, Y)

X is the grandparent of Y if X is the parent of Z and Z is the parent of Y

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X ≠ Y

X is the sibling of Y when Z is the parent of both X and Y, and X and Y are different people. (ie you cannot be your own sibling).

Example *goals* for the family database are:

grandmother(mahdu, karen) *this would evaluate to true based on the facts defined*

grandmother(mahdu, steve) *this would evaluate to false based on the facts defined*

The software used must allow students to:

- define facts
- create, edit and remove rules
- enter facts and display the solution/goal
- display the rules that the system used to reach a conclusion/arrive at a goal.

Languages such as Prolog are appropriate for this paradigm.

An example of a specific logic programming language is Strawberry Prolog. The files can be downloaded from www.dobrev.com/.

Object oriented paradigm

The examples below deal with points in a number plane in an object oriented environment. The basic *object* of the number plane is a point, with its x and y coordinates. The following code fragment declares a *class* called *Point* with its *attributes* and *methods*. Two *sub-classes* are defined to demonstrate polymorphism, where the method *getArea* is interpreted differently depending on the sub-class in which it is used.

```
class Point {
    private –
        point_no: integer
        x_coordinate: double
        y_coordinate: double
    public –
        getPoint(point_no):
            return x_coordinate and y_coordinate
}
sub-class Circle {
    is a Point
    private –
        circle_no: integer
        radius: double
    public –
        getArea(circle_no):
            return Math.PI*radius*radius
}
sub-class Rectangle {
    is a Point
    private –
        rectangle_no: integer
        height: double
        width: double
    public –
        getArea(rectangle_no):
            return height*width
}
```

The software used must allow students to:

- define classes, objects, attributes and methods
- make use of inheritance, polymorphism and encapsulation
- use control structures and variables.

Languages such as Java and C++ are appropriate for this paradigm.

An example of a specific object oriented programming language is BlueJ, which is an integrated Java environment which has been specifically designed for teachers by universities in Melbourne and overseas. The files can be downloaded from www.bluej.org/.

Past practice HSC and trial HSC papers contain examples of a wide variety of code fragments related to both programming paradigms. The answers provided often include a discussion as to the structure, content and purpose of these code fragments.

Students are expected to be able to read a set of specifications for a problem that can be solved using these paradigms, and to design or interpret an appropriate code fragment to achieve a prescribed purpose.

5.2.2 Option 2: The interrelationship between software and hardware

ASCII and Unicode

ASCII is the 7-bit binary representation of up to 128 binary combinations, in the range hexadecimal 00 to FF, which are symbols or control codes. Some characters (95) are printable and the others (33) are non-printable control characters.

The table is available on the internet at sites such as www.asciitable.com.

UTF-8 is a scheme for representing the Unicode character set and encodes characters using between one and four 8-bit bytes. The earliest proposal for Unicode was effectively an extension of ASCII to 16 bits (two bytes) capable of representing 2^{16} or 65,536 binary combinations.

Unicode is a scheme for representing a far wider range of characters than ASCII can represent. In 2012 Unicode had 1,114,112 available addresses or code points in the range hexadecimal 00 to 10FFFF. The full Unicode standards can represent characters in a wide range of languages and historical scripts.

The NULL value at 00000000 is not utilised in either ASCII or Unicode. Thus 128 opportunities for codes are reduced to 127 and the 65,536 are reduced to 65,535 available combinations.

The table is available on the internet from sites such as:

www.tamasoft.co.jp/en/general-info/unicode.html

www.unicode.org/charts/

www.unicode.org/charts/PDF/U0000.pdf

Electronic circuits

The simulation software used to build and test user-designed and speciality circuits should allow students to:

- drag and drop logic gate symbols and link them to create a circuit
- edit existing circuit designs
- print circuit designs
- simulate the working of designed circuits to show outputs for selected input values.


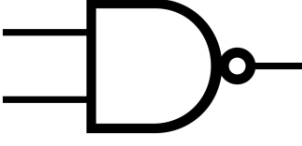
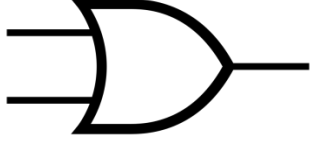
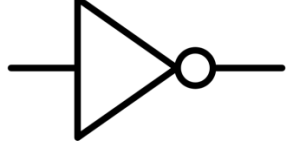
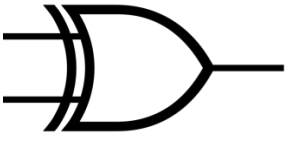
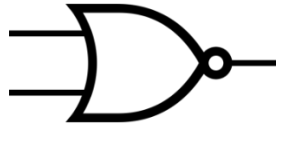
Software packages such as LogicSim, LogicCircuit, Circuit Wizard, YENKA and Crocodile Clips are appropriate for this purpose.

There are a number of appropriate interactive sites available on the internet, such as:

<http://logic.ly/demo/>

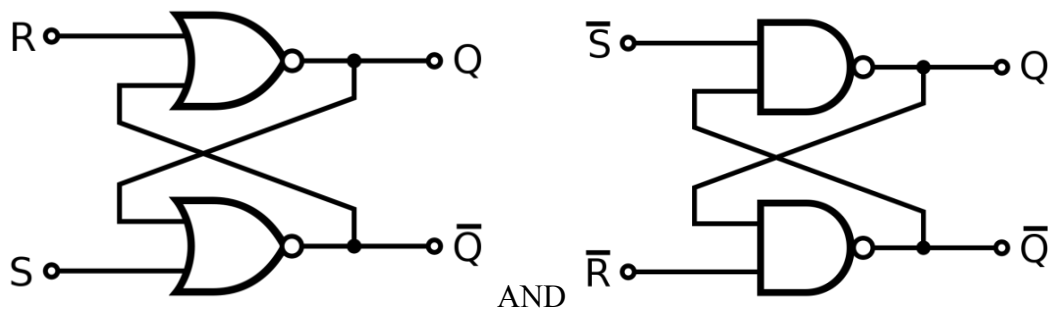
<http://courses.cs.vt.edu/~csonline/MachineArchitecture/Lessons/Gates/index.html>

Logic gates are typically represented by the following symbols:

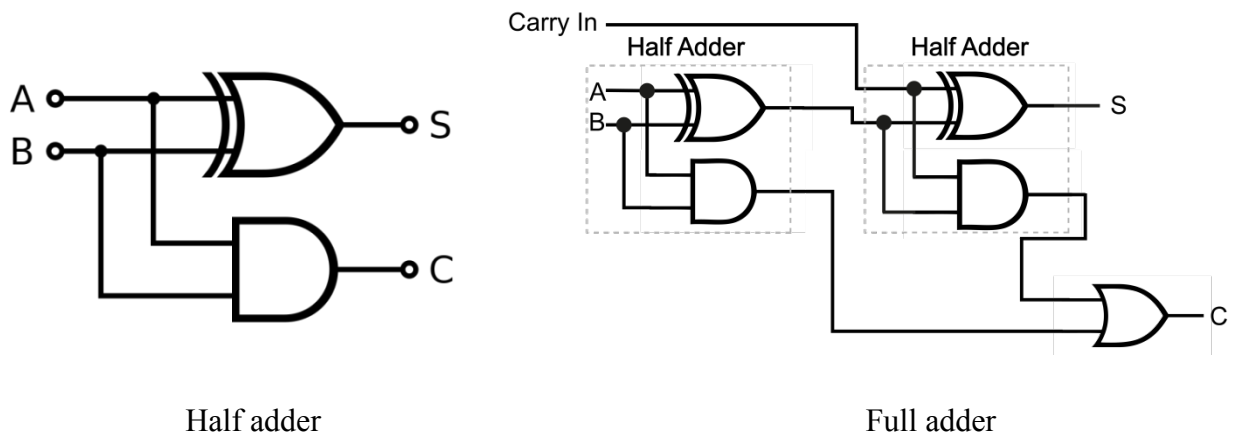
<p>AND</p> 	<p>NAND</p> 	<p>OR</p> 
<p>NOT</p> 	<p>XOR</p> 	<p>NOR</p> 

Specialty circuits, including:

Flip Flops or Latches can also be represented.



Half adder circuits and full adders can be similarly represented.



Boolean algebra for circuit design

There are a number of appropriate sites relating to the use of Boolean algebra for analysing and designing logic gates, such as:

www.play-hookey.com/

www.ee.surrey.ac.uk/Projects/Labview/boolalgebra/

www.allaboutcircuits.com/vol_4/chpt_7/5.html

www.scribd.com/doc/2176277/Boolean-Algebra

www.tutorvista.com/math/boolean-algebra-circuits

www.ibiblio.org/kuphaldt/electricCircuits/Digital/DIGI_7.html

Specialist devices with digital input and/or output

Past practice HSC and trial HSC examination papers contain examples of a wide variety of specialist devices and their relevant data streams. The answers provided often include a discussion as to the structure, content and purpose of these data streams. Questions have focused on data streams for devices such as:

- a model helicopter
- a model train/car/boat
- a USB mouse
- a plotter
- a security camera
- a cutter machine
- an automated jar-filling line with valves and conveyor belt
- a scrolling display system for 1 line text display.

Students are expected to be able to read a set of specifications for an unfamiliar device and to design or interpret an appropriate data stream to achieve a prescribed purpose.

6 Methods of algorithm description

6.1 Introduction

This document presents two methods for describing algorithms for use in the implementation of the Software Design and Development course.

In assessing the quality of algorithm descriptions, general criteria such as the correctness of the algorithm, the clarity of the description, the use of appropriate control structures and the embodiment of structured methods should be taken into consideration.

The document presents standards that students should aim for in publishing solutions to problems. The same standards should be used by teachers when presenting algorithms to students. In many cases there are alternatives that could be used and it should be noted that students can expect to see different methods of algorithm description in books and magazines. A preferred solution is one which is easy to read and interpret (and so is easy to maintain), is elegant and uses standard control structures with a top-down approach for a more complex solution, and includes a clear, uncluttered mainline and a separate subroutine for each logical task.

An algorithm is a step-by-step procedure for solving a problem; programming languages are essentially a way of expressing algorithms.

6.2 Overview of two methods

It is expected that students are able to develop and interpret algorithms using pseudocode and flowcharts.

Pseudocode

Pseudocode uses English-like statements with defined rules of structure and keywords.

Pseudocode guidelines

The pseudocode keywords are:

for each procedure or subroutine

```
BEGIN name  
END name
```

for binary selection

```
IF condition THEN  
    statements  
ELSE  
    statements  
ENDIF
```

for multi-way selection

```
CASEWHERE expression evaluates to  
    A: process A  
    B: process B  
    .....  
    OTHERWISE: process ...  
ENDCASE
```

for pre-test repetition

```
WHILE condition  
    statements  
ENDWHILE
```

for post-test repetition

```
REPEAT  
    statements  
UNTIL condition
```

For FOR / NEXT loops

```
FOR variable = start TO finish STEP increment  
    statements  
NEXT variable
```

In pseudocode:

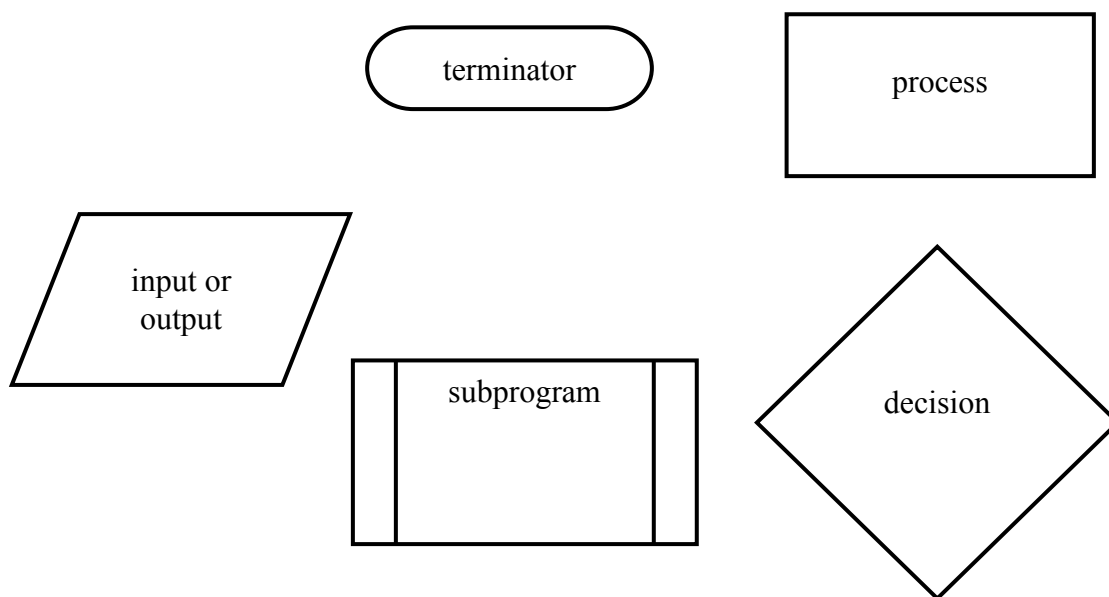
- keywords are written in capitals
- structural elements come in pairs, eg for every BEGIN there is an END, for every IF there is an ENDIF.
- indenting is used to identify control structures in the algorithm
- the names of subprograms are underlined. This means that when refining the solution to a problem, a subroutine can be referred to in an algorithm by underlining its name, and a separate subprogram developed to show the logic of that routine. This feature enables the use of the top-down development concept, where details for a particular process need only be considered within the relevant subroutine.

Flowcharts

Flowcharts are a diagrammatic method representing algorithms, which are read from top to bottom and left to right.

Flowchart elements

Flowcharts use the following symbols connected by lines with arrowheads to indicate the flow. It is common practice to show arrowheads to avoid ambiguity.



Flowcharts using these symbols should be developed using only the standard control structures (described on the following pages).

It is important to start any complex algorithm with a clear, uncluttered main line. This should reference the required subroutines, whose detail is shown in separate flowcharts.

A subroutine should rarely require more than one page, if it correctly makes use of further subroutines for detailed logic.

6.3 Programming structures

Control structures

Algorithms are developed using the basic control structures of *sequence*, *selection*, *repetition* and *subprograms*. A description of each of these structures, together with examples of their use, follows.

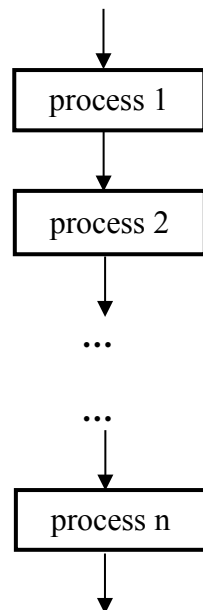
Sequence

In a computer program or an algorithm, sequence involves simple steps which are to be executed one after the other. The steps are executed in the same order in which they are written.

Pseudocode

```
process 1  
process 2  
...  
...  
process n
```

Flowchart



The arrowheads are optional if the flow is top-to-bottom.

Example using sequence

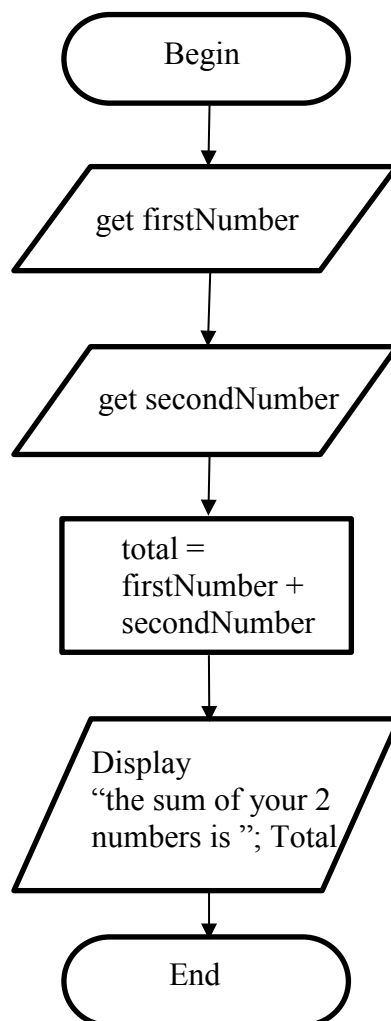
Problem

Write a set of instructions to add two numbers and display the answer.

Pseudocode

```
BEGIN Add2Numbers
  get firstNumber
  get secondNumber
  total = firstNumber + secondNumber
  Display "the sum of your 2 numbers is "; total
END Add2Numbers
```

Flowchart



Selection

Selection is used in a computer program or algorithm to determine which particular step or set of steps is to be executed. This is also referred to as a 'decision'. A selection statement can be used to choose a specific path dependent on a condition. There are two types of selection: binary selection (two possible pathways) and multi-way selection (many possible pathways).

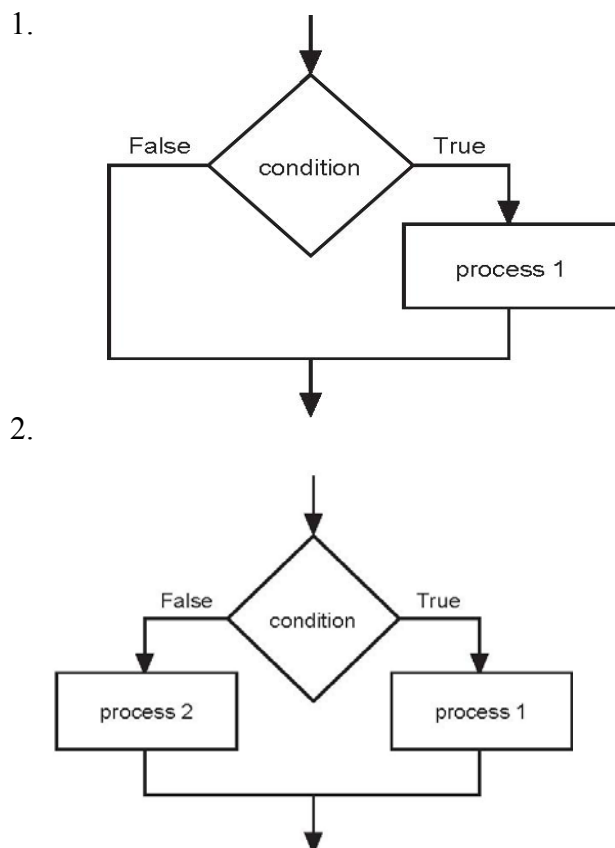
Binary selection

In binary selection, if the condition is met then one path is taken, otherwise the second possible path is followed. In each of the examples below, the first case described requires a process to be completed only if the condition is true. The process is ignored if the condition is false. In the second case, there is an alternative process if the condition is false.

Pseudocode

1. IF condition THEN
 process 1
ENDIF
2. IF condition THEN
 process 1
ELSE
 process 2
ENDIF

Flowchart



Note: in a flowchart it is most important to label the arrows coming from the decision diamond, to remove any ambiguity.

Example using binary selection

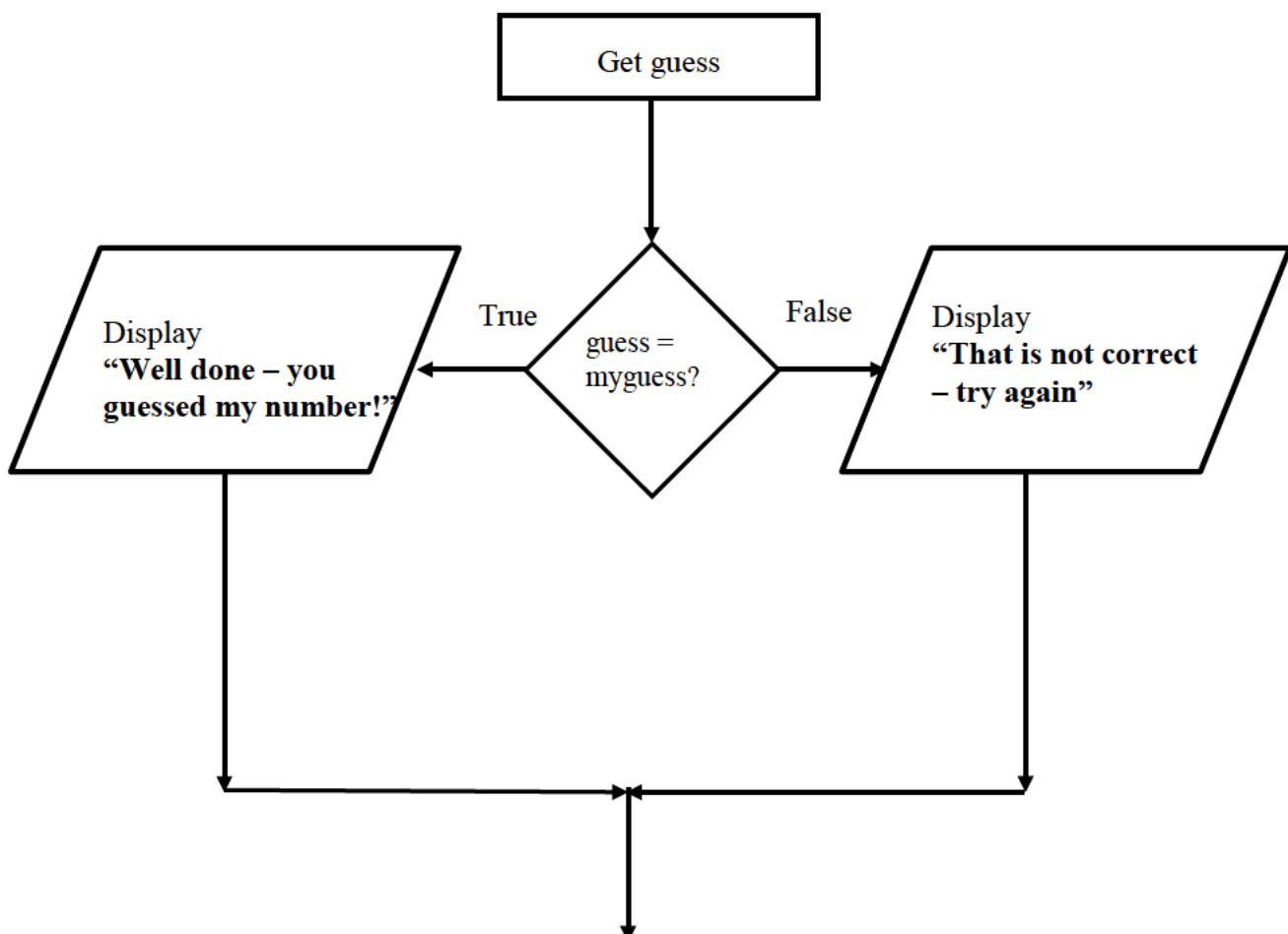
Problem

Determine the message to be displayed for a guess the number game.

Pseudocode

```
Get guess
IF guess = myguess THEN
    Display "Well done – you guessed my number!"
ELSE
    Display "That is not correct – try again"
END IF
```

Flowchart



Multi-way selection

Multi-way selection allows for any number of possible choices, or cases. The path taken is determined by the evaluation of the expression. Multi-way selection is often referred to as a case structure.

Pseudocode

CASEWHERE expression evaluates to

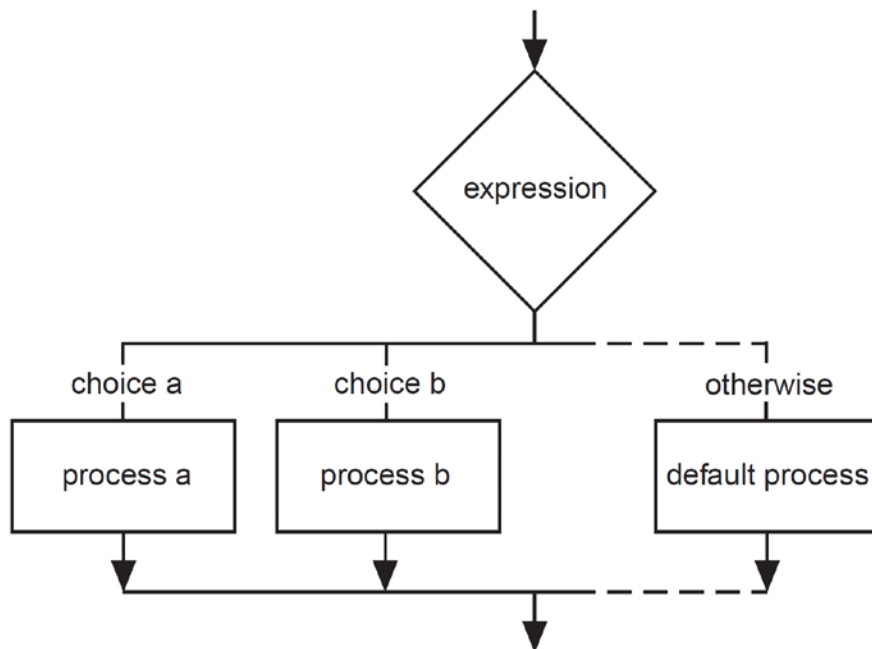
choice a : process a

choice b : process b

OTHERWISE : default process

ENDCASE

Flowchart



Note: As the flowchart version of the multi-way selection indicates, **only one** process on each pass is executed as a result of the implementation of the multi-way selection.

Example using multi-way selection

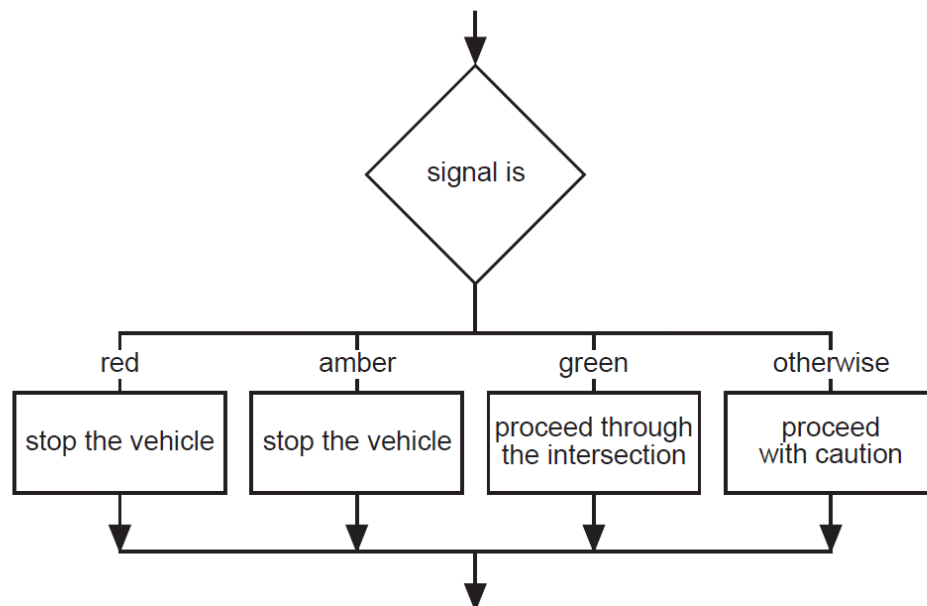
Problem

Write a set of instructions that describes how to respond to all possible signals at a set of traffic control lights.

Pseudocode

```
CASEWHERE signal is  
    red : stop the vehicle  
    amber : stop the vehicle  
    green : proceed through the intersection  
    OTHERWISE : proceed with caution  
ENDCASE
```

Flowchart



Repetition

Repetition allows for a portion of an algorithm or computer program to be executed any number of times dependent on some condition being met. An occurrence of repetition is usually known as a loop.

An essential feature of repetition is that each loop has a termination condition to stop the repetition, or the obvious outcome is that the loop never completes execution. This is known as an *infinite loop* and is obviously undesirable. The termination condition can be checked or tested at the beginning or end of the loop, and is known as a pre-test or post-test respectively. Following is a description of each of these types of loop.

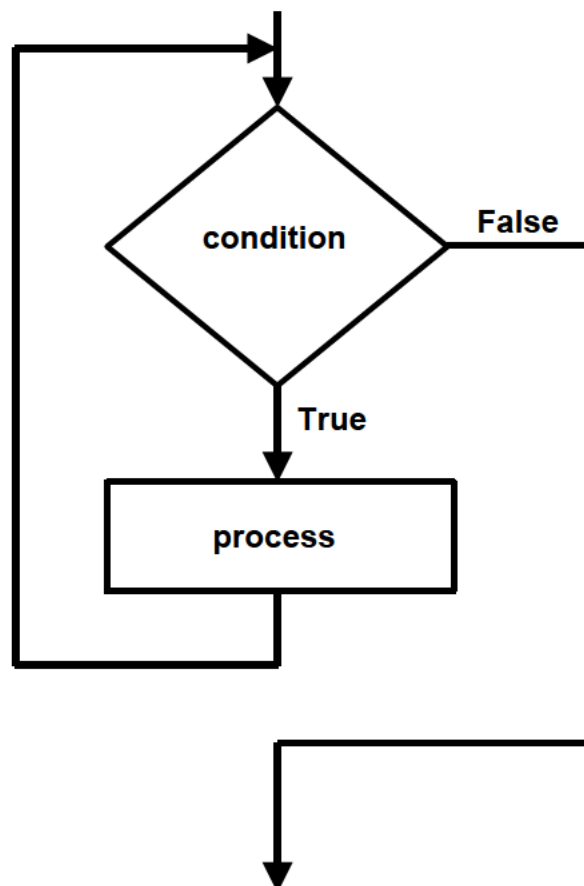
Repetition: pre-test

A pre-tested loop is so named because the condition has to be met at the very beginning of the loop or the body of the loop is not executed. This construct is often called a *guarded loop*. The body of the loop is executed repeatedly while the termination condition is true.

Pseudocode

```
WHILE condition is true  
    process(es)  
ENDWHILE
```

Flowchart



Example using pre-test repetition

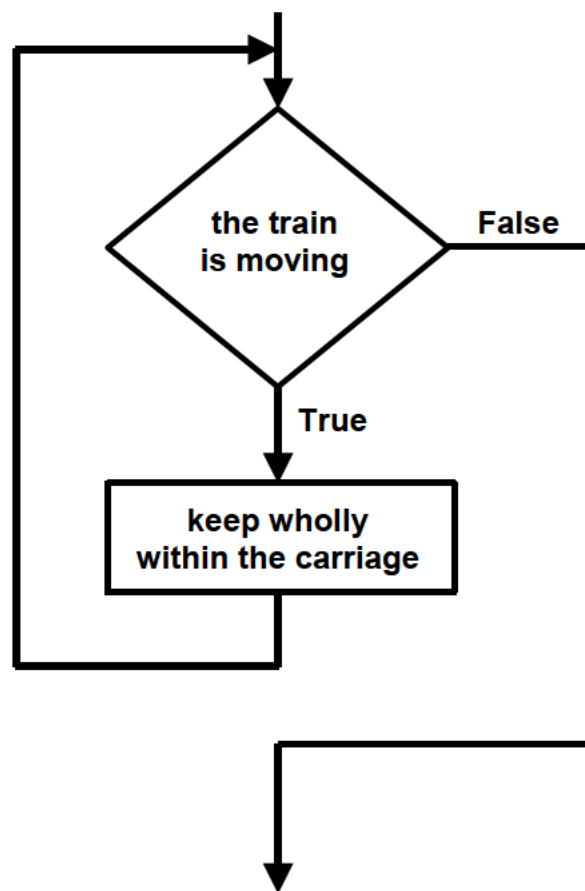
Problem

Determine a safety procedure for travelling in a carriage on a moving train.

Pseudocode

```
WHILE the train is moving  
    keep wholly within the carriage  
ENDWHILE
```

Flowchart

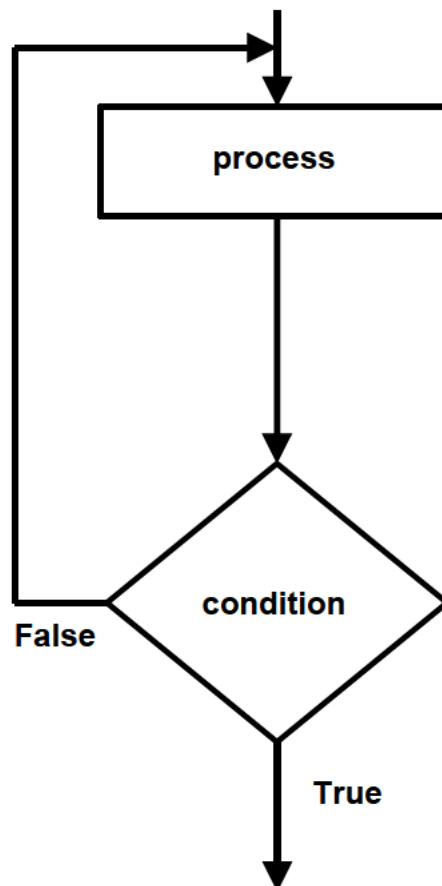


Repetition: post-test

A post-tested loop executes the body of the loop before testing the termination condition. This construct is often referred to as an *unguarded loop*. The body of the loop is repeatedly executed until the termination condition is true. An important difference between a pre-test and post-test loop is that the statements of a post-test loop are executed at least once even if the condition is originally true, whereas the body of the pre-test loop may never be executed if the termination condition is originally true. A close look at the representations of the two loop types makes this point apparent.

Pseudocode

```
REPEAT  
    process  
UNTIL condition is true
```

Flowchart

Example using post-test repetition

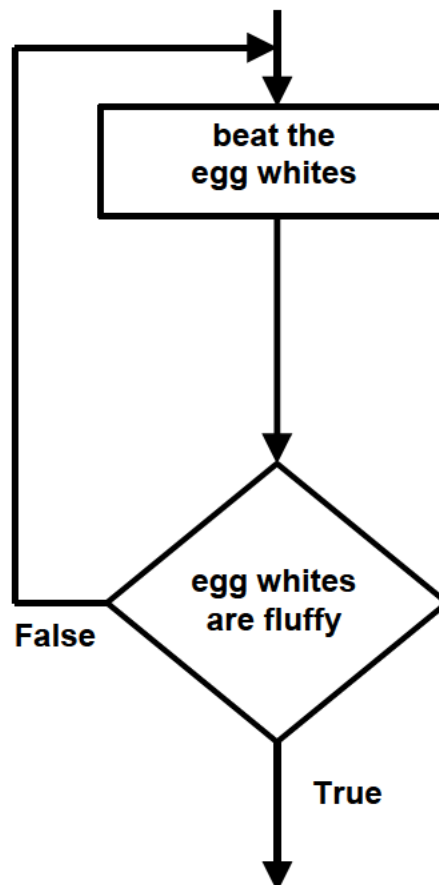
Problem

Determine a procedure to beat egg whites until fluffy.

Pseudocode

```
REPEAT  
    beat the egg whites  
UNTIL fluffy
```

Flowchart



Repetition: FOR / NEXT or counted loop

Counted loops or FOR / NEXT loops can be regarded as special cases of repetition and, depending on the language in which they are implemented, are implemented as either pre-test or post-test repetitions.

In pseudocode, a counted loop is expressed as:

Pseudocode

```
FOR variable = start TO finish STEP increment
    statements
NEXT variable
```

Note that increments can take either a positive or negative value

Flowchart

To represent a FOR / NEXT loop in flowchart format, it is necessary to describe the required logic as the equivalent pre- or post-test:

Example using a FOR / NEXT loop

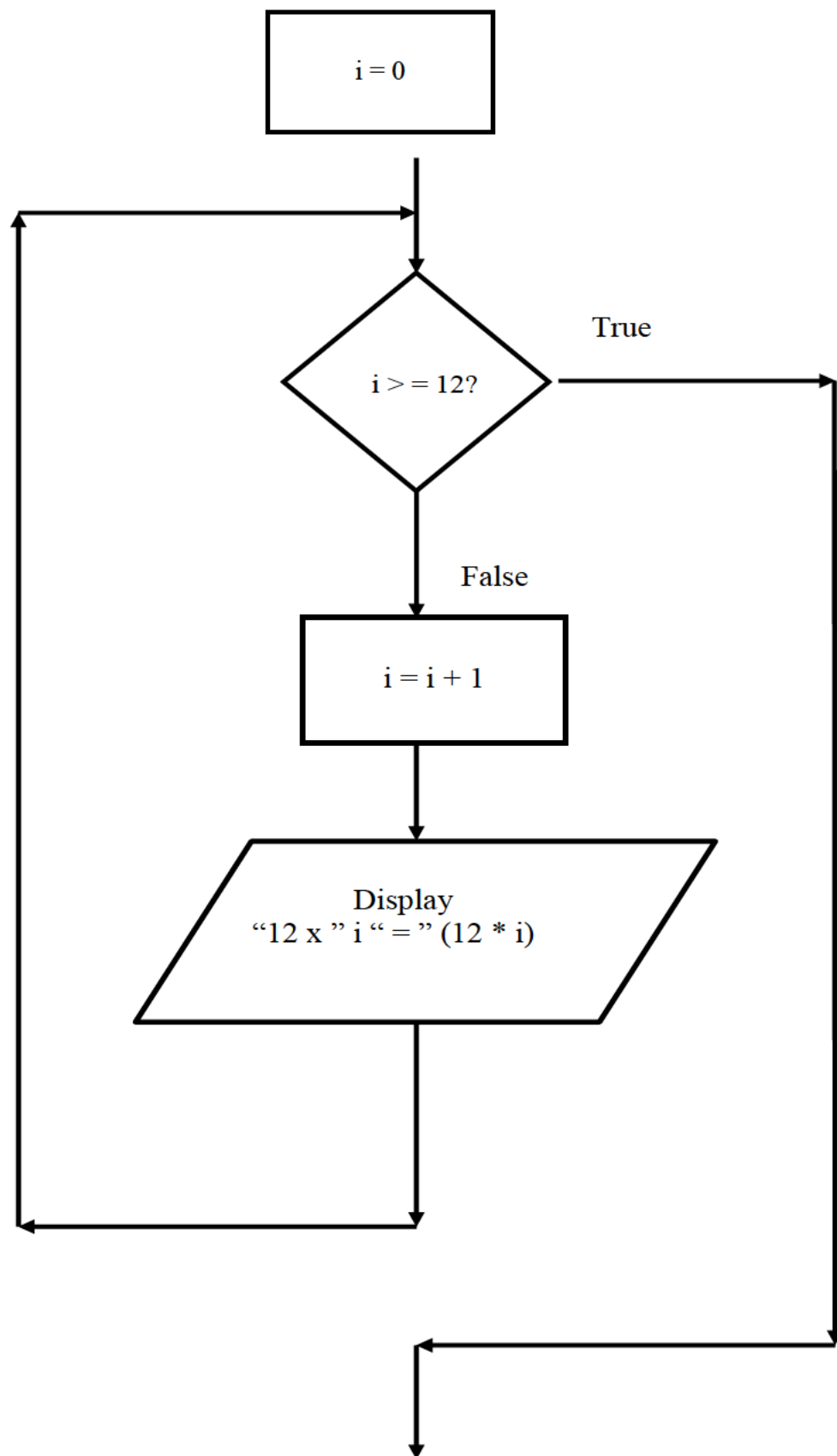
Problem

Develop a procedure to print out the 12 times table.

Pseudocode

```
FOR i = 1 to 12 STEP 1
    Display "12 x " i " = " (12 * i)
NEXT i
```

Flowchart



Subprograms

Subprograms, as the name implies, are complete part-programs that are used from within the main program section. They use refinement develop solutions to problems that are easy to follow. Sections of the solution are developed and presented in understandable chunks, and because of this, subprograms are particularly useful when using the top-down method of solution development.

When using subprograms it is important that the solution expression indicates where the main program branches to a subprogram. It is equally important to indicate exactly where the subprogram begins. In pseudocode, the statement in the main program that is expanded in a subprogram is underlined to indicate that further explanation follows. The expanded subprogram section should be identified by using the keyword BEGIN followed by the underlined title used in the main program. The end of the subprogram is marked by the keyword END and the underlined title used in the main program.

When using flowcharts, a subprogram is shown by an additional vertical line on each side of the process box. This indicates that the subprogram is expanded elsewhere. The start and end of the subprogram flowchart uses the name of the subprogram in the termination boxes.

Pseudocode

BEGIN MAINPROGRAM

 process 1

process 2

 process 3

 process 4

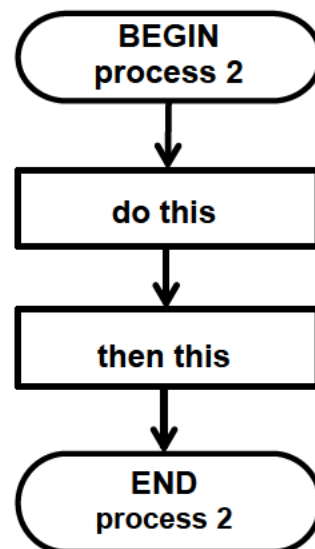
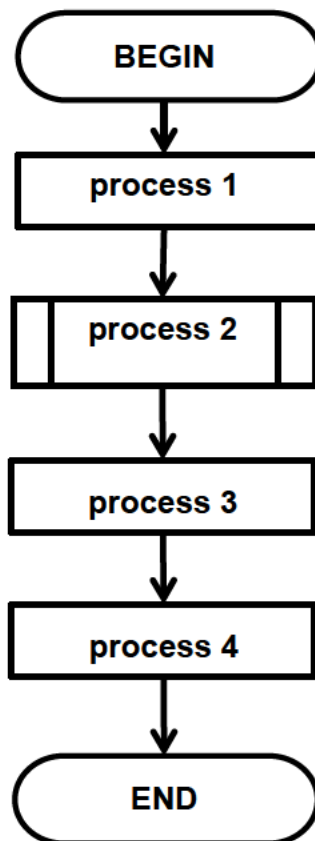
END MAINPROGRAM

BEGIN process 2

 do this

 do that

END process 2

Flowcharts

In many cases a subprogram can be written to do the same task at two or more points in an algorithm. Each time the subprogram is called, it may operate on different data. To indicate the data to be processed, one or more parameters are used. The parameters allow the author to write a general algorithm using the formal parameters. When the subprogram is executed, the algorithm carries out its task on the actual parameters given at the call. The parameters to be used by a subprogram are provided as a list in parentheses after the name of the subprogram. There is no need to include them at the end of the algorithm.

Example using a subprogram with one parameter

Problem

Determine the logic required to fill an array with the characters comprising a name, and then use the same logic to fill a second array with characters comprising an address.

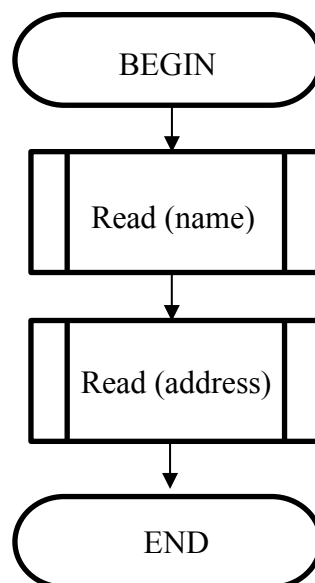
Pseudocode

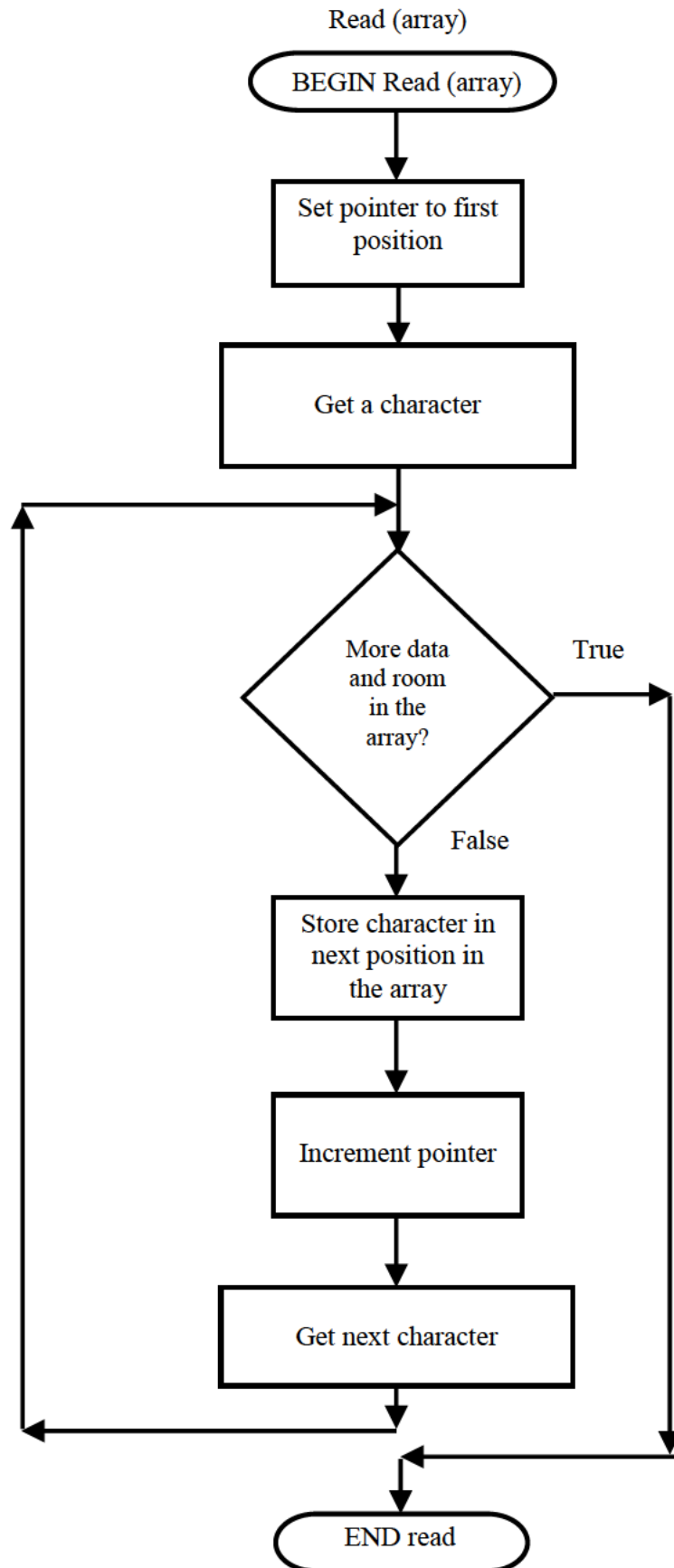
```
BEGIN MAINPROGRAM  
    read (name)  
    read (address)  
END MAINPROGRAM
```

```
BEGIN read (array)  
    Set pointer to first position  
    Get a character  
    WHILE there is still more data AND there is room in the array  
        Store data in the array at the position given by the pointer  
        Increment the pointer  
        Get next character  
    ENDWHILE  
END read
```

The first time that this subprogram is called, the characters are read into the array called 'name'.
The second time, the data characters are read into the array called 'address'.

Flowcharts





Passing values back from a subroutine

The method for passing values back from a subroutine depends on the implementation used in the particular language.

In general terms, some languages allow a number of parameters to be passed, some of which are used to pass values in to the subroutine, while others are used to pass generated values back to the calling routine.

Example using multiple parameters

Problem

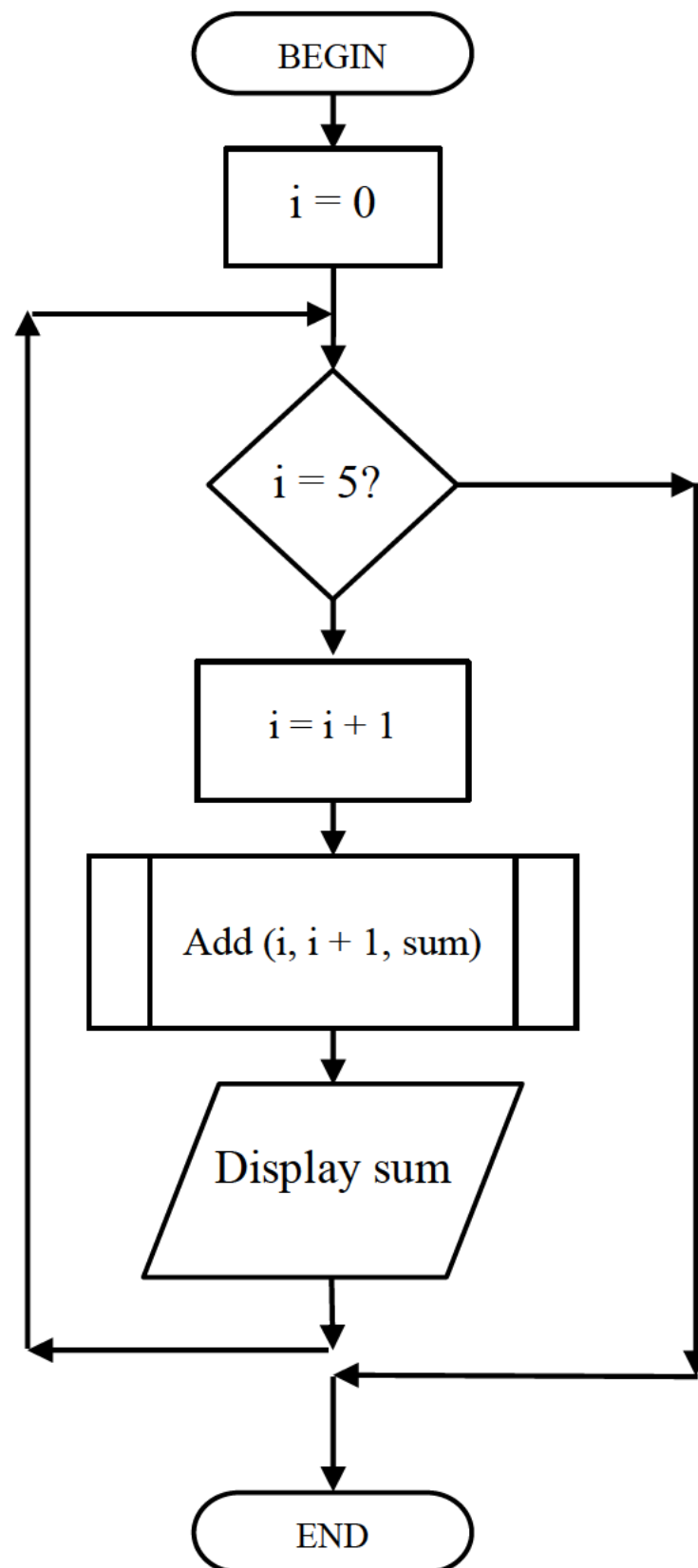
Determine the logic required to display the sum of 2 consecutive integers, where the smaller integer takes all possible values from 1 to 5.

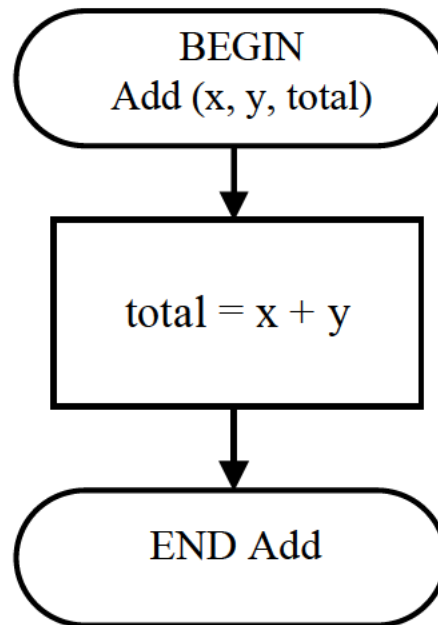
Pseudocode

```
BEGIN MAINPROGRAM
    FOR I = 1 to 5
        Add (i, i + 1, sum)
        Display sum
    NEXT i
END MAINPROGRAM

BEGIN Add (x, y, total)
    total = x + y
END Add
```

Flowcharts





Example using RETURN to pass a value back

Other languages require that if a value is to be passed back, it should be treated as if it were generated from a function call, using the verb RETURN to send the appropriate value back to the calling routine.

Problem

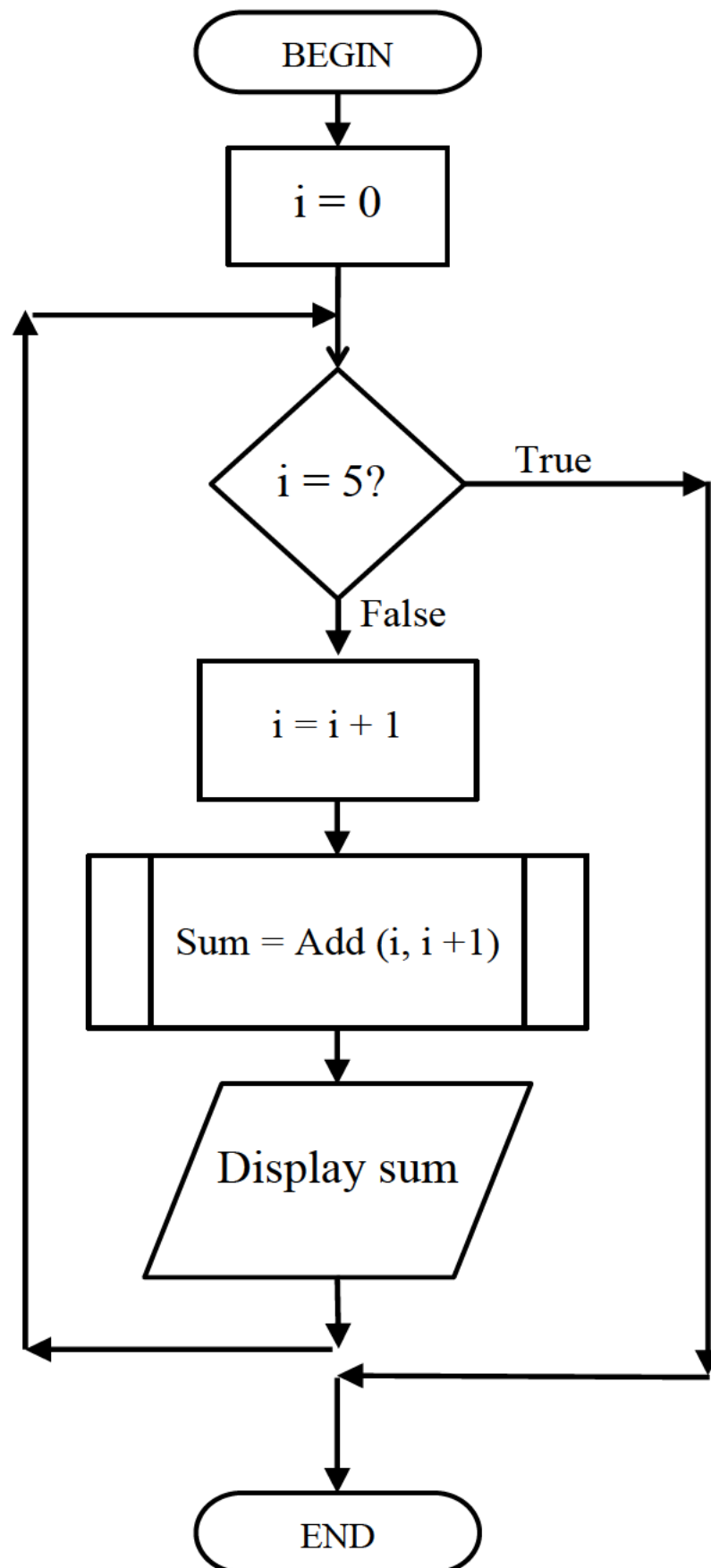
Determine the logic required to display the sum of 2 consecutive integers where the smaller integer takes all possible values from 1 to 5.

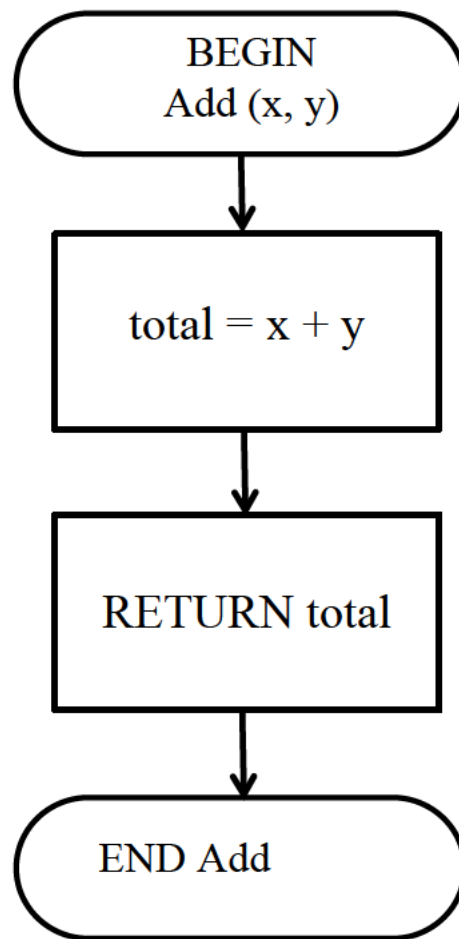
Pseudocode

```
BEGIN MAINPROGRAM
    FOR i = 1 to 5
        sum = Add (i, i + 1)
        Display sum
    NEXT i
END MAINPROGRAM
```

```
BEGIN Add (x, y)
    total = x + y
END Add
```

Flowcharts





7 Standard algorithms

Each of these algorithms should be desk checked using appropriate values to ensure that the logic is understood. It is also important for students to implement these routines into their code so they can see the resulting output and understand the processing that has occurred.

Elements in an array can be indexed from 0 or 1. For consistency, in all the following examples, processing starts with element 1.

Load an array and print its contents

An array can be loaded from data values or input from the keyboard. The following algorithm assumes that values are read from a list of data statements until a sentinel value of “xxx” is encountered.

```
BEGIN LoadArray
    Let i = 1
    Read DataValue
    WHILE DataValue <> “xxx”
        Let Element (i) = DataValue
        i = i + 1
        Read DataValue
    ENDWHILE
    Let NumElements = i
    Display “ There are” NumElements “ items loaded into the array”
END LoadArray
```

Note the use of a priming read to ensure that the sentinel value is not loaded into the array. The pre-test loop ensures that if there is no data in the list of data to be loaded (other than the sentinel value), then the loop will never be entered.

To print the array, it is assumed that there is a variable that stores the number of elements in the array.

```
BEGIN PrintArrayContents
    Let i = 1
    REPEAT
        Display Element (i)
        i = i + 1
    UNTIL i >= NumElements
END PrintArrayContents
```

Note that if the number of elements is not known, the sentinel value would be stored into the last element in the array, and printing would continue until the sentinel value of “xxx” is encountered.

Add the contents of an array of numbers

To add the contents of an array of numbers, it is assumed that there is a counter indicating the number of elements in the array.

```
BEGIN SumArrayContents
  Let i = 1
  Let total = 0
  REPEAT
    Let total = Element (i) + total
    i = i + 1
  UNTIL i > NumElements
  Display "The sum of all of the elements in the array = " total
END SumArrayContents
```

Note that if the number of elements is unknown, the sentinel value would be stored into the last element in the array, and the total would continue to accumulate until the sentinel value is encountered. In the example below, a sentinel of 999 is used.

Note the use of the Pre-test loop to ensure that the sentinel value 999 is not included in the total.

```
BEGIN SumArrayContents
  Let i = 1
  Let total = 0
  WHILE Element (i) <> 999
    Let total = Element (i) + total
    i = i + 1
  END WHILE
  Display "There are " i " elements"
  Display "The sum of all of the elements in the array = " total
END SumArrayContents
```

Finding a maximum value in an array

The value in the first element is stored in a temporary variable called Max. Each element is then considered in turn to determine if its value is larger than that stored value. If so, the value in Max is replaced by this larger value, and the index of this element is stored in a temporary variable called MaxIndex.

When all elements have been considered, Max will contain the largest value, and MaxIndex will contain the index of the largest element.

```
BEGIN FindMAX
  Let Max = Element (1)
  Let MaxIndex = 1
  Let i = 2
  REPEAT
    IF Element (i) > Max THEN
      Let Max = Element (i)
      Let MaxIndex = i
    END IF
    Let i = i + 1
  UNTIL i > NumElementsInArray
  Display "The highest value is " Max " at position " MaxIndex
END FindMAX
```

Finding a minimum value in an array

The value in the first element is stored in a temporary variable called Min. Each element is then considered in turn to determine if its value is smaller than this stored value. If so, the value in Min is replaced by this smaller value, and the index of this element is stored in a temporary variable called MinIndex.

When all elements have been considered, Min will contain the smallest value, and MinIndex will contain the index of the smallest element.

```
BEGIN FindMIN
  Let Min = Element (1)
  Let MinIndex = 1
  Let i = 2
  REPEAT
    IF Element (i) < Min THEN
      Let Min = Element (i)
      Let MinIndex = i
    END IF
    Let i = i + 1
  UNTIL i > NumElementsinArray
  Display "The smallest value is " Min " at position " MinIndex
END FindMIN
```

Processing strings

Extracting data from a string

Most languages offer the facility of extracting data from strings. Depending on the language used, the verbs can differ. Examples of typical verbs are `mid$` or `instr$`

In the algorithms that follow, a general statement is used:
extract from the i^{th} character (for n characters) from String into ExtractedString.

To extract the days and months from a date in the format DD/MM/YY the following algorithm is relevant:

Note: We need to pull out the first 2 characters and place them in Days, and the 4th and 5th characters and place them into Month.

```
BEGIN FindDaysandMonths
    Get DateString
    Let StartDays = 1
    Let StartMonths = 4
    'the fourth character in the date string is the start of the month value
    Extract from the StartDaysth character (for 2 characters) from DateString into Days
    Extract from the StartMonthsth character (for 2 characters) from DateString into Month
    Display"the month is " Month " and the day of the month is " Days
END FindDaysandMonths
```

Insertion of a string into another string

To insert a string into a second string, the best way is to concatenate the required strings together, as not all languages support the insertion of a string into second string with a single command.

The following example looks for the first occurrence of the delimiter “ ; ” in a string. It splits the string into two parts – the characters preceding the delimiter, and the characters following the delimiter. It then effectively inserts an entered word at that position in the existing string by joining all three parts together:

```
BEGIN InsertNewWordintoString
  Get InitialString, NewWord
  Let L = Length of InitialString
  Let Found = 0
  Let i = 1
  REPEAT
    extract from the ith character from InitialString into CheckLetter
    IF CheckLetter = “ ; ” THEN
      extract from the 1st character (for i – 1 characters) from InitialString into
      FirstPart

      extract from the (i + 1)th character (for L – i characters) from InitialString
      into SecondPart

      Let NewString = FirstPart + NewWord + SecondPart
      ‘note that if we use the + operator with strings, the values are concatenated

      found = 1

    END IF
    i = i + 1
  UNTIL i >= L OR found = 1
  IF found = 0 THEN
    Display “the delimiter ; could not be found in your string”
  ELSE
    Display “The new string is ” NewString
  ENDIF
END InsertNewWordintoString
```

Deletion of a string from another string

To delete a string from a second string, the best way is to concatenate two partial strings together.

The following example looks for the first occurrence of an entered word in a string, and deletes that word by effectively joining the first and last parts of the original string together:

```
BEGIN DeleteWordFromString
  Get InitialString, StringToGo
  Let LString = Length of InitialString
  Let Lword = Length of StringToGo
  Let found = 0
  Let i = 0
  REPEAT
    extract from the ith character (for Lword letters) from InitialString into
    CheckforWord
    IF CheckforWord = StringToGo THEN
      extract from the 1st character (for i – 1 characters) from InitialString
      into FirstPart
      extract from the (i + Lword)th character (for (LString – Lword – i + 1)
      characters) from InitialString into SecondPart
      Let NewString = FirstPart + SecondPart
      found = 1
    END IF
    i = i + 1
  UNTIL i >= LString OR found = 1
  IF found = 0 THEN
    Display "The word could not be found in your string"
  ELSE
    Display "The new string is " NewString
  ENDIF
END DeleteWordFromString
```

Generating a set of unique random numbers

To generate and display a set of unique random numbers, use an array of flags to indicate if a number has already been selected. If a number has already been chosen (indicated by the flag for that number being set to 1), another random number until we get to one which has not yet been selected.

This logic is shown in the algorithm below:

The aim is to generate a set of 6 unique lotto numbers from a possible set of 100 numbers, ranging from 1 to 99.

The following generic statement is used to generate a random number between two values (HighValue and LowValue):

Let $R = \text{Random}(\text{HighValue} - \text{LowValue}) + \text{LowValue}$

BEGIN PrintSixUniqueLottoNumbers

FOR i = 1 TO 99 ' set all flag values initially to zero

Let $\text{Flag}(i) = 0$

NEXT i

FOR i = 1 to 6 'do this 6 times to generate 6 numbers

REPEAT

Let $r = \text{Random}(98) + 1$

UNTIL Flag (r) = 0

'keep generating a random number until one is found which has not been used before

Let $\text{Flag}(r) = 1$

'set the flag for this number to show that it has now been used

Display “Your next number is ” r

NEXT i

END PrintSixUniqueLottoNumbers

Processing sequential files

Creating a file

Sequential files need to be opened for output when they are created and must be closed before the program ends.

A record containing sentinel values should be written as the last record in the file before it is closed, so that other programs using that same file do not have to know how many records there are in that file.

Note: It is also possible to use the system flag EOF which is used to detect the end of a sequential file in many languages.

Every language has its own particular syntax, but in an algorithm it is appropriate to include generic statements as shown in the following algorithm.

The following example writes 10 records to a sequential file called FriendsData. The data is entered by the user from the keyboard:

```
BEGIN CreateAFile
    Open FriendsData for output

    FOR i = 1 to 10
        Display "Please enter the details for the next person: "
        Get fname, sname, emailaddr, mobile
        Write FriendsData from fname, sname, emailaddr, mobile
    NEXT i
    Let fname = "xxx"
    Let sname = "xxx"
    Write FriendsData from fname, sname, emailaddr, mobile
    Close FriendsData
END CreateAFile
```

Printing the contents of a file

Sequential files that already exist need to be opened for input before they can be read, and closed before the program ends.

It is best to use a priming read as in the following algorithm, in case the located file contains no records in it. It also ensures that the sentinel values do not get processed or printed.

```
BEGIN DisplayFileContents
    Open FriendsData for input
    Read fname, sname, emailaddr, mobile from FriendsData
    'This is a priming read, performed just before entering the loop to provide the first record (if
    there is one) for printing

    WHILE fname <> "xxx" AND sname <> "xxx"
        Display fname, sname, emailaddr, mobile
        Read fname, sname, emailaddr, mobile from FriendsData
        'this reads subsequent records which can then be tested for the sentinel value before
        they are processed
    END WHILE

    Close FriendsData
END DisplayFileContents
```

Appending records to an existing sequential file

To enable new records to be added at the end of an existing sequential file, it must be opened so that records can be appended and closed before the program ends. To open it for this purpose, use the format

Open filename for append.

This algorithm asks for input from the keyboard for a series of new records to be added to an existing file.

Note: Where records are to be appended to an existing file, that file should not already include a record with sentinel values as its last record. If it does, that record will remain in the file immediately preceding the newly appended records.

```
BEGIN AddNewRecords
    Open FriendsData for append
    Display "Please enter the details for the first new person to be added: "
    Display "Enter xxx for both first name and surname to indicate there are no more records to
    be added."

    Get fname, sname, emailaddr, mobile
    WHILE fname <> "xxx" AND sname <> "xxx"
        Write FriendsData from fname, sname, emailaddr, mobile
        Display "Please enter the details for the next new person to be added:"
        Display "Enter xxx for both first name and surname to indicate there are no more
        records to be added"

        Get fname, sname, emailaddr, mobile

    END WHILE
    Close FriendsData
END AddNewRecords
```

Processing relative files

Creating a relative file

Relative files need to be opened for relative access when they are created and must be closed before the program ends. All records are accessed through the use of a key which specifies the relative position of that record within the file. The key field used must contain positive integer values only. There is no sentinel value written as the file is not accessed sequentially.

The following example writes 10 records to a relative file called ProductData. The data is entered by the user from the keyboard and the products are entered in no particular order:

```
BEGIN CreateARelativeFile
    Open ProductData for relative access
    FOR i = 1 to 10
        Display "Please enter the details for the next product: "
        Get ProdNumber, description, quantity, price
        Write ProductData from ProdNumber, description, quantity, price using
        ProdNumber

        'note the use of the variable ProdNumber as the key field, specifying where this
        record will be written in the file.
    NEXT i

    Close ProductData
END CreateARelativeFile
```

Reading records from a relative file

Relative files need to be opened for relative access when they are accessed for either input or output, and must be closed before the program ends. All records are accessed through the use of a key which specifies the relative position of that record within the file. The key field used must contain positive integer values only.

```
BEGIN ReadRecordsFromARelativeFile
  Open ProductData for relative access
  REPEAT
    Display "Please enter the product number for the next product you wish to see:"

    Display "Please enter 999 when you are done"
    Get RequiredProdNumber
    Read ProductData into ProdNumber, description, quantity, price using
    RequiredProdNumber

    'note the use of the variable RequiredProdNumber as the key field, specifying where
    this record will be found in the file

    IF RecordNotFound THEN
      'note the use of the flag RecordNotFound returned by the operating system
      Display "Sorry – no such product"
    ELSE
      Display ProdNumber, description, quantity, price

    END IF
  UNTIL RequiredProdNumber = 999
  Close ProductData
END ReadRecordsFromARelativeFile
```

Updating records in a relative file

Relative files need to be opened for relative access when they are updated and must be closed before the program ends. All records are accessed through the use of a key which specifies the relative position of that record within the file. The key field used must contain positive integer values only.

The following algorithm allows the price of any product to be changed.

```
BEGIN UpdateRecordsInARelativeFile
    Open ProductData for relative access
    Display "Please enter the product number for the next product whose price you wish to
    update:"
    Display "Please enter 999 when you are done"
    Get RequiredProdNumber
    'priming read in case they wish to exit immediately by entering 999
    WHILE RequiredProdNumber < > 999
        NotFound = 0
        Read ProductData into ProdNumber, description, quantity, price using
        RequiredProdNumber
        'note the use of the variable RequiredProdNumber as the key field, specifying where
        this record will be found in the file

        IF RecordNotFound THEN
            'note the use of the flag RecordNotFound returned by the operating system

            Display "Sorry – no such product"
            NotFound = 1
        ELSE
            Display ProdNumber, description, quantity, price
            Display " Is this the correct product?"
            Get Reply
        END IF
        'do not update until the correct product record is retrieved
        IF NotFound = 0 AND Reply = "Y" THEN
            Get NewPrice
            Write ProductData from ProdNumber, description, quantity, NewPrice using
            ProdNumber

            'update record using data for the new price and the existing data in the other
            fields
        END IF
        Display "Please enter the product number for the next product whose price you wish
        to update:"

        Display "Please enter 999 when you are done"
        Get RequiredProdNumber
    ENDWHILE
    Close ProductData
END UpdateRecordsInARelativeFile
```

8 Advanced data structures

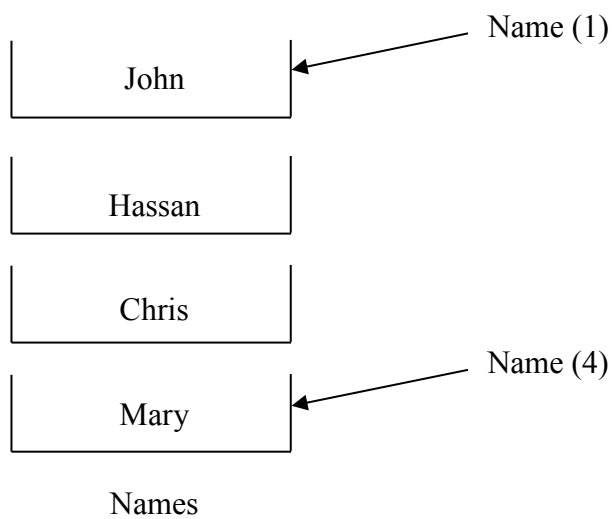
Single dimensional arrays

A simple array is a grouped list of similar variables, all of which are used for the same purpose and are of the same data type.

Individual elements are accessed using an index, which is a simple integer variable.

Before using an array in a program, some languages first require that it must be dimensioned (that is, defined) in order to allocate sufficient memory for the specified number of elements. A statement such as the following is a typical example of the required code.

DIM Names (20) as string



Multidimensional arrays

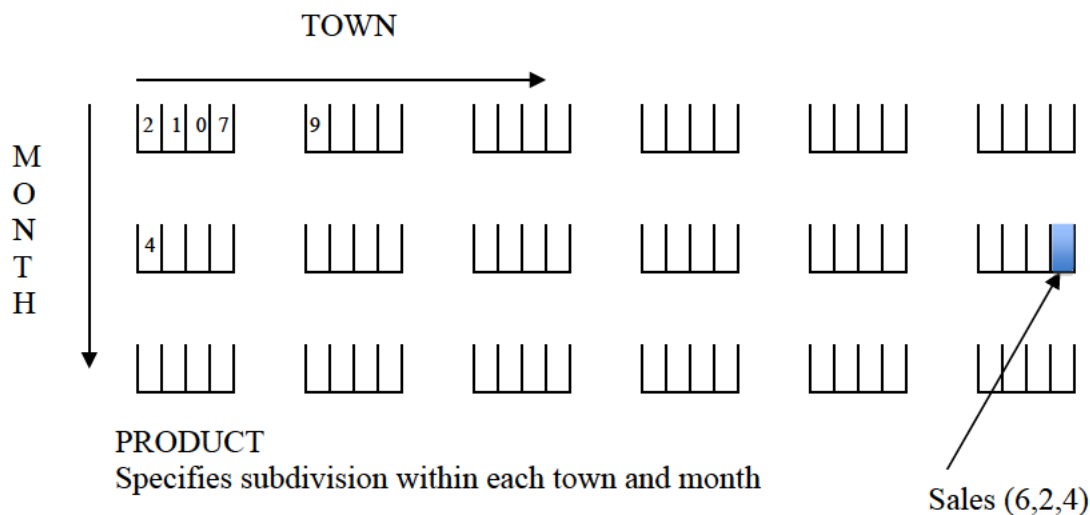
Arrays can be of many dimensions. A useful way to diagrammatically represent a multidimensional array is to think of each element being further subdivided with each successive dimension. Individual elements are accessed using one index for each dimension, each of which is a simple integer variable.

Consider a 3-dimensional array Sales (Town, Month, Product) which stores information about sales of four different products in each month for a number of towns.

Before using a multi-dimensional array in a program, some languages require that it must first be dimensioned (that is, defined) in order to allocate sufficient memory for the specified number of elements.

A statement such as the following is a typical example of the required code.

DIM Sales (6, 12, 4) as integer



To access each element in a multidimensional array, we use a series of nested FOR / NEXT loops:

```
BEGIN PrintProductSalesAndTotal
  FOR Town = 1 to 6
    FOR Month = 1 to 12
      FOR Product = 1 to 4
        Display "Town" Town, "Month" Month, "Product" Product;
        Display Sales (Town, Month, Product)
        Add Sales (Town, Month, Product) to Total
      NEXT Product
    NEXT Month
  NEXT Town
  Display "Total sales across all towns for all products sold this year =" Total
END PrintProductSalesAndTotal
```

Records

A record is a grouped list of variables, which may each be of different data types. Individual elements are accessed using their field names within the record.

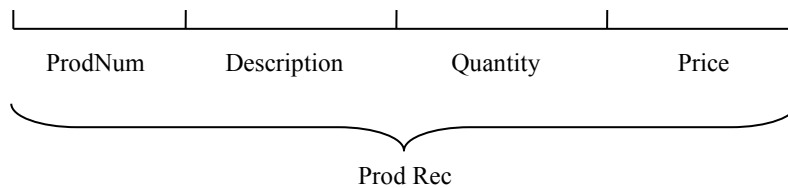
Before using a record in a program, most languages require that the record first be dimensioned (that is, defined) as a record type to specify the component field names and data types. If the component fields within the record are strings, their length must also be specified.

In the following example, a Product record called ProdRec is defined to contain ProdNum, description, quantity and price.

A statement such as the following is a typical example of the required code.

```
DIM RECORD ProdRec
    ProdNum as Integer
    Description as String (20)
    Quantity as Integer
    Price as Real
END RECORD
```

Although it is not mandatory to include such a definition in an algorithm, students may find it beneficial to include a relevant diagram, such as the one below, to help clarify their thinking.



Arrays of records

An array of records is an array, each element of which consists of a single record. The fields in that record may be of different data types.

Every record in the array must have the same structure, that is the same component fields in the same order.

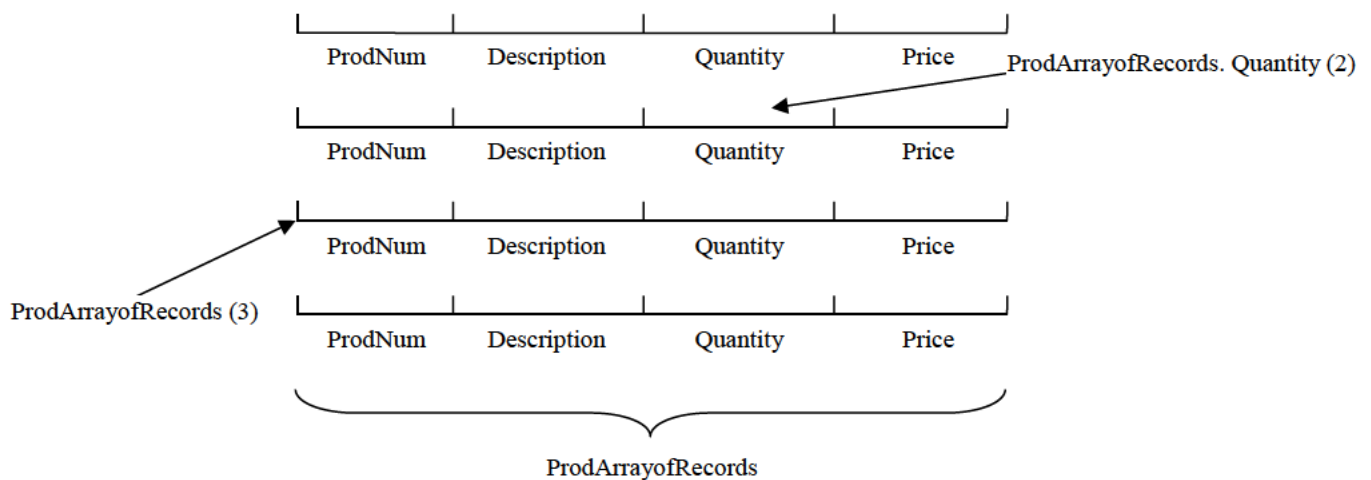
Before using an array of records in a program, most languages require that it must first be dimensioned (that is, defined) in order to allocate sufficient memory for the specified number of elements. This includes defining the record type to specify the component fields as in the simple record example on the previous page.

The array of records can then be defined as an array where each element is defined as one of these records.

In the following example, an array of records consisting of 20 such records is defined:

`DIM ProdArrayOfRecords (20) as TYPE ProdRec`

Although it is not mandatory to include such a definition in an algorithm, students may find it beneficial to include a relevant diagram, such as the one below, to help clarify their thinking.



Note: Individual fields within a record can either be accessed using their field names within the indexed record:

Display ProductArrayofRecords. ProdNum (3)

or individual records can be accessed. Whole records can be assigned, read or written:

```
BEGIN SwapProductRecords
    Let Temp = ProductArrayofRecords (i)
    Let ProductArrayofRecords (i) = ProductArrayofRecords (i + 1)
    Let ProductArrayofRecords (i + 1) = Temp
    'note that in this case, most languages require that the variable Temp also be defined as a
    record of type ProdRec
END SwapProductRecords
```

To read data from a sequential file into an array of records (for subsequent sorting, for example), the following pseudocode is used:

```
BEGIN LoadArrayof Records
    Open ProductFile for input
    Set i to 1
    Read ProdArrayofRecords (i) from ProductFile
    'note the use of the priming read in case the file is empty
    WHILE not EOF
        i = i + 1
        Read ProdArrayofRecords (i) from ProductFile
    END WHILE
    Close ProductFile
    Display "There are " i "product records read in from the file"
END LoadArrayofRecords
```

To write updated data from the array of records to a sequential file, the following pseudocode is used:

```
BEGIN WriteArrayofRecords
    Open ProductFile for output
    Set i to 1
    Write ProductFile from ProdArrayofRecords (i)
    WHILE i <= NumRecords
        i = i + 1
        Write ProductFile from ProdArrayofRecords (i)
    END WHILE
    Close ProductFile
    Display i "product records have been written to the file"
END WriteArrayofRecords
```

9 Searching and sorting algorithms

Each of these algorithms should be desk checked using appropriate values to ensure that the logic is understood. It is also important for students to implement these routines into their code so they can see the resultant output and understand the processing that has occurred.

Students are encouraged to make use of these routines (and relevant variations) in their major project(s).

Linear search

A linear search accepts a target value and checks every element of the array to be searched in turn, until either a match is found or the end of the array is reached.

The following example looks for a name in an array of names, and if found, retrieves the position of that name in the array:

```
BEGIN LinearSearch
  Let i = 1
  Let FoundIt = false
  Get RequiredName
  WHILE FoundIt is false AND i <= number of names
    IF Names(i) <> RequiredName THEN
      i = i + 1
    ELSE
      Let FoundIt = true
    ENDIF
  ENDWHILE
  IF FoundIt THEN
    'note the use of the expression FoundIt (which returns either true or false) rather than
    the more cumbersome FoundIt = true

    Display "Name found at position " i
  ELSE
    Display "Required person not found"
  ENDIF
END LinearSearch
```

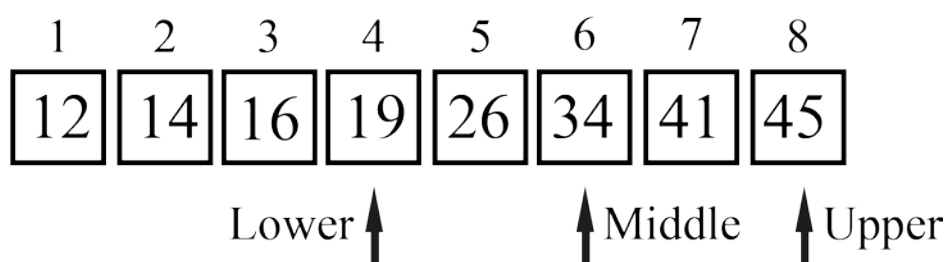
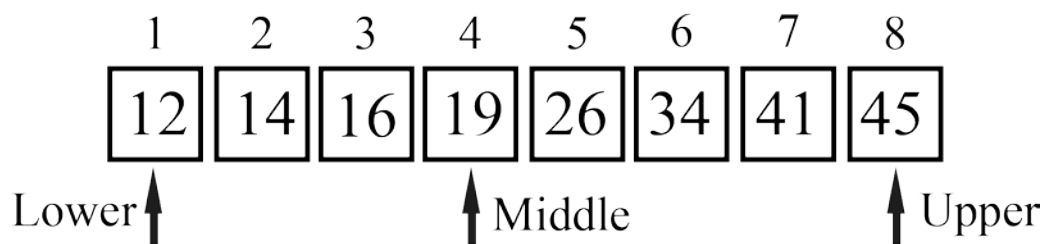
Binary Search

A binary search is used on a sorted array to find a required element more quickly than using a linear search. It divides the data set into two parts and determines in which part the element is to be found. That part of the array is again divided into two parts and a further decision is made as to which part may contain the target value. The process is continued until either the value is found or there are no more elements in the data set to be checked. If a match is found, the position of the match is reported, otherwise a message is written telling the user that the target is not present in the data.

At each division there are three possibilities for the target (if it exists in the array):

- (1) the target lies at the division point;
- (2) the target lies to the left of the division point
- (3) the target lies to the right of the division point

34 Target



```
BEGIN BinarySearch
  Let Lower = 1
  Let Upper = number of elements in the array
  Let FoundIt = false
  Get RequiredName
  REPEAT
    Let Middle = (Upper + Lower) / 2
    Let Middle = integer part of Middle
    IF RequiredName = Name (Middle) THEN
      Let FoundIt = true
      Let PositionFound = Middle
    ELSE
      IF RequiredName < Name (Middle) THEN
        Let Upper = Middle - 1
      ELSE
        Let Lower = Middle + 1
      ENDIF
    ENDIF
  UNTIL FoundIt OR Lower > Upper
  IF FoundIt THEN
    Display "Required name found at " PositionFound
  ELSE
    Display "Required name " RequiredName " not found."
  ENDIF
END BinarySearch
```

Bubble sort

A sort is used to arrange elements in an array into either ascending or descending order.

In a bubble sort the elements are compared in pairs and swapped if necessary. In this way, the larger of the pair ‘bubbles’ towards one end of the array. After each pass one more element will have moved to its correct position in the array.

1	2	3	4	5	6	
42	32	23	12	19	54	Compare first with second and Swap
1	2	3	4	5	6	
32	42	23	12	19	54	Compare second with third and Swap
1	2	3	4	5	6	
32	23	42	12	19	54	Compare third with fourth and Swap
1	2	3	4	5	6	
32	23	12	42	19	54	Compare fourth with fifth and Swap
1	2	3	4	5	6	
32	23	12	19	42	54	Compare fifth with sixth and leave
1	2	3	4	5	6	
32	23	12	19	42	54	Data set at end of first pass

The following algorithm sorts an array of names into ascending order.

```
BEGIN BubbleSort
  Let Last = number of names in the array
  Let Swapped = true
  WHILE Swapped = true
    Let Swapped = false
    Let i = 1
    WHILE i < Last
      IF Name (i) > Name (i+1) THEN
        Swap (Name (i), Name (i+1))
        Let Swapped = true
      ENDIF
      Increment i
    ENDWHILE
    Decrement Last
  ENDWHILE
END BubbleSort
```

When BubbleSort calls the subroutine Swap, the parameters passed are Name (i) and Name (i+1). In the subroutine Swap, more general names (A and B) are used for the parameters. This allows this swap routine to be used for swapping any two variables.

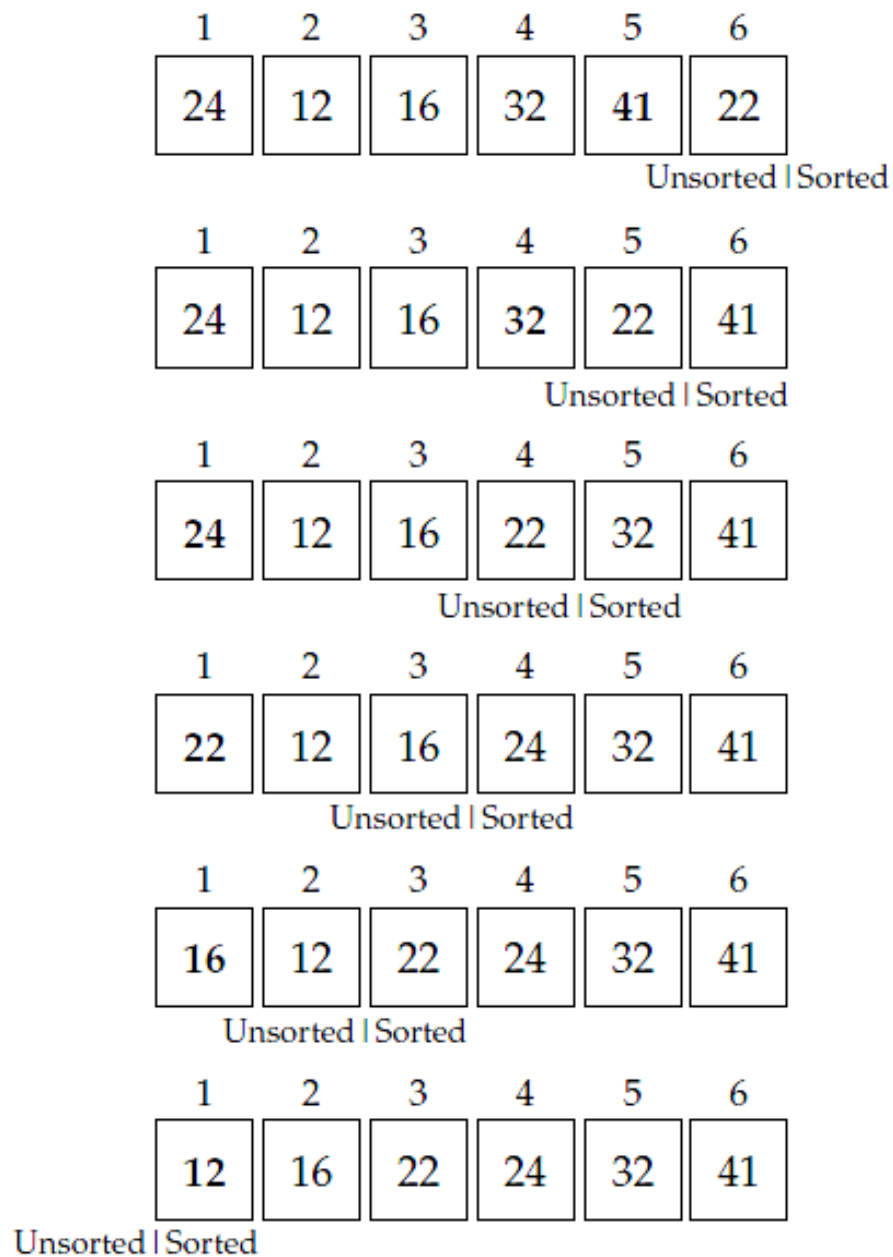
Note: Assuming that the array is declared as a global variable, any changes made to the array in this Swap routine will be reflected in the array as seen and used by all routines.

```
BEGIN Swap (A, B)
  Let Temp = A
  Let A = B
  Let B = Temp
END Swap
```

Selection sort

A sort is used to arrange elements in an array into either ascending or descending order. An ascending selection sort successively looks for the largest value in the array and swaps it with the last element.

To achieve this we need to logically divide the array into two parts – an unsorted part and a sorted part. Each pass through the unsorted part finds the largest value and places it at the start of the sorted part. Initially the sorted part is empty. The size of the sorted part of the array increases by 1 with each pass.



The following algorithm sorts an array of names into ascending order.

```
BEGIN SelectionSort
  Let EndUnsorted = number of names in the array
  WHILE EndUnsorted > 1
    Let i = 1
    Let Max = Name (i)
    Let PosMax = i
    WHILE i <= EndUnsorted
      Increment i
      IF Name (i) > Max THEN
        Let Max = Name (i)
        Let PosMax = i
      ENDIF
    ENDWHILE
    Swap (Name (PosMax), Name (EndUnsorted))
    Decrement EndUnsorted
  ENDWHILE
END SelectionSort
```

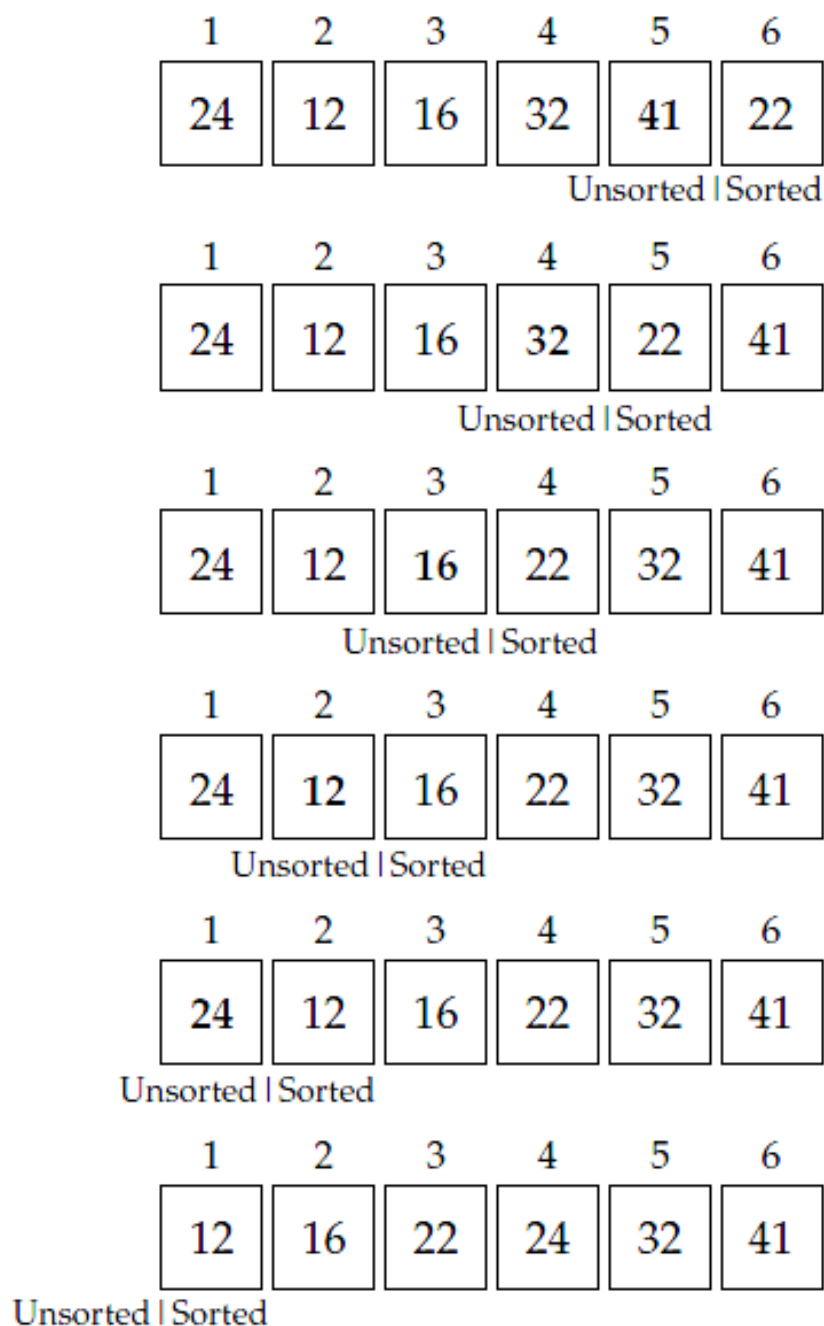
When SelectionSort calls the subroutine Swap, the parameters passed are Name (PosMax) and Name (EndUnsorted). In the subroutine Swap, general names (A and B) are used for the parameters. This allows this swap routine to be used for swapping any two variables.

```
BEGIN Swap (A, B)
  Let Temp = A
  Let A = B
  Let B = Temp
END Swap
```

Insertion sort

A sort is used to arrange elements in an array into either ascending or descending order. An ascending insertion sort successively takes the next element to be sorted and places it in its correct position in the sorted part of the array.

To achieve this we need to logically divide the array into two parts – an unsorted part and a sorted part. Each pass through the unsorted part takes the end value of the unsorted part and places it in the correct position. It achieves this by successively moving the correct number of elements in the sorted part by one position to make room. Initially the sorted part contains one element. The size of the sorted part of the array increases by 1 with each pass.



The following algorithm sorts an array of names into ascending order.

```
BEGIN InsertionSort
  Let First = 1
  Let Last = number of elements in the array
  Let PositionOfNext = Last – 1
  WHILE PositionOfNext >= First
    Let Next = Name (PositionOfNext)
    Let Current = PositionOfNext
    WHILE (Current < Last) AND (Next > Name (Current + 1))
      'look for the position into which to insert the current name, and shuffle the
      sorted elements along until we find it.

      Increment Current
      Let Name (Current – 1) = Name (Current)
    ENDWHILE
    Let Name (Current) = Next
    'put the current name to be sorted into its correct place
    Decrement PositionOfNext
    'effectively shorten the length of the unsorted portion of the array
  ENDWHILE
END InsertionSort
```

Note: Variations of the algorithm are possible. In some versions, the task of finding the correct place to insert is separated from the task of moving the elements to make room. By doing this, the task of finding the correct place to insert can be done more quickly by using a binary search rather than a linear search.

Adding a new element to a sorted array

Frequently a new element needs to be added to an already sorted array. Although this can be done by appending the element to the end of the array and performing a complete sort, this is unnecessarily wasteful of CPU resources.

It is more efficient to locate the correct position for the element and shuffle down all the elements after this position by one to make room. This uses the same logical approach as the insertion sort.

The following algorithm inserts a new name into its correct position in an existing array of names, which is already in ascending order. Note that the array and the number of names in it are assumed to be global.

```

BEGIN InsertElement
  Get NewName
  LinearSearch (NewName, Position)
  'this returns the position in the array of names where the new name is to be inserted
  Let First = 1
  Let Last = number of elements in the array
  WHILE (i < Position)
    'shuffle the sorted elements down one position working backwards from the end of
    the array.

    Let Name (i + 1) = Name (i)
    Decrement i
  ENDWHILE
  Let Name (Position) = NewName
  'put the new name into its correct place
END InsertElement

BEGIN LinearSearch (RequiredName, i)
  Let i = 1
  Let Found = false
  Get RequiredName
  WHILE Found is false AND i <= number of names
    IF Names(i) < RequiredName THEN
      i = i + 1
    ELSE
      Let Found = true
    ENDIF
  ENDWHILE
  IF not Found THEN
    i = number of names + 1
  ENDIF
END LinearSearch

```

Note: The linear search used in this algorithm could be replaced by a binary search for more efficient processing. The same general approach can be used to delete an element from an array.

10 Documentation

Testing report

When testing in a commercial environment it is important that the process is fully documented. This allows the project leader to be fully informed of relevant issues and how they were handled. It also ensures that the final system has been through a rigorous testing process, resulting in a quality product. It is one of the steps in quality assurance.

An example of a test report for a payroll system currently being developed is shown below. Note the inclusion of a narrative discussion of problems encountered during testing, a table of expected outcomes and a table of what occurred during testing.

Test Report

Item being tested: Payroll Program to Calculate Gross Pay.

Date 28/8/95

Tester: Arthur Galletly

Type of test: Black box test

Appendix 1: Test data used and expected results

Appendix 2: Actual test results

Interpretation of Test

Sample representative data was processed adequately for typical data. There were, however, problems encountered with atypical data.

The following problems were encountered during the test:

1. There was no warning given when a value of 240 hours was entered as the number of hours worked.
2. When the hourly rate was unusually large, there was no warning. Also, the gross pay was then too large to fit in the space permitted and came out as #####.
3. Program behaved as though negative hours worked or negative rate of pay were normal, instead of displaying an error message.
4. When either the hours worked or rate of pay were not specified, the program halted and had to be restarted. It should give an error message and continue operation.
5. Program truncated rather than rounded. Instead of \$6.1076 being rounded to \$6.11, the result was truncated to \$6.10. Another problem with the formatting of the output is that the result was displayed at \$6.1 instead of \$6.10.
6. The program crashed when there was data of incorrect type entered for either the hours or the rate. The program needs to be revised so that it prints an error message and does not crash.

Appendix 1 Test data used and expected results

Name	Hours	Rate	Expected gross pay	OR Error response
Fred	25	10	250	
Ted	40	15	600	
Ted	42 +	15	645	
Ted	60	15	1050	
Mary	240	15	5100	Invalid number of hours
Jill	10	9999 .99	99999 .99	
Sue	10	-10	-100	Error: do not print
Joan	-10	10	-100	Error: do not print
Bill	0	0	0	Do not print
Ben	25			Insufficient data
John	49			Insufficient data
Karen	10	0		Do not print
Ric	0	10		Do not print
James	20 .375	10	203 .75	
Neville	10	12 .125	120 .25	
Eric	2 .25	2 .25	6 .0625	
Sonya	2 .26	2 .26	6 .1076	
Graham	20A	5		Print data error
Bob	20	A5		Print data error

Appendix 2 Actual test results

Name	Hours	Rate	Gross pay	OR Error response
Fred	25	10	250	
Ted	40	15	600	
Ted	42 +	15	645	
Ted	60	15	1050	
Mary	240	15	3100	
Jill	10	9999 .99	#####	
Sue	10	-10	-100	
Joan	-10	10	-100	
Bill	0	0	0	
Ben	25			Program halted
John		49		Program halted
Karen	10	0		Do not print
Ric	0	10		Do not print
James	20 .375	10	203 ##	
Neville	10	12 .125	121 ##	
Eric	2 .25	2 .25	6 ##	
Sonya	2 .26	2 .26	6 .1	
Graham	20A	5		Program crashed
Bob	20	A5		Program crashed

Note: This is just one possible representation. There are many other valid ways of documenting the testing process.

Maintenance documentation

When maintaining software in a commercial environment it is important that the process is fully documented. This ensures that all versions are clearly identified and that all changes made, and the reasons for them, are recorded.

Below is an example of maintenance documentation included at the top of the current version of the source code for a payroll system. Note the inclusion of author name(s), dates, version numbers and reasons for change. There will also be supporting documentation in the form of comments in the relevant subroutine(s).

```

* *****
* System Name: Payroll Suite
*
* Program Name: Payslip Production Program          Author: Arthur Galletly
*
* Program ID: PR060                                Date first written:    1/4/95
*
* Current Version Number: 4                        Date released:       8/8/95
*
* Compiler:      MMB D ANSI 85 Version 7.6
*
* Purpose:       This program calculates gross pay and tax, prints a payslip,
*                records the details of the payslips in the payslip file in case
*                of later query, and updates the year to date gross pay, tax and
*                nett pay in the employee file
* *****
* Version Number: 4                                Date released:       8/8/95
*
* Purpose of change: Fix bug calculating Annual Holidays Due
*
* Description of   Version 3 did not deduct holidays taken during
* changes made:    the current pay period. This has been fixed in
*                version 4
* *****
* Version Number: 3                                Date released:       7/7/95
*
* Purpose of change: New style of payslip introduced
*
* Description of   Print_Proc modified to include modified layout and
* changes made:    extra fields Days of Annual Holidays Due and YTD
*                Sick Leave Used and Long Service Leave Accrued
* *****
* Version Number: 2                                Date released:       6/6/95
*
* Purpose of change: To allow reprints of specified employees where there
*                is a problem with the printer
*
* Description of   New procedure added which prompts user to indicate
* changes made:    whether a full run is required or a reprint is
*                required. New procedure is called "Prompt_Proc"
* *****

```

This maintenance documentation and the source code are clearly quite old. This highlights the need for thorough documentation, as maintenance and upgrades may be required at any stage. It is crucial for an organisation to store original documentation of software systems for future reference.