Introduction to C++ Programming I

Ian Aitchison and Peter King

August 1997

Contents

About the course

Welcome to Heriot-Watt University *Introduction to Computing I* course. This course is being provided for students taking the Certificate in Science of Aberdeen and Heriot-Watt Universities. It is being delivered by way of the World Wide Web, however all students will also receive the entire course in a printed format.

The course is arranged in a series of Lessons. As well as new material to learn, each Lesson contains review questions, multiple choice questions, and programming exercises. The review questions are designed to get you to think about the contents of the Lesson. These review questions should be straightforward; if the answer is not obvious, reread the Lesson. The multiple choice questions can be answered on-line when you are using the World Wide Web; feedback is given if you choose the wrong answers. The programming exercises are an essential part of the course. Computer programming is a practical subject, without practice no progress will be made.

There will be some assessed coursework, this may be submitted by electronic mail. Electronic mail will also be used to provide help with problems.

Course contents

This course is intended as a first introduction to programming computers using the C++ programming language. It is not assumed that the student has done any programming before hence this course is not comprehensive and does not cover all of C++. In particular it does not cover any of the object-oriented features of C++, these are introduced in the following course (Introduction to Computing II). Because this is a first programming course emphasis is placed on the design of programs in a language-independent fashion. A brief introduction to computers is also given.

The lessons of the course may be split into groups as follows:

1. About the computer and computer systems.

    - Lesson 1 - The Computer. Covers the Central processor, memory, information representation and the operation cycle of the computer.

- Lesson 2 - Programming Languages. Covers the various levels of Programming Languages.
- Lesson 3 - Operating Systems. Covers the purpose of Operating systems and the major types.
- Lesson 4 - Preparing a Computer Program. Covers the steps that are carried out in going from a problem specification to having a well-tested and reliable program to solve the problem.

2. About the design of programs.
    - Lesson 5 - Algorithms. The basic constructs used in designing programs. Sequence, selection and repetition.
    - Lesson 9 - Introduction to Structured Design. Top-down design of algorithms using sequence and selection only.
    - Lesson 15 - Further Structured Design. Top-down design of algorithms using repetition.
    - Lesson 20 - Top-down design using Functions. An introduction to problem-solving by splitting the problem into sub-problems whose solutions are implemented as functions.

3. About C++
    - Lesson 6 - A simple C++ program. Looks at a simple C++ program and identifies features. Covers simple ideas about variables and their declaration and input and output.
    - Lesson 7 - The Assignment statement. How values are assigned to variables.
    - Lesson 8 - Further Assignment Statements & Control of Output. More forms of assignment statement and simple formatting of output.
    - Lesson 10 - Conditions. How expressions that can be true or false are written in C++.
    - Lesson 11 - The `if` statement. How conditional execution of a statement is carried out.
    - Lesson 12 - The `if-else` statement. How a choice can be made between which of two statements to execute.
    - Lesson 13 - Nested `if` and `if-else` statements. How multi-choice decisions can be made.
    - Lesson 14 - The `switch` statement. An alternative way to implement multi-choice conditions.
    - Lesson 16 - The `while` statement. The basic way to carry out repetition.

- Lesson 17 - The `do-while` statement. An alternative way to carry out repetition.
- Lesson 18 - The `for` statement. Repetition a set number of times or as a control loop variable takes a set of values.
- Lesson 19 - Streams and External Files. How to input and output data from and to external files.
- Lesson 21 - An introduction to User-defined functions in C++. How to design your own functions.
- Lesson 22 - Further User-defined functions. Returning information by parameters.
- Lesson 23 - Arrays. How to work with large collections of indexed data.

Note that there is also a document 'The Computer Exercises' which gives some help in using Windows and the Borland C++ compilers.

Suggested study timetable

It is suggested that you should cover the contents of the course in about 13/14 weeks. The lessons are not equal in size or in the difficulty of their content. The following timetable is suggested.

Week 1 Try to cover most of the material in Lessons 1 to 4 during this first week. This is mainly background material for those who have not had much contact with computers before but some (or all) of it will probably be familiar to most people taking the course. During this first week try to familiarise yourself with the computer system you are going to use so that you are ready to start programming as soon as possible. Included with the course notes (The Computer Exercises) is a document about using the Borland C++ compilers, check over this and go through the suggested steps in compiling a small supplied C++ program. This activity may well expand into week 2.

Week 2 Read Lesson 5 on Algorithms. Don't worry if you find it difficult to make up your own algorithms at this stage, but do make an attempt. You'll continue to get practice at this as long as you program! Also start looking at Lesson 6 on simple programs and their features. Implement your solutions to the exercises on the computer.

Week 3 Cover Lessons 7 and 8 on assignment and implement the exercises on simple calculation type programs.

Week 4 Now study Lesson 9 on Algorithms with simple selection. Read Lesson 10 on how conditional expressions are constructed in C++.

Week 5 Study Lessons 11 and 12 on `if` and `if-else` statements and implement the exercise on programs with simple selection.

Week 6 Continue the study of selection mechanisms with Lessons 13 and 14 on nested `if` and `if-else` statements and the `switch` statement. The `switch` statement is not as fundamental as the others so spend less time on it than the other selection mechanisms.

Week 7 Study Lesson 15 on repetition constructs and practice designing algorithms using repetition. Commence Lesson 16 on the **while** statement.

Week 8 Continue Lesson 16 on the `while` statement. Also cover Lesson 17 on the `do-while` statement this week. Do not spend so much time on it as on the more fundamental `while` statement.

Week 9 Cover Lesson 18 on the `for` statement. This statement is used frequently and so is an important statement. Have a look at Lesson 19 on Streams and Files so that you can use an External file in the last assignment.

Week 10 Study Lesson 20 on structured design using functions. This is a very important Lesson. If you have time start on Lesson 21 on how to construct C++ functions.

Week 11 Carry on with Lesson 21 on user-defined functions. Continue on to Lesson 22. In carrying out the exercises in these Lessons you will also get further practice on using the conditional and repetition control structures.

Week 12 Study Lesson 23 which introduces the concept of arrays. Again this is an important concept in programming. The exercises for this chapter will provide you with more practice in the use of control structures and functions.

Week 13 Finish off the course and start revision.

Assessment

There will be two class assignments to carry out. One will be given out which requires that you know the material up to and including Lesson 12. The other will require that you know the material up to and including Lesson 20 (19 is not needed for the assignment). The first assignment will be worth 10% in the final assessment and the second assignment will be worth 15% in the final assessment. The remaining 75% will come from the class examination.

Further information will be available on the World Wide Web version of the course.

Accessing the course on the World Wide Web

The course, and its successor Introduction to Computing II, are available on the World Wide Web (WWW). You are assumed to know how to connect to WWW. The course has been developed using Netscape as the browser, but it should be possible to access it equally well with Microsoft's Internet Navigator, or with Mosaic. To access the course, use your browser to open the `URL`

`http://www.cee.hw.ac.uk/`

This will present you with a page about the Department of Computing and Electrical Engineering at Heriot-Watt University. Follow the link labelled

Distance Learning Courses

The page loaded will give general information about the courses. There will also be reminders about approaching deadlines for coursework, and notification of any corrections to parts of the course. (Any corrections will have been made in the WWW version of the course, but this will enable you to annotate your printed version of the notes.) You are advised to always enter the course in this fashion. Saving `bookmarks` with your browser may cause problems if any corrections cause extra sections or questions to be added to the course.

Coursework should be submitted using electronic mail. Send any files requested to `pjbk@cee.hw.ac.uk`, with clear indications of which assignment is being submitted within the file.

Getting Help

Although the notes are intended to be comprehensive, there will undoubtedly be times when you need help. There are two mechanisms for this. An electronic mailing list has been set up at Heriot-Watt University which copies messages submitted to it to all members of the course. The course organisers and teachers receive all messages sent to this list. All students should join this mailing list, and use it to replace the discussion that would normally take place in a classroom. In order to join the mailing list, send an electronic mail message to `majordomo@cee.hw.ac.uk` with the following contents:

```
subscribe pathways
```

Once you have joined the list, messages to the list are sent as ordinary electronic mail addressed to `pathways@cee.hw.ac.uk`

If your problems are of a more individual nature, you may prefer to send them to the course teacher only, in which case an email message should be addressed to `pjbk@cee.hw.ac.uk`

Lesson 1

The Computer

Before considering the programming of a computer a brief overview of the basic structure of a Computer in terms of its main components is given. Every Computer has the following general structure:

1.1 Central Processing Unit

The Central Processing Unit (CPU) performs the actual processing of data. The data it processes is obtained, via the system bus, from the main memory. Results from the CPU are then sent back to main memory via the system bus. In addition to computation the CPU controls and co-ordinates the operation of the other major components. The CPU has two main components, namely:

1. **The Control Unit** — controls the fetching of instructions from the main memory and the subsequent execution of these instructions. Among other tasks carried out are the control of input and output devices and the passing of data to the Arithmetic/Logical Unit for computation.

2. **The Arithmetic/Logical Unit (ALU)** — carries out arithmetic operations on integer (whole number) and real (with a decimal point) operands. It can also perform simple logical tests for equality and greater than and less than between operands.

It is worth noting here that the only operations that the CPU can carry out are simple arithmetic operations, comparisons between the result of a calculation and other values, and the selection of the next instruction for processing. All the rest of the apparently limitless things a computer can do are built on this very primitive base by programming!

Modern CPUs are very fast. At the time of writing, the CPU of a typical PC is capable of executing many tens of millions of instructions per second.

1.2 Memory

The memory of a computer can hold program instructions, data values, and the intermediate results of calculations. All the information in memory is encoded in fixed size cells called **bytes**. A byte can hold a small amount of information, such as a single character or a numeric value between 0 and 255. The CPU will perform its operations on groups of one, two, four, or eight bytes, depending on the interpretation being placed on the data, and the operations required.

There are two main categories of memory, characterised by the time it takes to access the information stored there, the number of bytes which are accessed by a single operation, and the total number of bytes which can be stored. **Main Memory** is the working memory of the CPU, with fast access and limited numbers of bytes being transferred. **External memory** is for the long term storage of information. Data from external memory will be transferred to the main memory before the CPU can operate on it. Access to the external memory is much slower, and usually involves groups of several hundred bytes.

1.2.1 Main memory

The main memory of the computer is also known as **RAM**, standing for Random Access Memory. It is constructed from integrated circuits and needs to have electrical power in order to maintain its information. When power is lost, the information is lost too! It can be directly accessed by the CPU. The access time to read or write any particular byte are independent of whereabouts in the memory that byte is, and currently is approximately 50 **nanoseconds** (a thousand millionth of a second). This is broadly comparable with the speed at which the CPU will need to access data. Main memory is expensive compared to external memory so it has limited capacity. The capacity available for a given price is increasing all the time. For example many home Personal Computers now have a capacity of 16 **megabytes** (million bytes), while 64 megabytes is commonplace on commercial workstations. The CPU will normally transfer data to and from the main memory in groups of two, four or eight bytes, even if the operation it is undertaking only requires a single byte.

1.2.2 External Memory

External memory which is sometimes called *backing store* or *secondary memory*, allows the permanent storage of large quantities of data. Some method of magnetic recording on magnetic disks or tapes is most commonly used. More recently optical methods which rely upon marks etched by a laser beam on the surface of a disc (CD-ROM) have become popular, although they remain more expensive than magnetic media. The capacity of external memory is high, usually measured in hundreds of megabytes or even in **gigabytes** (thousand million bytes) at present. External memory has the important property that the information stored is not lost when the computer is switched off.

The most common form of external memory is a **hard disc** which is permanently installed in the computer and will typically have a capacity of hundreds of megabytes. A hard disc is a flat, circular oxide-coated disc which rotates continuously. Information is recorded on the disc by magnetising spots of the oxide coating on concentric circular tracks. An access arm in the disc drive positions a read/write head over the appropriate track to read and write data from and to the track. This means that before accessing or modifying data the read/write head must be positioned over the correct track. This time is called the **seek time** and is measured in **milliseconds**.

There is also a small delay waiting for the appropriate section of the track to rotate under the head. This **latency** is much smaller than the seek time. Once the correct section of the track is under the head, successive bytes of information can be transferred to the main memory at rates of several megabytes per second. This discrepancy between the speed of access to the first byte required, and subsequent bytes on the same track means that it is not economic to transfer small numbers of bytes. Transfers are usually of blocks of several hundred bytes or even more. Notice that the access time to data stored in secondary storage will depend on its location.

The hard disc will hold all the software that is required to run the computer, from the operating system to packages like word-processing and spreadsheet programs. All the user's data and programs will also be stored on the hard disc. In addition most computers have some form of removable storage device which can be used to save copies of important files etc. The most common device for this purpose is a **floppy disc** which has a very limited capacity. Various magnetic tape devices can be used for storing larger quantities of data and more recently removable optical discs have been used.

It is important to note that the CPU can only directly access data that is in main memory. To process data that resides in external memory the CPU must first transfer it to main memory. Accessing external memory to find the appropriate data is slow (milliseconds) in relation to CPU speeds but the rate of transfer of data to main memory is reasonably fast once it has been located.

1.3 Input/Output Devices

When using a computer the text of programs, commands to the computer and data for processing have to be entered. Also information has to be returned from the computer to the user. This interaction requires the use of input and output devices.

The most common input devices used by the computer are the **keyboard** and the **mouse**. The keyboard allows the entry of textual information while the mouse allows the selection of a point on the screen by moving a screen cursor to the point and pressing a mouse button. Using the mouse in this way allows the selection from menus on the screen etc. and is the basic method of communicating with many current computing systems. Alternative devices to the mouse are tracker balls, light pens and touch sensitive screens.

The most common output device is a **monitor** which is usually a Cathode Ray Tube device which can display text and graphics. If **hard-copy** output is required then some form of **printer** is used.

1.4 The system bus

All communication between the individual major components is via the **system bus**. The bus is merely a cable which is capable of carrying signals representing data from one place to another. The bus within a particular individual computer may be specific to that computer or may (increasingly) be an industry-standard bus. If it is an industry standard bus then there are advantages in that it may be easy to upgrade the computer by buying a component from an independent manufacturer which can plug directly into the system bus. For example most modern Personal Computers use the PCI bus.

When data must be sent from memory to a printer then it will be sent via the system bus. The control signals that are necessary to access memory and to activate the printer are also sent by the CPU via the system bus.

1.5 More about memory and information representation

All information inside the computer and on external storage devices is represented in a form related to the **Binary** number system. Binary is a number system which uses the base 2 instead of the base 10 decimal system that is used in normal life. In the decimal system a positional number system is used which allows all numbers to be expressed using only the digits 0–9. Successive digits from the right represent the number of the corresponding power of 10. Thus 123 in decimal is

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

that is, one hundred, plus two tens, plus three units. Similarly the binary number system builds all numbers from the bits 0 and 1, and powers of 2. 101 in the binary number system represents the number with value

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

which is of course the decimal number 5. Using the binary system allows all computation inside the computer to take place using cheap two-state electronic devices.

Modern computers organise information into small units called **bytes** which hold eight **bits**, each bit representing a 0 or a 1. Each byte may hold a single character of text or a small integer. Larger numbers, computer instructions and character strings occupy several bytes.

The main memory can be thought of as a series of bytes numbered from 0, 1, 2, 3, . . . upwards, each byte containing a pattern of eight bits which can be accessed by the CPU when it supplies the number, or **Address**, of the byte required. For example consider the section of memory illustrated below:

Address	Contents
3168	`10110111`
3167	`01000111`
3166	`01010101`

The byte with the address 3167 contains the binary pattern 01000111. Depending on circumstances the CPU may interpret this as an instruction, as a number (or part of a number) or as a character. This pattern can represent the character `G` in the ASCII character code that is used almost universally now or it could represent the decimal number 71.

It is important to keep a clear distinction in your mind of the difference between the address of a memory location and the contents of that memory location.

1.5.1 Representation of information in external memory

Information on external memory is organised into **files**, each file containing some related information. For example a file could hold the text of a document, a computer program, a set of experimental results etc. Just as in main memory the information in a file is represented as a collection of bytes.

1.6 The execution cycle

When a computer obeys the instructions in a computer program it is said to be **running** or **executing** the program. Before a computer can execute a computer program the program must be resident in memory. The program must occupy a set of consecutive bytes in memory and must be written in the **internal machine language** for the computer. Each CPU has its own machine language which means that before a program can be executed on another CPU it has to be re-written in the internal machine language of the other CPU.

The concept that the program to be executed is stored in the computer memory is an important one. This is what makes the computer such a general-purpose problem-solving machine — merely by changing the program stored in memory the computer can be used to carry out an entirely different task.

Once the program is loaded into memory then the following sequence is carried out:

```
set instruction address to the address of
```

```
    the first instruction
while program not finished
  {
    fetch instruction from current instruction address
    update current instruction address
    execute the fetched instruction
   }
```

This sequence is usually called the **fetch-execute cycle**.

1.7 Summary

- A computer consists of a Central Processing Unit(CPU), memory and various devices which can be categorised as Input/Output Devices. Information is communicated between these separate units by the Systems Bus.
- The Central Processing Unit (CPU) consists of a Control Unit and an Arithmetic/Logic Unit (ALU). The control unit controls the operation of the peripheral devices and the transfer of information between the units that make up the computer. The Arithmetic/Logic Unit performs calculation.
- The memory of the computer is split into main memory and external memory.
- Main memory is fast and limited in capacity. The CPU can only directly access information in main memory. Main memory cannot retain information when the computer is switched of. Main memory consists of a series of numbered locations called **bytes**, each byte being eight **bits**. The number associated with a byte is the **address** of the byte.
- Secondary memory is slow and virtually unlimited in capacity. It retains information when the computer is switched off. Information on external memory can only be accessed by the CPU if it is first transferred to main memory.
- The internal representation of information in the computer and on external memory is in terms of the Binary system using only the basic symbols 0 and 1.
- Programs to be executed by the computer are placed in main memory and the CPU fetches each instruction in turn from memory and executes it.

- Each type of CPU has its own machine language, the set of instructions it can obey.

1.8 Multiple Choice Questions

1. What are the five main components of a computer system?
   (a) CPU, CD-rom, mouse, keyboard, sound card
   (b) Memory, Video Card, Monitor, Software, Hardware.
   (c) Modem, Keyboard, Word Processor, Printer, Screen.
   (d) CPU, memory, system bus, input, output
2. How do the main components of the computer communicate with each other?
   (a) system bus
   (b) memory
   (c) keyboard
   (d) monitor

1.9 Review Questions

1. What are the tasks carried out by the Central Processing Unit
2. What are the major differences between main memory and external memory?
3. What is stored in main memory? What is stored in external memory?
4. How is information represented inside computer systems?
5. In the following diagram of memory what are the contents of the memory location with the address 98? What is the address of the memory location whose contents are 10100101? What could these contents represent?

100	`00010010`
99	`10100101`
98	`11011101`

6. What has to be done before a program can be executed in a computer? How is the program executed by the CPU?
7. What benefit derives from the 'stored program concept'?

Lesson 2

Programming Languages

As noted in section 1.6 all computers have an internal machine language which they execute directly. This language is coded in a binary representation and is very tedious to write. Most instructions will consist of an operation code part and an address part. The operation code indicates which operation is to be carried out while the address part of the instruction indicates which memory location is to be used as the operand of the instruction. For example in a hypothetical computer successive bytes of a program may contain:

operation code	address	meaning
`00010101`	`10100001`	load c(129) into accumulator
`00010111`	`10100010`	add c(130) to accumulator
`00010110`	`10100011`	store c(accumulator) in location 131

where c( ) means 'the contents of' and the accumulator is a special register in the CPU. This sequence of code then adds the contents of location 130 to the contents of the accumulator, which has been previously loaded with the contents of location 129, and then stores the result in location 131. Most computers have no way of deciding whether a particular bit pattern is supposed to represent data or an instruction.

Programmers using machine language have to keep careful track of which locations they are using to store data, and which locations are to form the executable program. Programming errors which lead to instructions being overwritten with data, or erroneous programs which try to execute part of their data are very difficult to correct. However the ability to interpret the same bit pattern as both an instruction and as data is a very powerful feature; it allows programs to generate other programs and have them executed.

2.1 Assembly Language

The bookkeeping involved in machine language programming is very tedious. If a programmer is modifying a program and decides to insert an extra data item, the addresses of other data items may be changed. The programmer will have to carefully examine the whole program deciding which bit patterns represent the addresses which have changed, and modify them.

Human beings are notoriously bad at simple repetitive tasks; computers thrive on them. Assembly languages are a more human friendly form of machine language. Machine language commands are replaced by mnemonic commands on a one-to-one basis. The assembler program takes care of converting from the mnemonic to the corresponding machine language code. The programmer can also use symbolic addresses for data items. The assembler will assign machine addresses and ensure that distinct data items do not overlap in storage, a depressingly common occurrence in machine language programs. For example the short section of program above might be written in assembly language as:

operation code	address
LOAD	A
ADD	B
STORE	C

Obviously this leaves less scope for error but since the computer does not directly understand assembly language this has to be translated into machine language by a program called an **assembler**. The assembler replaces the mnemonic operation codes such as ADD with the corresponding binary codes and allocates memory addresses for all the symbolic variables the programmer uses. It is responsible for associating the symbol `A`, `B`, and `C` with an addresses, and ensuring that they are all distinct. Thus by making the process of programming easier for the human being another level of processing for the computer has been introduced. Assembly languages are still used in some time-critical programs since they give the programmer very precise control of what exactly happens inside the computer. Assembly languages still require that the programmer should have a good knowledge of the internal structure of the computer. For example, different `ADD` instructions will be needed for different types of data item. Assembly languages are still machine specific and hence the program will have to be re-written if it is to be implemented on another type of computer.

2.2 High level Languages

Very early in the development of computers attempts were made to make programming easier by reducing the amount of knowledge of the internal workings of the computer that was needed to write programs. If programs could be presented in a language that was more familiar to the person solving the problem, then fewer mistakes would be made. **High-level** programming languages allow the specification of a problem solution in terms closer to those used by human beings. These languages were designed to make programming far easier, less error-prone and to remove the programmer from having to know the details of the internal structure of a particular computer. These high-level languages were much closer to human language. One of the first of these languages was Fortran II which was introduced in about 1958. In Fortran II our program above would be written as:

```
C = A + B
```

which is obviously much more readable, quicker to write and less error-prone. As with assembly languages the computer does not understand these high-level languages directly and hence they have to be processed by passing them through a program called a **compiler** which translates them into internal machine language before they can be executed.

Another advantage accrues from the use of high-level languages if the languages are standardised by some international body. Then each manufacturer produces a compiler to compile programs that conform to the standard into their own internal machine language. Then it should be easy to take a program which conforms to the standard and implement it on many different computers merely by re-compiling it on the appropriate computer. This great advantage of portability of programs has been achieved for several high-level languages and it is now possible to move programs from one computer to another without too much difficulty. Unfortunately many compiler writers add new features of their own which means that if a programmer uses these features then their program becomes non-portable. It is well worth becoming familiar with the standard and writing programs which obey it, so that your programs are more likely to be portable.

As with assembly language human time is saved at the expense of the compilation time required to translate the program to internal machine language. The compilation time used in the computer is trivial compared with the human time saved, typically seconds as compared with weeks.

Many high level languages have appeared since Fortran II (and many have also disappeared!), among the most widely used have been:

COBOL	Business applications
FORTRAN	Engineering & Scientific Applications
PASCAL	General use and as a teaching tool
C & C++	General Purpose - currently most popular
PROLOG	Artificial Intelligence
JAVA	General Purpose, gaining popularity rapidly,

All these languages are available on a large variety of computers.

2.3 Summary

- Each CPU has its own internal machine language. Programming at this internal machine level is usually carried out in Assembly language which is machine specific and relates on an instruction to instruction basis to the internal machine language.
- High-level languages are translated by a compiler program into internal machine language. The compiled code is linked with any system libraries required and the final program loaded into memory before execution of the program can take place.
- If an agreed standard is produced for a high-level language then any program which conforms to the standard should be able to run on any computer after compiling it with a machine-specific compiler. This gives the advantage of portability of programs.

2.4 Multiple Choice Questions

1. What is the only language that a computer understands directly?
   (a) English, as spoken in Boston, Mass.
   (b) BASIC, the Beginners' All-purpose Symbolic Instruction Code
   (c) machine language, different for every type of CPU
2. What are the three main *types* of computer programming languages?
   (a) machine language, assembly language, high level language
   (b) imperative language, functional language, declarative language
   (c) COBOL, Fortran-77, C++
3. From the point of view of the programmer what are the major advantages of using a high-level language rather than internal machine code or assembler language?

(a) Program portability
(b) Easy development
(c) Efficiency

Lesson 3

Operating Systems

The **Operating System** of a computer is a large program which manages the overall operation of the computer system. On a simple one-user computer the Operating System will:

1. Provide an interface to allow the user to communicate with the computer. This interface may be a text-oriented interface where the user types commands in response to a prompt from the computer or may be a mouse-driven Windows operating system.

2. Control the various peripherals e.g. Keyboard, Video Display Unit (VDU), Printer etc. using special programs called Device Drivers.

3. Manage the user's files, keeping track of their positions on disk, updating them after user makes changes to them etc. An important facility that the Operating System must supply in this respect is an **Editor** which allows users to edit their files.

4. Provide system facilities, e.g. **Compilers** to translate from high-level programming languages used by the user to the internal machine language the computer uses.

Because of the disparity in speed between input/output devices (and the human entering data) and the CPU most modern operating systems will allow **Multi-tasking** to take place. Thus while the Computer is held up waiting for input from one program the operating system will transfer control to another program which can execute until it, in turn, is held up. Multi-tasking may take place in a stand-alone computer (for example using an operating system such as Windows 95 on a PC) and allow the user to simultaneously use several different programs simultaneously. For example a user may be running a large computational task in the background while using a word-processor package to write a report.

It is now common for computers to be linked together in **networks**. The network may consist of many dumb terminals, Personal Computers and workstations linked together with perhaps several larger, more powerful computers which provide a large amount of computer power and file storage facilities to the network. This allows many people access to computing facilities and access to common data-bases, electronic mail facilities etc. Networks may be local to a building or a small area (**Local Area Network (LAN)**) or connect individual networks across the country or world (**Wide Area Network (WAN)**).

A particular form of network operating system is a **Timesharing** operating system. Many large modern computers are set up to serve many simultaneous users by means of a time-sharing system. Each user has a direct connection to a powerful central computer, normally using a Visual Display Unit (VDU) which has a keyboard (and often a mouse) for user input and a screen for feedback from the computer to the user. There may be several hundred simultaneous users of a large computing system. Computing is **Interactive** in that the time from a user entering a command until a response is obtained will typically be a few seconds or less. The Operating System will cycle in turn through each connected terminal and if the terminal is awaiting computation will give it a **Time-slice** of dedicated CPU time. This process is continuous thus each program receives as many time-slices as it requires until it terminates and is removed from the list of programs awaiting completion.

In a system with multiple users the operating system must also carry out other tasks such as:

1. Validating the user's rights to use the system

2. Allocating memory and processor time to individual programs

3. Maintaining the security of each user's files and program execution

In a time-sharing system the processing power is contained in a central machine. Users access this central machine from a non-intelligent terminal that can do no processing itself. The advent of cheap powerful workstations has lead to the distribution of computer power around the network. A **distributed computer system** consists of a central processor with a large amount of disk storage and powerful input/output facilities connected to a network of machines, each with its own main memory and processor.

The central processor (or Server) provides storage for all system files and user files. Each computing node in the network downloads any files and system facilities it requires from the server and then carries out all computation internally. Any changes to files or new files generated have to be sent by the network to the server. To make it easier to find a particular file it is usual to collect all related files into a separate **directory**. Each user will be allocated a certain amount of space on the external memory, this space will be set up as a single directory called the user's **home directory**.

The user can further split this space into various other directories. For example a lecturer writing a course may well set up a directory to contain all the files relevant to the course. Within this directory it is best to organise the files into groups by setting up various sub-directories, a sub-directory to hold course notes, another to hold tutorials, another to hold laboratory sheets etc. Within one of these directories, say the tutorials directory, will be held the relevant files — tutorial1, tutorial2 etc. This hierarchical file storage structure is analogous to the storage of related files in a filing system. A filing cabinet could hold everything relevant to the course, each drawer could hold a different sub-division, such as notes, and each folder within the drawer would be a particular lecture.

Space will also be allocated on the server for system files. These also will be allocated to directories to facilitate access by the operating system.

3.1 Summary

- The operating system provides an interface between the user and the computer and controls the internal operation of the computer and its peripherals. It also manages the users' files and system utilities such as compilers, editors, application packages etc.
- Computer networks allow many users to simultaneously access computer facilities, to access common data-bases and to use facilities such as electronic mail.
- Files are stored in external storage in a hierarchical structure of directories. Each user will have their own home directory and the operating system facilities will be allocated to system directories.

3.2 Multiple Choice Questions

1. The Operating System is responsible for
   (a) Controlling peripheral devices such as monitor, printers, disk drives
   (b) Detecting errors in users' programs
   (c) Make the coffee
   (d) Provide an interface that allows users to choose programs to run and to manipulate files
   (e) Manage users' files on disk
2. Which of the following does an operating system do in a stand-alone computer system?

   (a) Manages the user's files.
   (b) Provides the system facilities.
   (c) Provides the interface to allow the user to communicate with the computer.
   (d) Controls the various peripherals

3. Which is the following is *TRUE* about a terminal on a **Time-sharing** computer system?

   (a) Has its own CPU and some memory.
   (b) Has no memory or CPU of its own.

4. Which is the following is *TRUE* about a terminal on a **Distributed** computer system?

   (a) Has its own CPU and some memory.
   (b) Has no memory or CPU of its own.

Lesson 4

Preparing a Computer Program

There are various steps involved in producing a computer program for a particular application. These steps are independent of which computer or programming language that is used and require the existence of certain facilities upon the computer. The steps are:

1. Study the **requirement specification** for the application. It is important that the requirements of the application should be well specified. Before starting to design a program for the application it is necessary that the requirement specification is complete and consistent. For example a requirement specification that says 'write a program to solve equations' is obviously incomplete and you would have to ask for more information on 'what type of equations?', 'how many equations?', 'to what accuracy?' etc.

2. Analyse the problem and decide how to solve it. At this stage one has to decide on a method whereby the problem can be solved, such a method of solution is often called an **Algorithm**.

3. Translate the algorithm produced at the previous step into a suitable high-level language. This written form of the program is often called the **source program** or **source code**. At this stage the program should be read to check that it is reasonable and a **desk-check** carried out to verify its correctness. A programmer carries out a desk-check by entering a simple set of input values and checking that the correct result is produced by going through the program and executing each instruction themselves. Once satisfied that the program is reasonable it is entered into the computer by using an **Editor**.

4. **Compile** the program into machine-language. The machine language program produced is called the **object code**. At this stage the com-

piler may find **Syntax** errors in the program. A syntax error is a mistake in the grammar of a language, for example C++ requires that each statement should be terminated by a semi-colon. If you miss this semi-colon out then the compiler will signal a syntax error. Before proceeding any syntax errors are corrected and compilation is repeated until the compiler produces an executable program free from syntax errors.

5. The object code produced by the compiler will then be linked with various function libraries that are provided by the system. This takes place in a program called a **linker** and the linked object code is then loaded into memory by a program called a **loader**.

6. Run the compiled, linked and loaded program with test data. This may show up the existence of **Logical** errors in the program. Logical errors are errors that are caused by errors in the method of solution, thus while the incorrect statement is syntactically correct it is asking the computer to do something which is incorrect in the context of the application. It may be something as simple as subtracting two numbers instead of adding them. A particular form of logical error that may occur is a **run-time error**. A run-time error will cause the program to halt during execution because it cannot carry out an instruction. Typical situations which lead to run-time errors are attempting to divide by a quantity which has the value zero or attempting to access data from a non-existent file.

   The program must now be re-checked and when the error is found it is corrected using the Editor as in (3) and steps (4) and (5) are repeated until the results are satisfactory.

7. The program can now be put into general use - though unless the testing was very comprehensive it is possible that at some future date more logical errors may become apparent. It is at this stage that good **documentation** produced while designing the program and writing the program will be most valuable, especially if there has been a considerable time lapse since the program was written.

4.1 Summary

- Before a computer program can be written the requirements of the application must be investigated and defined comprehensively and unambiguously . This leads to the production of the requirements specification.

- A precise set of instructions to solve a problem is called an **algorithm**.

The first step in writing a computer program is to produce an algorithm.

- Once the algorithm is designed it must be translated into a suitable high-level programming language. This program is then compiled to machine code, linked with any system libraries required and loaded into memory. At the compilation stage syntactic errors in the program may be found and have to be corrected before any further progress can be made.
- Once the program has been compiled, linked and loaded it can be tested with realistic test data. Testing may show up the presence of logical errors in the program. These may lead to the production of wrong results or cause the program to halt on a run-time error.

4.2 Multiple Choice Questions

1. A compiler produces an error in compiling a program because a closing round bracket has been missed out in the source code. What type of error is this?
   (a) Syntax Error
   (b) Logical Error
   (c) Linker Error

4.3 Review Questions

1. What are the steps involved in going from the specification of a problem to producing an executable program which will solve the problem?
2. What types of error can occur in computing and at what stages of the processes of producing a program do they occur?
3. When the program is running it produces a number that is too large to fit into the space allocated for it in memory. What type of error is this?
4. A program runs without any errors being reported but outputs results that are wrong, what type of error is likely to have caused this?

Lesson 5

Algorithms

Informally, an algorithm is a series of instructions which if performed in order will solve a problem. For an algorithm to be suitable for computer use it must possess various properties:

1. **Finiteness**: The algorithm must terminate after a finite number of steps. For example the algorithm:

   ```
   produce first digit of 1/7.
   while there are more digits of 1/7 do
       produce next digit.
   ```

   never terminates because 1/7 cannot be expressed in a finite number of decimal places.

2. **Non-ambiguity**: Each step must be precisely defined. For example the statement

   ```
   set k to the remainder when m is divided by n.
   ```

   is not precise because there is no generally accepted definition for what the remainder is when `m` is divided by `n` when `m` and `n` are negative. Different programming languages may well interpret this differently.

3. **Effectiveness**: This basically means that all the operations performed in the algorithm can actually be carried out, and in a finite time. Thus statements like 'if there are 5 successive 5's in the expansion of π then . . . ' may not be able to be answered.

Even if an algorithm satisfies the above criteria it may not be a practical way of solving a problem. While an algorithm may execute in a finite time it is not much use if that finite time is so large as to make solution completely impractical. Thus there is a lot of interest in finding 'good' algorithms which generate correct solutions in a short time compared with other algorithms.

In sorting 10,000 numbers into ascending order a ‘good’ algorithm executing on a PC took less than a second while a ‘poor’ algorithm took over 10 minutes.

5.1 Describing an Algorithm

A simple example is used to look at the problem of designing an algorithm in a suitable form for implementation on a computer. The simple computational problem considered is:

> Write a program to input some numbers and output their average.

This is a very informal specification and is not complete enough to define exactly what the program should do. For example where are the numbers going to come from—entered by the user from the keyboard or perhaps read from a file? Obviously to find the average of a series of numbers one adds them together to find their sum and then divides by the number of numbers in the series. So how is the number of numbers entered known? For example it could be input as part of the data of the program or the program could count the numbers as they are entered. This supposes that the program has some way of knowing when the user has stopped entering numbers or when the end of the file has been reached. Another important question is ‘what should the program do if no data is entered?’. It would then be nonsensical to print out an average value! Keeping these questions in mind leads to the following more specific requirement specification:

> Write a program which inputs a series of numbers and outputs their average. The numbers are supplied in a data file and are terminated by an end-of-file marker. In the event that the file is empty a message to that effect should be output.

An algorithm is now required to solve this problem. Obviously the algorithm could be described by ‘read in the numbers from the file, add them up and divide by the number of numbers to get the average’. In practice this algorithmic description is not sufficiently detailed to describe a computer algorithm. The computer can only carry out simple instructions like ‘read a number’, ‘add a number to another number’, etc. Thus an algorithm must be described in such simple terms. Consider the following first version of a solution:

```
1   carry out any initialisations required.
2   while not reached end of file do
3       {
4        read in next number.
5        add the number to the accumulated sum.
6        increment the count of numbers input.
7       }
8   evaluate the average.
```

Note that the line numbers are not required, they are merely for reference. It is commonplace in computer algorithms that certain **initialisations** have to be carried out before processing begins. Hence as a reminder that this might be necessary a phrase is inserted to indicate that initialisation may be required at the beginning of every algorithm description as in line 1. Once the rest of the algorithm has been developed this initialisation step can be expanded to carry out any initialisations that are necessary.

At line 2 a statement is used which describes a **loop**. This loop executes the statements contained between the brackets in lines 3–7 continuously as long as the **condition** 'not reached end of file' remains true. The brackets are used to show that the statements in lines 4, 5 and 6 are associated together and are executed as if they were one statement. Without the brackets only statement 4 would be executed each time round the loop. This is not the only **repetition statement** that may be used, others are possible and will be seen later. In the **body** of the loop, lines 4–6, each instruction is a simple executable instruction. Once the end of the file is reached then control transfers from the loop to the next instruction after the loop, evaluating the average at line 8. This requires more expansion, since, if there are no numbers input then the average cannot be evaluated. Hence line 8 is expanded to:

```
8a  if no numbers input
8b     then print message 'no input'
8c     otherwise {
8d                set average equal to the accumulated sum
8e                       divided by the number of numbers.
8f                print the average.
8g               }
```

Here a **conditional statement** has been used with the condition 'no numbers input', if this is true then the statement after the `then` is executed, while if it is not true the statement after the `otherwise` is executed. This process is often called a '**Selection**' process.

On studying the above algorithm it is obvious that the process carried out is what a human being might do if given a sheet of paper with many numbers on it and asked to find the average. Thus a person might run their finger down the numbers adding them as they go to an accumulated total until there were no more and then counting the number of numbers and dividing the total by this number to get the average. When this is done it is implicit in their reasoning that the total and the count will both start of at zero before starting to process the numbers. This is not obvious to the computer at all, hence it must be remembered in describing computer algorithms to initialise all sum accumulation variables and counters suitably before commencing processing. Also it must be ensured that processing starts at the beginning of the file. Hence the initialisation (line 1) can be expanded as follows:

```
1a  set accumulated sum to zero.
1b  set number count to zero.
1c  open file ready to read first data item.
```

Note that if there were no data items in the file then the first element in the file would be the end-of-file marker. Thus the 'while-loop' condition would be false initially so the loop would not be executed and the number of numbers would remain at zero. This would ensure that the condition at line 8a was true hence the appropriate message would be output.

The whole algorithm is now:

```
set accumulated total to zero.
set number count to zero.
open file ready to read first item.
while not at end of file do
    {
     read next number from file.
     add number to accumulated total.
     increment number count.
    }
if number count is zero
   then print message 'no input'
   otherwise {
              set average to accumulated total divided
                         by the number count.
              print average.
             }
```

5.2 Statements required to describe algorithms

Very few statement types have been used in describing the above algorithm. In fact these few statement types are sufficient to describe all computer algorithms. In practice other forms of loop may be introduced but they can all be implemented using the while loop used in the previous section. The following basic concepts are required:

1. The idea that statements are executed in the order they are written. **Sequence**.
2. A construct that allows the grouping of several statements together so that they can be handled as if they were one statement by enclosing them in brackets. **Compound Statement**.
3. A construct that allows the repetition of a statement or compound statement several times. **Repetition**.
4. A construct that allows the selection of the next statement to execute depending on the value of a condition. **Selection**.
5. The ability to assign a value to a quantity. **Assignment**.

All high-level programming languages have equivalents of such constructions and hence it would be very easy to translate the above algorithm into a computer program. In fact each statement of this algorithm can be written as a statement in the language C++.

If you compare the following program written in C++ with the algorithm you should be able to spot the similarities. Note that the program as written is not complete, it would require further declarations and definitions to be added before the compiler would accept it as syntactically correct. What is listed here are the executable statements of the program.

```
ins.open(infile);
total = 0;
count = 0;
while (!ins.eof())
  {
    ins >> number;
    total = total + number;
    count = count + 1;
  }
if (count == 0)
  {
    cout << "No input" << endl;
  }
else
  {
    average = total/count;
    cout << "Average is "
         << average
         << endl;
  }
ins.close(infile);
```

5.3 Verifying the correctness of the algorithm

Before proceeding to implement an algorithm as a program it should be checked for correctness. There are formal ways of doing this but informal methods are used here.

In considering the algorithm just developed the general case, i.e. there are numbers in the file, is checked first. After opening the file the current reading point is set at the first number. On entering the loop statement the current number is read and the reading point in the file is advanced to the next item. Ultimately the the end-of-file marker will be the next item to be read and the condition 'not at end of file' becomes false and exit is made from the loop. This shows that inside the loop numbers only are read from the file and no attempt is made to treat the end-of-file marker as a number. Hence all the numbers in the file are read but nothing more. By studying the body of the loop it is seen that every time a number is read it is added to the accumulated total and the count is incremented. Hence as both these quantities start of at zero and the body of the loop is executed for each number in turn the accumulated total and number count must be correct, hence the average must be correct.

Having checked the general case any boundary conditions should then be checked. In this case the only boundary condition is the case of the file being empty. In this case the count is initialised to zero, the body of the loop is never executed hence the number count remains at zero. The condition 'number count is zero' in the conditional statement is then true and the appropriate message is output.

This form of informal reasoning should be applied to check an algorithm before implementing it as a program.

5.3.1 Desk-checking

Another way of testing the logic of programs is to carry out a desk-check, that is execute the statements of the algorithm yourself on a sample data set. This method of course is not foolproof, you would have to be sure that you traversed all possible paths through your algorithm, this might require you to use many data sets. It is useful to use a tabular layout for this. For example say the initial data file was

```
     2 6 eof
```

then a tabular layout as follows could be used:

```
 file    number count total average

2 6 eof                            before initialisation

*
2 6 eof           0     0          after initialisation

  *
2 6 eof    2      1     2          after first loop execution

     *
2 6 eof    6      2     8          after second loop execution

     *
2 6 eof    6      2     8     4    exit loop, evaluate average
```

The * indicates the file element which is available for reading on the next input statement. Obviously this gives the correct result for this file. This simple file is adequate to check the general case here because there are only two paths through this algorithm. The other path is the case where the file is empty. This could be checked in the same way.

5.4 Series Minimum and Maximum Algorithm

Consider the following requirement specification:

A user has a list of numbers and wishes to find the minimum value and the maximum value in the list. A program is required which will allow the user to enter the numbers from the keyboard and which will calculate the minimum and maximum values that are input. The user is quite happy to enter a count of the numbers in the list before entering the numbers.

A first version at an algorithm for this might be:

```
initialise.
get count of numbers.
enter numbers and find maximum and minimum.
output results.
```

The algorithm is now made completely general to allow for a list with no numbers in it, this may seem a bit stupid but it is not uncommon when writing a general-purpose function to allow for the possibility of null input. A user might change their mind after calling the program for example and it is sensible that the program should respond sensibly if the user enters zero for the count. Incorporating this into the algorithm above gives the next version:

```
initialise.
get count of numbers.
if count is zero
   then exit
   otherwise {
              enter numbers and find maximum
                                   and minimum.
              output results.
             }
```

Once the count of the numbers is known then a loop has to be executed that number of times, each time reading in a number and somehow using that number in finding the maximum and minimum of the numbers. In this loop the number of times the loop is executed is known, i.e. equal to the count of numbers. Thus another type of repetition command is introduced:

```
loop n times
  {
   body of loop.
  }
```

This operation of repeating some set of instructions, the **body** of the loop, a set number of times occurs frequently in algorithm design. Note that braces have been used to make the body of the loop into a **compound**

statement. Each time the loop is executed it is the instructions between the braces that are executed.
Hence the following version of the algorithm:

```
initialise.
get count of numbers.
if count is zero
   then exit
   otherwise {
              loop count times
                {
                 enter a number.
                 process the number.
                }
              output results.
             }
```

It has not yet been considered how to compute the maximum and minimum values so this has been indicated by using the phrase 'process the number'. Given a large list of numbers written down on a sheet of paper how could the smallest number in the list be found in a methodical fashion? One way would be to start at the beginning of the list and work through the list systematically always remembering the smallest number seen so far, whenever a number is found smaller than the memorised number the memorised number is replaced by the smaller number. At the start of the list the smallest number yet seen is of course the first number, and when the end of the list is reached the memorised number is the smallest. Similarly for the largest number. Hence the following expansion:

```
get count of numbers.
if count is zero
   then exit
   otherwise {
              get first number.
              set small to number.
              set large to number.
              loop count-1 times
                {
                 enter a number.
                 if number is less than small
                    then set small to number.
                 if number is greater than large
                    then set large to number.
                }
              print small and large.
```

```
                }
```

In this algorithm no initialisation is required at the beginning, the only quantities that have to be initialised are small and large, and they are initialised to the first number entered by the user. Similarly count is entered by the user and requires no initialisation.

A brief reading of this should convince us that this algorithm is correct. If the user enters zero for the count then exit takes place immediately. If the user enters a non-zero count the first number is input, then the loop is executed count-1 times and one number is entered in each execution of the loop body. This means that 1 plus count-1 numbers are entered so all input numbers are considered. Large and small are initialised to the first number entered and each time round the loop are updated if the current number is larger or smaller than the memorised large or small. The boundary case where the count is zero has been considered but there is another boundary case here, namely count equal to one. If count is equal to 1 then large and small are set to the single number and the loop is executed zero times, hence when large and small are output they are set to the only number input, which is the correct result.

It is worth checking this algorithm by doing a simple desk check, say with the following data:

```
count = 5
numbers are 3 5 1 7 2
```

The different values taken during the algorithm are as follows:

```
count  number large small

 -       -      -     -     begin
 5       -      -     -     enter count
 5       3      3     3     enter first number
 5       5      5     3     first loop execution
 5       1      5     1     second loop execution
 5       7      7     1     third loop execution
 5       2      7     1     final loop execution
```

The symbol '-' has been used to indicate an unknown value. At the end of execution of the algorithm large and small do hold the correct maximum and minimum values

5.5 Summary

- An algorithm is a sequence of steps which will produce a solution to a problem.

- An algorithm must be **finite**, **non-ambiguous** and **effective**.
- The basic control structures used in specifying an algorithm are **sequence**, **selection** and **repetition**.
- A **compound statement** allows several statements to be grouped together as a single entity.
- The process of giving a value to an object is called **assignment**
- A **selection** control structure allows the next statement to be executed to be determined based on the value of some **condition**.
- A **repetition** control structure allows a statement or group of statements to be executed repeatedly.

5.6 Multiple Choice Questions

1. Which of the following are essential statement types for describing algorithms?
   (a) sequence
   (b) selection
   (c) repetition
2. What is a condition?
   (a) A value which is **true** or **false**
   (b) A numerical value

5.7 Review Questions

1. What are the types of control structure used in describing algorithms?
2. What is a condition?
3. Why may an algorithm be finite and yet impractical to use?
4. How would you endeavour to show that an algorithm you have produced is correct?

5.8 Exercises

1. A program is required which will read in the breadth and height of a rectangle and which will output the area and the length of the perimeter of the rectangle. Write an algorithm for this problem.

2. The requirement specified in question 1 is extended to include the requirement that if the entered breadth and height are equal then the output should be 'the area and perimeter of the square are ...' whereas if they are not equal then the output should be 'the area and perimeter of the rectangle are ...'. Alter the algorithm that you produced for question 1 to take account of this new requirement.

3. A series of positive numbers are to be entered from the keyboard with the end of the series indicated by a negative number. The computer should output the sum of the positive numbers. Write an algorithm for this task. Use a while type of loop with the condition 'number just entered is positive'. Think carefully about what initialisations are required before entering the while loop. Do a desk check of your algorithm with a small data set, say 3 positive numbers then a negative number. What would your algorithm do if the user entered a negative number first? Is what your algorithm would do in this circumstance sensible?

4. Extend the algorithm in the previous question so that it also counts the number of numbers entered in the series.

5. Another form of repetition statement is a **repeat** loop. This has the form:

   ```
   repeat
     statement 1.
         .
     statement n.
   until condition
   ```

   which executes the statements contained between repeat and until until the condition becomes true. For example to allow a user of a program the chance to re-run it with a new set of data a repeat loop might be used as follows:

   ```
   repeat
     enter and process data.
     ask if user wishes to process more data.
     read reply.
   until reply is no
   ```

Now extend your algorithm for question 2 so that it repeats entering the dimensions of rectangles and calculating the results. After each calculation the user should be asked if they wish to continue.

Lesson 6

A simple C++ program

Before looking at how to write C++ programs consider the following simple example program.

```
// Sample program
// IEA September 1995
// Reads values for the length and width of a rectangle
// and returns the perimeter and area of the rectangle.

#include <iostream.h>

void main()
{
  int length, width;
  int perimeter, area;              // declarations
  cout <<  "Length = ";             // prompt user
  cin >> length;                    // enter length
  cout << "Width = ";               // prompt user
  cin >> width;                     // input width
  perimeter = 2*(length+width);     // compute perimeter
  area = length*width;              // compute area
  cout << endl
       << "Perimeter is " << perimeter;
  cout << endl
       << "Area is " << area
       << endl;                    // output results
} // end of main program
```

rect_1.cpp

The following points should be noted in the above program:

1. Any text from the symbols `//` until the end of the line is ignored by the compiler. This facility allows the programmer to insert **Comments** in the program. Every program should at least have a comment indicating the programmer's name, when it was written and what the program actually does. Any program that is not very simple should also have further comments indicating the major steps carried out and explaining any particularly complex piece of programming. This is essential if the program has to be amended or corrected at a later date.

2. The line

   ```
   #include <iostream.h>
   ```

   must start in column one. It causes the compiler to include the text of the named file (in this case `iostream.h`) in the program at this point. The file `iostream.h` is a system supplied file which has definitions in it which are required if the program is going to use stream input or output. All your programs will include this file. This statement is a **compiler directive** — that is it gives information to the compiler but does not cause any executable code to be produced.

3. The actual program consists of the **function** `main` which commences at the line

   ```
   void main()
   ```

   All programs must have a function `main`. Note that the opening brace (`{`) marks the beginning of the body of the function, while the closing brace (`}`) indicates the end of the body of the function. The word `void` indicates that `main` does not return a value. Running the program consists of obeying the statements in the body of the function `main`.

4. The body of the function `main` contains the actual code which is executed by the computer and is enclosed, as noted above, in braces `{}`.

5. Every statement which instructs the computer to do something is terminated by a semi-colon. Symbols such as `main()`, `{ }` etc. are not instructions to do something and hence are not followed by a semi-colon.

6. Sequences of characters enclosed in double quotes are literal strings. Thus instructions such as

   ```
   cout << "Length = "
   ```

send the quoted characters to the output stream `cout`. The special identifier `endl` when sent to an output stream will cause a newline to be taken on output.

7. All variables that are used in a program must be declared and given a type. In this case all the variables are of type `int`, i.e. whole numbers. Thus the statement

   ```
   int length, width;
   ```

   declares to the compiler that integer variables `length` and `width` are going to be used by the program. The compiler reserves space in memory for these variables.

8. Values can be given to variables by the **assignment** statement, e.g. the statement

   ```
   area = length*width;
   ```

   evaluates the expression on the right-hand side of the equals sign using the current values of `length` and `width` and assigns the resulting value to the variable `area`.

9. Layout of the program is quite arbitrary, i.e. new lines, spaces etc. can be inserted wherever desired and will be ignored by the compiler. The prime aim of additional spaces, new lines, etc. is to make the program more readable. However superfluous spaces or new lines must not be inserted in words like `main`, `cout`, in variable names or in strings (unless you actually want them printed).

6.1 Variables

A variable is the name used for the quantities which are manipulated by a computer program. For example a program that reads a series of numbers and sums them will have to have a variable to represent each number as it is entered and a variable to represent the sum of the numbers.

In order to distinguish between different variables, they must be given **identifiers**, names which distinguish them from all other variables. This is similar to elementary algebra, when one is taught to write "Let a stand for the acceleration of the body ...". Here a is an identifier for the value of the acceleration. The rules of C++ for valid identifiers state that:

An identifier must:

- start with a letter
- consist only of letters, the digits 0–9, or the underscore symbol `_`

- not be a **reserved word**

Reserved words are otherwise valid identifiers that have special significance to C++. A full list is given below in section 6.1.1. For the purposes of C++ identifiers, the underscore symbol, `_`, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers. Similarly, the use of two consecutive underscore symbols, `__`, is forbidden.

The following are valid identifiers

```
length  days_in_year  DataSet1  Profit95
Int     _Pressure     first_one first_1
```

although using `_Pressure` is not recommended.
The following are invalid:

```
days-in-year   1data   int    first.val throw
```

Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using *meaningful* identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

At this stage it is worth noting that C++ is **case-sensitive**. That is lower-case letters are treated as distinct from upper-case letters. Thus the word `main` in a program is quite different from the word `Main` or the word `MAIN`.

6.1.1 Reserved words

The **syntax rules** (or grammar) of C++ define certain symbols to have a unique meaning within a C++ program. These symbols, the **reserved words**, must not be used for any other purposes. The reserved words already used are `int` and `void`. All reserved words are in lower-case letters. The table below lists the reserved words of C++.

C++ Reserved Words

`and`	`and_eq`	`asm`	`auto`	`bitand`
`bitor`	`bool`	`break`	`case`	`catch`
`char`	`class`	`const`	`const_cast`	`continue`
`default`	`delete`	`do`	`double`	`dynamic_cast`
`else`	`enum`	`explicit`	`export`	`extern`
`false`	`float`	`for`	`friend`	`goto`
`if`	`inline`	`int`	`long`	`mutable`
`namespace`	`new`	`not`	`not_eq`	`operator`
`or`	`or_eq`	`private`	`protected`	`public`
`register`	`reinterpret_cast`	`return`	`short`	`signed`
`sizeof`	`static`	`static_cast`	`struct`	`switch`
`template`	`this`	`throw`	`true`	`try`
`typedef`	`typeid`	`typename`	`union`	`unsigned`
`using`	`virtual`	`void`	`volatile`	`wchar_t`
`while`	`xor`	`xor_eq`		

Some of these reserved words may not be treated as reserved by older compilers. However you would do well to avoid their use. Other compilers may add their own reserved words. Typical are those used by Borland compilers for the PC, which add `near`, `far`, `huge`, `cdecl`, and `pascal`.

Notice that `main` is *not* a reserved word. However, this is a fairly technical distinction, and for practical purposes you are advised to treat `main`, `cin`, and `cout` as if they were reserved as well.

6.2 Declaration of variables

In C++ (as in many other programming languages) all the variables that a program is going to use must be **declared** prior to use. Declaration of a variable serves two purposes:

- It associates a **type** and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.

- It allows the compiler to decide how much storage space to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

For the moment only four variable types are considered, namely, **int**, **float**, **bool** and **char**. These types hold values as follows:

int variables can represent negative and positive integer values (whole numbers). There is a limit on the size of value that can be represented, which depends on the number of bytes of storage allocated to an `int` variable by the computer system and compiler being used. On a PC most compilers allocate two bytes for each `int` which gives a range of -32768 to +32767. On workstations, four bytes are usually allocated, giving a range of -2147483648 to 2147483647. It is important to note that **integers are represented exactly** in computer memory.

float variables can represent any real numeric value, that is both whole numbers and numbers that require digits after the decimal point. The accuracy and the range of numbers represented is dependent on the computer system. Usually four bytes are allocated for `float` variables, this gives an accuracy of about six significant figures and a range of about -10^{38} to $+10^{38}$. It is important to note that **float values are only represented approximately**.

bool variables can only hold the values **true** or **false**. These variables are known as *boolean* variables in honour of George Boole, an Irish mathematician who invented boolean algebra.

char variables represent a single character — a letter, a digit or a punctuation character. They usually occupy one byte, giving 256 different possible characters. The bit patterns for characters usually conform to the American Standard Code for Information Interchange (ASCII).

Examples of values for such variables are:

```
int    123      -56    0      5645

float  16.315  -0.67   31.567

char   '+'     'A'     'a'    '*'    '7'
```

A typical set of variable declarations that might appear at the beginning of a program could be as follows:

```
int i, j, count;
float sum, product;
char ch;
bool passed_exam;
```

which declares integer variables `i`, `j` and `count`, real variables `sum` and `product`, a character variable `ch`, and a boolean variable `pass_exam`.
A variable declaration has the form:

type identifier-list;

type specifies the type of the variables being declared. The *identifier-list* is a list of the identifiers of the variables being declared, separated by commas.

Variables may be initialised at the time of declaration by assigning a value to them as in the following example:

```
int i, j, count = 0;
float sum = 0.0, product;
char ch = '7';
bool passed_exam = false;
```

which assigns the value 0 to the integer variable `count` and the value 0.0 to the real variable `sum`. The character variable `ch` is initialised with the character `7`. `i`, `j`, and `product` have no initial value specified, so the program should make *no* assumption about their contents.

6.3 Constants and the declaration of constants

Often in programming numerical constants are used, e.g. the value of π. It is well worthwhile to associate meaningful names with constants. These names can be associated with the appropriate numerical value in a **constant declaration**. The names given to constants must conform to the rules for the formation of identifiers as defined above. The following constant declaration

```
const int days_in_year = 365;
```

defines an integer constant `days_in_year` which has the value 365. Later in the program the identifier `days_in_year` can be used instead of the integer 365, making the program far more readable.

The general form of a constant declaration is:

`const` *type constant-identifier* `=` *value* `;`

type is the type of the constant, *constant-identifier* is the identifier chosen for the constant, which must be distinct from all identifiers for variables, and *value* is an expression involving only constant quantities that gives the constant its value. It is not possible to declare a constant without giving it an initial value.

Another advantage of using constant declarations is illustrated by the following declaration:

```
const float VatRate = 17.5;
```

This defines a constant `VatRate` to have the value 17.5, however if the Government later changes this rate then instead of having to search through the program for every occurrence of the VAT rate all that needs to be done is to change the value of the constant identifier `VatRate` at the one place in the program. This of course only works if the constant identifier `VatRate` has been used throughout the program and its numeric equivalent has never been used.

Constant definitions are, by convention, usually placed before variable declarations. There is no limit on how many constant declarations can be used in a program. Several constant identifiers **of the same type** can be declared in the same constant declaration by separating each declaration by a comma. Thus

```
const int days_in_year = 365,
          days_in_leap_year = 366;
```

Note that it is illegal in C++ to attempt to change the value of a constant.

6.4 General form of a C++ Program

At this stage the programs considered will fit into the following general format:

```
// Introductory comments
// file name, programmer, when written or modified
// what program does

#include <iostream.h>

void main()
{
   constant declarations
   variable declarations
   executable statements
}
```

Note that it makes complex programs much easier to interpret if, as above, closing braces `}` are aligned with the corresponding opening brace `{`. However other conventions are used for the layout of braces in textbooks and other C++ programmers' programs. Also additional spaces, new lines etc. can also be used to make programs more readable. The important thing is to adopt one of the standard conventions and stick to it consistently.

6.5 Input and Output

Input and output use the **input stream** `cin` and the **output stream** `cout`. The input stream `cin` is usually associated with the keyboard and the output stream `cout` is usually associated with the monitor.

The following statement waits for a number to be entered from the keyboard and assigns it to the variable `number`:

```
cin >> number;
```

The general form of a statement to perform input using the input stream `cin` is:

`cin` *input-list*;

where *input-list* is a list of identifiers, each identifier preceded by the **input operator** `>>`. Thus

```
cin >> n1 >> n2;
```

would take the next two values entered by the user and assign the value of the first one to the variable `n1` and the second to the variable `n2`.

The program must read a value for each variable in the input-list before it executes any more statements. The order in which the values are entered must correspond to the order of the variables in the input-list and they must be of the same type as the corresponding variable. They should be separated by spaces. Normally, the C++ system will not pass any values to the variables in the input-list until a complete line of input has been read, i.e. until the **return** or **enter** key has been pressed. If more values are supplied than are required to give each variable in the input-list a value, the unused values will be used for any subsequent input statements using `cin`. For example given the following declarations and input statement:

```
int count, n;
float value;
cin >> count >> value >> n;
```

the user could enter

```
  23  -65.1 3
```

to assign 23 to `count`, -65.1 to `value` and 3 to `n`. There is no indication in the data of which value is to be associated with which variable; the order of the data items must correspond to the order of the variables in the input list. The data items on input should be separated by spaces or new lines. Any number of these will be skipped over before or between data items. Thus the input above could equally well have been entered as:

```
     23
-65.1        3
```

The following statement outputs the current value of the variable `count` to the output stream `cout`, which is usually associated with the monitor. The value will be printed on the current line of output starting immediately after any previous output.

```
cout << count;
```

The general form of a statement to perform output using the output stream `cout` is:

`cout` *output-list*;

where *output-list* is a list of variables, constants, or character strings in quotation marks, each preceded by the **output operator `<<`**. The output operator displays the value of the item that follows it. The values are displayed in the order in which they appear in the output-list. A new line is taken if the special end-of-line character `endl` is output. If an `endl` is not output, the output line will either be chopped off at the right hand edge of the screen or it may wrap round on to the next line. Do not rely on either behaviour as different computer systems may do things differently. Thus

```
cout << "Hello there" << endl;
```

will print `Hello there` on the current output line and then take a new line for the next output. The statements:

```
float length, breadth;
cout << "Enter the length and breadth: ";
cin >> length >> breadth;
cout << endl << "The length is " << length;
cout << endl << "The breadth is " << breadth << endl;
```

will display, if the user enters 6.51 and 3.24 at the prompt, the following output:

```
The length is 6.51
The breadth is 3.24
```

Note that a value written to `cout` will be printed immediately after any previous value with no space between. In the above program the character strings written to `cout` each end with a space character. The statement

```
cout << length << breadth;
```

would print out the results as

```
6.513.24
```

which is obviously impossible to interpret correctly. If printing several values on the same line remember to separate them with spaces by printing a string in between them as follows:

```
cout << length << " " << breadth;
```

6.6 Programming Style

As was remarked in note 6.4 above, any number of spaces and or new lines can be used to separate the different symbols in a C++ program. The identifiers chosen for variables mean nothing to the compiler either, but using identifiers which have some significance to the programmer is good practice. The program below is identical to the original example in this Lesson, except for its layout and the identifiers chosen. Which program would you rather be given to modify?

```
#include <iostream.h>
void main(
) { int a,b,
c,d;  cout <<  "Length = ";  cin >> a; cout<<"Width = "
;cin >> b; c = 2*(a+
  b);     d = a*b;  cout
<< endl << "Perimeter is "  <<
c << endl << "Area is " << d
       << endl;}
```

6.7 Summary

- An international standard for the C++ language is soon to be produced. This will make programs written in standard obeying C++ capable of being transported from one computer to another.
- All C++ programs must include a function `main()`.
- All executable statements in C++ are terminated by a semi-colon.
- Comments are ignored by the compiler but are there for the information of someone reading the program. All characters between `//` and the end of the line are ignored by the compiler.
- All variables and constants that are used in a C++ program must be declared before use. Declaration associates a type and an identifier with a variable.
- The type `int` is used for whole numbers which are represented exactly within the computer.
- The type `float` is used for real (decimal) numbers. They are held to a limited accuracy within the computer.
- The type `char` is used to represent single characters. A `char` constant is enclosed in single quotation marks.

- Literal strings can be used in output statements and are represented by enclosing the characters of the string in double quotation marks `"`.
- Variables names (identifiers) can only include letters of the alphabet, digits and the underscore character. They must commence with a letter.
- Variables take values from input when included in input statements using `cin >>` *variable-identifier*.
- The value of a variable or constant can be output by including the identifier in the output list `cout <<` *output-list*. Items in the output list are separated by `<<`.

6.8 Multiple Choice Questions

1. Which of the following are valid C++ identifiers?
   (a) `const`
   (b) `y=z`
   (c) `xyz123`
   (d) `Bill`
   (e) `ThisIsALongOne`
   (f) `Sue's`
   (g) `two-way`
   (h) `int`
   (i) `so_is_this_one`
   (j) `_amount`
   (k) `2ndclass`
2. Values can be input to variables in the program using the stream
   (a) `cin`
   (b) `cout`
3. Comments in C++ are started with
   (a) `//`
   (b) `{`
   (c) `;`

6.9 Review questions

1. Write a constant declaration that declares constants to hold the number of days in a week and the number of weeks in a year. In a separate constant statement declare a constant `pi` as 3.1415927.

2. Write declaration statements to declare integer variables `i` and `j` and float variables `x` and `y`. Extend your declaration statements so that `i` and `j` are both initialised to 1 and `y` is initialised to 10.0.

3. Write C++ instructions to ask a user to type in three numbers and to read them into integer variables `first`, `second` and `third`.

4. Write C++ instructions to output the value of a variable `x` in a line as follows:

   ```
   The value of x is ......
   ```

5. Write C++ instructions to generate output as follows:

   ```
   A circle of radius .....
             has area .....
    and circumference .....
   ```

   where the values of the radius, the area and the circumference are held in variables `rad`, `area`, and `circum`.

6. Correct the syntax errors in the following C++ program:

   ```
   include iostream.h

   Main();
   {
    Float x,y,z;
    cout < "Enter two numbers ";
    cin >> a >> b
    cout << 'The numbers in reverse order are'
         << b,a;
   }
   ```

   syntax.cpp

7. Show the form of output displayed by the following statements when `total` has the value 352.74.

   ```
   cout << "The final total is: " << endl;
   cout << "$" << total << endl;
   ```

8. What data types would you use to represent the following items?

(a) the number of students in a class
(b) the grade (a letter) attained by a student in the class
(c) the average mark in a class
(d) the distance between two points
(e) the population of a city
(f) the weight of a postage stamp
(g) the registration letter of a car

9. Write suitable declarations for variables in question 8. Be sure to choose meaningful identifiers.

6.10 Exercises

1. Using literal character strings and `cout` print out a large letter `E` as below:

```
XXXXX
X
X
XXX
X
X
XXXXX
```

2. Write a program to read in four characters and to print them out, each one on a separate line, enclosed in single quotation marks.

3. Write a program which prompts the user to enter two integer values and a float value and then prints out the three numbers that are entered with a suitable message.

Lesson 7

The Assignment statement

The main statement in C++ for carrying out computation and assigning values to variables is the **assignment statement**. For example the following assignment statement:

```
average = (a + b)/2;
```

assigns half the sum of `a` and `b` to the variable `average`. The general form of an assignment statement is:

result = *expression* ;

The *expression* is evaluated and then the value is assigned to the variable *result*. It is important to note that the value assigned to *result* must be of the same type as *result*.

The *expression* can be a single variable, a single constant or involve variables and constants combined by the arithmetic operators listed below. Rounded brackets `()` may also be used in matched pairs in expressions to indicate the order of evaluation.

`+` addition
`-` subtraction
`*` multiplication
`/` division
`%` remainder after division (modulus)

For example

```
i = 3;
sum = 0.0;
perimeter = 2.0 * (length + breadth);
ratio = (a + b)/(c + d);
```

The type of the operands of an arithmetic operator is important. The following rules apply:

- if both operands are of type **int** then the result is of type **int**.
- if either operand, or both, are of type **float** then the result is of type **float**.
- if the expression evaluates to type **int** and the result variable is of type **float** then the `int` will be converted to an equivalent `float` before assignment to the result variable.
- if the expression evaluates to type **float** and the result variable is of type **int** then the `float` will be converted to an `int`, usually by rounding towards zero, before assignment to the result variable.

The last rule means that it is quite easy to lose accuracy in an assignment statement. As already noted the type of the value assigned must be the same type as the variable to which it is assigned. Hence in the following example in which `i` is a variable of type `int`

```
i = 3.5;
```

the compiler will insert code to convert the value 3.5 to an integer before carrying out the assignment. Hence the value 3 will be assigned to the variable `i`. The compiler will normally truncate float values to the integer value which is nearer to zero. Rounding to the nearest integer is *not* carried out.

A similar problem arises with the division operator. Consider the following rule:

- the result of a division operator between two **int** operands is of type **int**. It gives the result truncated towards zero if the result is positive, the language does not define what should happen if the result is negative, so beware! This of course means that it is very easy to lose accuracy if great care is not taken when using division with integer variables.

For example the statement

```
i = 1/7;
```

will assign the value zero to the integer variable `i`. Note that if the quotient of two integers is assigned to a float then the same loss of accuracy still occurs. Even if `i` in the above assignment was a variable of type `float` `1/7` would still be evaluated as an integer divided by an integer giving zero, which would then be converted to the equivalent `float` value, i.e. 0.0, before being assigned to the `float` variable `i`.

The modulus operator `%` between two positive integer variables gives the remainder when the first is divided by the second. Thus `34 % 10` gives 4 as the result. However if either operand is negative then there are ambiguities since it is not well-defined in C++ what should happen in this case. For example `10 % -7` could be interpreted as 3 or -4. Hence it is best to avoid this situation. All that C++ guarantees is that

```
i % j = i - (i / j) * j
```

7.1 Priority of Operators

Another problem associated with evaluating expressions is that of order of evaluation. Should

```
a + b * c
```

be evaluated by performing the multiplication first, or by performing the addition first? i.e. as

`(a + b) * c` or as `a + (b * c)` ?

C++ solves this problem by assigning priorities to operators, operators with high priority are then evaluated before operators with low priority. Operators with equal priority are evaluated in left to right order. The priorities of the operators seen so far are, in high to low priority order:

```
( )
* / %
+ -
=
```

Thus

```
a + b * c
```

is evaluated as if it had been written as

```
a + (b * c)
```

because the `*` has a higher priority than the `+`. If the `+` was to be evaluated first then brackets would need to be used as follows:

```
(a + b) * c
```

If in any doubt use extra brackets to ensure the correct order of evaluation.

It is also important to note that two arithmetic operators cannot be written in succession, use brackets to avoid this happening.

7.2 Examples of Arithmetic Expressions

The following examples illustrate how some more complex mathematical expressions can be written in C++.

Mathematical	C++ Equivalent
$\dfrac{a+b}{c+d}$	`(a+b)/(c+d)`
$\dfrac{a}{bc}$	`a/(b*c)`
$\dfrac{ka(t_1-t_2)}{b}$	`k*a*(t1-t2)/b`

7.3 Type Conversions

The rules stated above mean that division of integers will always give an integer result. If the correct float result is required, then the compiler must be forced to generate code that evaluates the expression as a `float`. If either of the operands is a constant, then it can be expressed as a floating point constant by appending a `.0` to it, as we have seen. Thus assuming that `n` is an `int` variable, `1/n` does not give the correct reciprocal of `n` except in the situation `n=1`. To force the expression to be evaluated as a floating point expression, use `1.0/n`.

This solves the problem when one of the operands is a constant, but to force an expression involving two `int` variables to be evaluated as a `float` expression, at least one of the variables must be converted to `float`. This can be done by using the **cast** operation:

```
    f = float(i)/float(n);
```

The type `float` is used as an operator to give a floating point representation of the variable or expression in brackets. Notice that `f = float(i/n);` will still evaluate the expression as an `int` and only convert it to `float` after the integer division has been performed.

Other types can be used to cast values too. `int(x)` will return the value of `x` expressed as an `int`. Similarly, `char(y)` will return the character corresponding to the value `y` in the ASCII character set.

7.4 Example Program: Temperature Conversion

The following program converts an input value in degrees Fahrenheit to the corresponding value in degrees Centigrade. Note how the constant `mult` has been defined using an expression. A constant can be defined using an expression as long as the operands in the expression are numeric constants or the names of constants already defined. Also note that the constant has been given the value 5.0/9.0, if it had been defined by 5/9 then this would have evaluated to zero (an integer divided by an integer) which is not the intention.

```
// Convert Fahrenheit to Centigrade
// Enters a Fahrenheit value from the user,
// converts it to centigrade and outputs
// the result.

#include <iostream.h>

void main()
{
  const float mult = 5.0/9.0;  // 5/9 returns zero
                               // integer division
  const int sub = 32;
  float fahr, cent;
  cout << "Enter Fahrenheit temperature: ";
  cin >> fahr;
  cent = (fahr - sub) * mult;
  cout << "Centigrade equivalent of " << fahr
       << " is " << cent << endl;
}
```

ftoc.cpp

7.5 Example Program: Pence to Pounds and Pence

The following program converts an input value in pence to the equivalent value in pounds and pence. Note how integer division has been used to find the whole number of pounds in the value of pence by dividing pence by 100. Also how the % operator has been used to find the remainder when pence is divided by 100 to produce the number of pence left over.

```
// Convert a sum of money in pence into the equivalent
// sum in pounds and pence.

#include <iostream.h>

void main()
{
  int pence, pounds;
  cout << "Enter the amount in pence: ";
```

```
    cin >> pence;
    cout << pence << " pence is ";
    pounds = pence / 100; // note use of integer division
    pence = pence % 100;  // modulus operator -> remainder
    cout << pounds << " pounds and "
         << pence << " pence" << endl;
}
```

ptolp.cpp

7.6 Summary

- Expressions are combinations of operands and operators.
- The order of evaluation of an expression is determined by the precedence of the operators.
- In an assignment statement, the expression on the right hand side of the assignment is evaluated and, if necessary, converted to the type of the variable on the left hand side before the assignment takes place.
- When `float` expressions are assigned to `int` variables there may be loss of accuracy.

7.7 Multiple Choice Questions

1. Evaluate `4+5*3`
   (a) `19`
   (b) `27`
2. Evaluate `7*3+2`
   (a) `23`
   (b) `35`
3. Evaluate `(4+2)*3`
   (a) `18`
   (b) `10`
4. Evaluate `17/3`
   (a) `5`

   (b) `6`

5. Evaluate `17%3`
   (a) `0`
   (b) `1`
   (c) `2`
   (d) `3`

6. Evaluate `1/2`
   (a) `0`
   (b) `1`

7. Evaluate `2*8/2*4`
   (a) `32`
   (b) `2`

7.8 Review questions

1. Write C++ expressions for the following mathematical formulae:

$$b^2 - 4ac \quad \frac{a^2+1}{bc} \quad \frac{1}{1+x^2} \quad a * -(b+c)$$

2. Write C++ statements to change an integer number of centimetres into the equivalent in kilometres, metres and centimetres. For example 164375 centimetres is 1 kilometre, 643 metres and 75 centimetres. Include declarations of suitable variables.

3. To what do the following expressions evaluate?

```
17/3  17%3  1/2  1/2*(x+y)
```

4. Given the declarations:

```
float x;
int k, i = 5, j = 2;
```

   To what would the variables `x` and `k` be set as a result of the assignments

   - `k = i/j;`
   - `x = i/j;`
   - `k = i%j;`
   - `x = 5.0/j;`

7.9 Exercises

1. Write a C++ program which reads values for two floats and outputs their sum, product and quotient. Include a sensible input prompt and informative output.

2. Write a program to evaluate the fuel consumption of a car. The mileage at the start and end of the journey should be read, and also the fuel level in the tank at the start and end of the journey. Calculate fuel used, miles travelled, and hence the overall fuel consumption in miles travelled per gallon of fuel.

3. In many countries using the metric system, car fuel consumptions are measured in litres of fuel required to travel 100 kilometres. Modify your solution to question 2 so that the output now specifies the distance travelled in kilometres as well as in miles, and the fuel consumed in litres as well as in gallons, and the consumption in litres per 100 kilometres as well as in miles per gallon. Use `const` for the conversion factors between miles and kilometres, and gallons and litres.

4. Write a program to convert currency from pounds sterling to deutsch marks. Read the quantity of money in pounds and pence, and output the resulting foreign currency in marks and pfennigs. (There are 100 pfennigs in a mark). Use a `const` to represent the conversion rate, which is 2.31DM to £1 at the time of writing. Be sure to print suitable headings and or labels for the values to be output.

5. Modify your answer to question 4 so that a commission of £2 is charged.

6. A customer's gas bill is calculated by adding a standard charge of 9.02 pence per day to the charge for gas, calculated as 1.433 pence per cubic metre. The whole bill is then liable for VAT at 8%. Write a program that reads the number of days, and the initial and final meter readings, and calculates and prints the bill. Use `const` for the various rates of charging.

7. If three integers a, b and c are such that $a^2 + b^2 = c^2$ then they constitute a Pythagorean triple. There is an infinite number of such triples. One way of generating them is as follows:
   Consider two integers m, and n, such that $m > n$ then the three numbers $m^2 - n^2$, $2mn$ and $m^2 + n^2$ are a Pythagorean triple. Write a C++ program that reads values for m and n and prints the values of the corresponding Pythagorean triple.

Lesson 8

Further Assignment Statements & Control of Output

8.1 Increment and Decrement Operators

There are some operations that occur so frequently in writing assignment statements that C++ has shorthand methods for writing them.

One common situation is that of incrementing or decrementing an integer variable. For example:

```
n = n + 1;
n = n - 1;
```

C++ has an **increment** operator `++` and a **decrement** operator `--`. Thus

`n++;` can be used instead of `n = n + 1;`
`n--;` can be used instead of `n = n - 1;`

The `++` and `--` operators here have been written after the variable they apply to, in which case they are called the **postincrement** and **postdecrement** operators. There are also identical **preincrement** and **predecrement** operators which are written before the variable to which they apply. Thus

`++n;` can be used instead of `n = n + 1;`
`--n;` can be used instead of `n = n - 1;`

Both the pre- and post- versions of these operators appear to be the same from the above, and in fact it does not matter whether `n++` or `++n` is used if all that is required is to increment the variable `n`. However both versions of the increment and decrement operators have a side effect which means that they are not equivalent in all cases. These operators as well as incrementing or decrementing the variable also return a value. Thus it is possible to write

```
i = n++;
```

What value does `i` take? Should it take the old value of `n` before it is incremented or the new value after it is incremented? The rule is that a postincrement or postdecrement operator delivers the old value of the variable before incrementing or decrementing the variable. A preincrement or predecrement operator carries out the incrementation first and then delivers the new value. For example if `n` has the value 5 then

```
i = n++;
```

would set `i` to the original value of `n` i.e. 5 and would then increment `n` to 6. Whereas

```
i = ++n;
```

would increment `n` to 6 and then set `i` to 6.

For the moment this notation will only be used as a shorthand method of incrementing or decrementing a variable.

8.2 Specialised Assignment Statements

Another common situation that occurs is assignments such as the follows:

```
sum = sum + x;
```

in which a variable is increased by some amount and the result assigned back to the original variable. This type of assignment can be represented in C++ by:

```
sum += x;
```

This notation can be used with the arithmetic operators `+`, `-`, `*`, `/` and `%`. The general form of such *compound assignment operators* is:

variable op= *expression*

which is interpreted as being equivalent to:

variable = *variable op* (*expression*)

the *expression* is shown in brackets to indicate that the expression is evaluated first before applying the operator *op*. The following example illustrate the use of compound assignment operators.

```
total += value;    or      total = total + value;
prod *= 10;        or      prod = prod * 10;
x /= y + 1;        or      x = x/(y + 1);
n %= 2;            or      n = n % 2;
```

Except for the case of the compound modulus operator `%=` the two operands may be any arithmetic type. The compound modulus operator requires that both operands are integer types.

8.3 Formatting of output

When considering output in 6.5 no consideration was given to the format of the output produced. It was assumed that all output would be formatted using the default settings. The default settings print integers using as many characters as are required and real values are printed with up to six decimal digits (some compilers give six digits after the decimal point).

Consider the following portion of C++. This portion uses a construct not covered yet, namely, a `for` statement. The `for` statement is used for implementing loops and will be covered later. The statement starting with `for` has the effect of executing the statements between the braces `{}` as `i` takes the values 1, 2, 3 and 4.

```
for (i=1; i<5; i++)
  {
   cout << "Enter an integer value: ";
   cin >> x;
   cout << x << "   " << sqrt(x) << endl;
  }
```

then output as follows might be produced:

```
1   1
5   2.23607
1678   40.9634
36   6
```

This is very untidy and difficult to read, it would be preferable if it could appear as follows:

```
    1   1.00000
    5   2.23607
 1678  40.96340
   36   6.00000
```

with the least significant digits of the integers aligned and the decimal points in the real numbers aligned. It is possible to achieve this degree of control on the output format by using **output manipulators**.

Before looking at manipulators **scientific notation** for the display of floating point numbers is considered. Scientific notation allows very large or very small numbers to be written in a more convenient form. Thus a number like 67453000000000000 is better written as 6.7453×10^{16} and a number like 0.0000000000001245 is better written as 1.245×10^{-13}. C++ allows this type of notation by replacing the 'ten to the power of' by e or E. Thus the above numbers could be written in C++ as 6.7453e16 and 1.245e-13. These forms can be used in writing constants in C++ and in input and output. On output, if a number is too large to display in six digits then scientific notation will be used by default. For example 12345678.34 might be output as 1.23457e+07.

The first manipulator considered is **setiosflags**, this allows the output of floating point numbers to be

fixed	fixed format i.e. no scientific notation
scientific	scientific notation
showpoint	displays decimal point and trailing zeros

The flags that are to be set are specified in the `setiosflags` manipulator as follows:

```
setiosflags(ios::flagname)
```

If more than one flag is to be set then another `ios::`*flagname* can be included, separated by a `|` from the other setting in the above call. Thus the following output statement would set fixed format with the decimal point displayed:

```
cout << setiosflags(ios::fixed | ios::showpoint);
```

This would ensure that a number like 1.0 would be displayed as 1.0 rather than as 1. These flags **remain in effect** until explicitly changed.

Another useful manipulator is the **setprecision** manipulator, this takes one parameter which indicates the number of decimal places of accuracy to be printed. This accuracy **remains in effect** until it is reset. The `setprecision` manipulator may be used when none of the iosflags have been set. However there is some confusion over what constitutes precision, some compilers will produce n digits in total and others n digits after the point when `setprecision(`n`)` is used on its own. However if it is used after the flags `fixed` or `scientific` have been set it will produce n digits after the decimal point.

For the moment the most suitable setting of the iosflags for output are `fixed` and `showpoint`.

The following portion of C++

```
float x, y;
x = 12.2345,
y = 1.0;
cout << setiosflags(ios::fixed | ios::showpoint)
     << setprecision(2);
cout << x << endl
     << y << endl;
```

would output

```
12.23
1.00
```

that is, in `fixed` format with two places after the point and the point displayed. Without the `ios` flag set to `showpoint` `y` would have been printed as 1. If the decimal points have to be aligned then the field width has to be set. This is done using the **setw** manipulator which takes a single parameter which indicates the width of field in which the output value is to be placed. The value is placed **right-justified** in the field. The field width remains **in effect only for the next data item displayed**. Thus if the lines:

```
cout << setw(7) << x << endl
     << setw(7) << y << endl;
```

were added to the above portion of code the output would be:

```
12.23
1.00
  12.23
   1.00
```

Note 1: The file `iomanip.h` must be included if the above manipulators are to be used. There are many more facilities available by using input/output manipulators but the above is enough to allow the writing of programs that produce sensible formatting of output in most cases.

Note 2: The output width is reset to the default after every variable is output, so that it was necessary to use `setw(7)` *twice*, once before each variable that was output.

8.4 Example Program: Tabulation of sin function

The following example program tabulates values of the sin function, using manipulators to align the output neatly in columns. The `for` statement, which will be covered in Lesson 18, repeats the statements after it. In this case, `i` takes values 0, 1, 2, ... 16.

```
// IEA Oct 1995
// Outputs a table of x and sin(x)
// Uses manipulators for output

#include <iostream.h>
#include <math.h>
#include <iomanip.h>

void main()
{
 float x;
 int i;
 cout << setiosflags(ios::fixed | ios::showpoint);
 for (i = 0; i <= 16; i++ )
   {
    x = 0.1 * i;
    cout << setprecision(1) << setw(4) << x;
    cout << setprecision(6) << setw(10) << sin(x) << endl;
   }
}
```

sin_1.cpp

produces nicely aligned tabular output as follows:

```
 0.0  0.000000
 0.1  0.099833
 0.2  0.198669
 0.3  0.295520
  .
  .
  .
 1.5  0.997495
 1.6  0.999574
```

Note how the `iosflags` were set at the beginning but that the precision and width were set individually in the `cout` stream output as required.

8.5 Summary

- The unary increment and decrement operators are applied to integer variables to increase or decrease the value by 1.
- If the increment (or decrement) operator is placed after the variable, the operation takes place after the value has been returned.

- If the increment (or decrement) operator is placed before the variable, the operation takes place before the value is returned.
- The operators `+`, `-`, `*`, `/`, and `%` can all be used in the form

  variable op `=` *expression*

  which is identical in operation to

  variable `=` *variable op* `(` *expression* `)`
- I/O manipulators can be used to control the format of output. The file `iomanip.h` must be included in the program if they are to be used.

8.6 Multiple Choice Questions

1. Consider the following section of C++ program, in which `i` and `n` are `int` variables

   ```
   n = 7;
   i = 4;
   i = n++;
   ```

   What are the values of `i` and `n`?

   (a) `i=7 n=8`
   (b) `i=7 n=7`
   (c) `i=8 n=8`
   (d) `i=4 n=7`

2. Consider the following section of C++ program, in which `i` and `n` are `int` variables

   ```
   n = 5;
   i = 9;
   i = --n;
   ```

   What are the values of `i` and `n`?

   (a) `i=9 n=5`
   (b) `i=4 n=4`
   (c) `i=4 n=5`
   (d) `i=5 n=4`

8.7 Exercises

1. Alter the program you wrote for exercise 2 in Lesson 7 so that the printed values for mileage and tank contents have two digits printed after the decimal point and the consumption in miles per gallon is printed with one digit after the decimal point.

2. Amend your program for exercise 6 in Lesson 7 so that the output appears in a format as follows:

```
Standing charge  9.02 pence for  61 days    5.50
Gas consumption 1632.1 cu.m @ 1.433 pence  23.39
Sub-total                                  28.89

VAT at 8%                                   2.31

Total                                      31.20
```

   Note that allowance has been made for future increases in the standing charge to more than 10 pence per day, and for the number of days to be anything up to 999. You may wish to consider suitable limits on the amount of gas consumed, in order that the format of you final output is not disrupted.

Lesson 9

Introduction to structured design

9.1 Conditional Control Structures

In Section 5.2 it was shown that algorithms can be described using a few concepts, namely, sequence, conditional execution (or selection) and repetition. The idea of assignment was also used.

In designing programs it is best to proceed with algorithm/program design in a **top-down structured manner**, that is by first recognising the major components of the solution, then expressing the solution as a sequence of these major components and then expanding the major components themselves similarly. These ideas have already been illustrated in Lesson 5. They are further expanded below.

Consider the example C++ program that computes the area and perimeter of a rectangle (see Lesson 6) when the length and width are entered by a user. A possible initial algorithm is as follows, where each step is numbered:

```
1. Enter the length and width of the rectangle.

2. Calculate the area and perimeter of the rectangle.

3. Output the area and the perimeter.
```

Notice that the idea of **sequence** has been used here, that is that the operations are carried out in the order they are written. This is the simplest way of structuring the solution to a problem. Each step in the above is now expanded as follows:

```
1. Enter
   1.1 Prompt user to enter length
   1.2 Enter length
   1.3 Prompt user to enter width
```

```
      1.4 Enter width

   2. Calculate
      2.1 Calculate perimeter as twice sum
          of length and width
      2.2 Calculate area as product of length and width

   3. Output
      3.1 Output perimeter
      3.2 Output area
```

At this stage the problem has now been completely solved independent of the language the program is to be written in. It is now simple to write the program in any suitable programming language. In fact in Lesson 6 this program was given as an example and Lesson 7 covered enough C++ to allow this program to be written in C++.

Unfortunately not all problems are so simple to solve. Frequently the simple idea of sequence used above is insufficient to describe the solution of many problems.
Consider the following problem:

> Write a program which enters the number of hours worked in a week and an hourly rate of pay of an employee. The program should output the wage of the employee. The employee is paid at the normal hourly rate for the first forty hours and subsequently at one and a half times the hourly rate.

The problem solution now appears fairly straightforward and modelling it on the previous case an algorithm is written as follows:

```
   1. Enter the hours worked and the hourly rate.

   2. Calculate the wage.

   3. Output the wage.
```

However in attempting to expand step 2, the simple idea of sequence is not sufficient. This is because if the number of hours worked is less than or equal to forty then the final wage is the number of hours multiplied by the hourly rate whereas if more than forty hours are worked then it is the hourly rate for the first forty hours and one and a half times the hourly rate for the remaining hours. Thus the truth of a **condition** 'number of hours less than or equal to forty' determines which calculation should be carried out. Such a **conditional** statement has already been encountered in Lesson 5. There it was introduced as being a statement which tests a condition and depending on the result of the test carries out one operation or another.

Thus using a conditional statement the algorithmic solution above could be expanded as follows:

```
1. Enter
   1.1 Prompt user for hours worked.
   1.2 Enter hours worked.
   1.3 Prompt user for hourly rate.
   1.4 Enter hourly rate.

2. Calculate wage
   2.1 If hours worked is less than or equal to forty
       then
          2.1.1 calculate normal wage.
       otherwise
          2.1.2 calculate over hours wage.
3. Output the wage
```

The details of working out the wage are not important here, what is important is that in describing the solution a conditional statement was used. Conditional statements are often characterised by the words

if condition **then** A **else** B

which carries out the process A if the condition evaluates to true and otherwise carries out the process B. All programming languages have some form of conditional statement. The conditional statements available in C++ are considered in the following few lessons.

9.2 Summary

- The solution of a problem is usually best approached in a top-down manner. Sub problems are identified and solutions are developed in turn for each sub problem.
- A condition is a statement which can be either false or true.
- Conditional statements carry out one operation or another depending on the truth value of a condition.

9.3 Review Questions

9.4 Exercises

1. In question 6 of Lesson 7 you wrote a program to process a gas bill. If you haven't already done so write down an algorithm for this program.

Now extend the algorithm so that the option of paying the bill by installments is given. Thus as well as entering the numeric data the user answers a question as to whether they wish to pay the bill in full or by installments. If paid in installments then a five percent interest charge is added to the total bill before VAT is charged. An initial twenty percent must be paid initially and the remainder is paid in three equal monthly installments. The output should indicate the initial payment and monthly installment.

2. A program is required as part of an invoicing system to enter the reference number of a product, its unit price in pounds and pence and a quantity ordered. The program should output the total cost of the order given that the first 100 items ordered are charged at the unit price and that the remainder are charged at 75% of unit price. Write an algorithm for this program.

Lesson 10

Conditions

10.1 Relational Expressions

A condition or **logical expression** is an expression that can only take the values **true** or **false**. A simple form of logical expression is the **relational expression**. The following is an example of a relational expression:

```
x < y
```

which takes the value **true** if the value of the variable `x` is less than the value of the variable `y`.

The general form of a relational expression is:

operand1 relational-operator operand2

The *operand*s can be either variables, constants or expressions. If an operand is an expression then the expression is evaluated and its value used as the operand. The *relational-operators* allowable in C++ are:

`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`==`	equals
`!=`	not equals

Note that equality is tested for using the operator `==` since `=` is already used for assigning values to variables.

The condition is **true** if the values of the two operands satisfy the relational operator, and **false** otherwise.

10.2 Examples using Relational Operators

```
i < 10
total <= 1000.0
```

```
count != n
discriminant < 0.0
x * x + y * y < r*r
```

Obviously, depending on the values of the variables involved, each of the above relational expressions is **true** or **false**. For example if `x` has the value 3, `y` is 6, and `r` is 10, the last expression above evaluates to **true**, whereas if `x` was 7 and `y` was 8 then it would evaluate to **false**.

The value of a logical expression can be stored in a `bool` variable for later use. Any numerical expression can be used for the value of a condition, with 0 being interpreted as **false** and any non zero value as **true**.

This means that the value returned by a relational expression could be used in arithmetic. This is often done by programmers but it is a practice not to be recommended. It leads to a program that is difficult to understand.

10.3 Logical Expressions

It is possible to specify more complex conditions than those which can be written using only the relational operators described above. Since the value of a condition has a numerical interpretation it could be operated on by the usual arithmetic operators, this is not to be recommended. There are explicit **logical operators** for combining the logical values **true** and **false**.

The simplest logical operator is **not** which is represented in C++ by `!`. It operates on a single operand, and returns **false** if its operand is **true** and **true** if its operand is **false**.

The operator **and**, represented by `&&`, takes two operands and is **true** only if both of the operands are **true**. If either operand is **false**, the resulting value is **false**.

or is the final logical operator and is represented by `||`. It results in **true** if either of its operands is **true**. It returns **false** only if *both* its operands are **false**.

The logical operators can be defined by **truth tables** as follows. Note that F is used for **false** and T is used for **true** in these tables.

		and `&&`			**or** `||`		
not `!`		`A`	`B`	`A && B`	`A`	`B`	`A || B`
`A`	`!A`	F	F	F	F	F	F
F	T	F	T	F	F	T	T
T	F	T	F	F	T	F	T
		T	T	T	T	T	T

These tables show that **not** reverses the truth value of the operand, that the **and** of two operands is only true if both operands are true and that the **or** of two operands is true if either or both of its operands are true. Using these logical operators more complex conditions can now be written.

If `i` has the value 15, and `j` has the value 10, then the expression `(i > 10) && (j > 0)` is evaluated by evaluating the relation `i > 10` (which is **true**), then evaluating the relation `j > 0` (which is also **true**), to give **true**. If `j` has the value -1 then the second relation would be **false**, so the overall expression would be **false**. If `i` has the value 5, then the first relation would be **false** and the expression will be **false** irrespective of the value of the second relation. C++ does not even evaluate the second relation in this situation. Similarly, if the first relation is **true** in an **or** (`||`) expression then the second relation will not be evaluated. This **short-circuit evaluation** enables many logical expressions to be efficiently evaluated.

10.4 Examples using logical operators

```
(i < 10)  &&  (j > 0)
((x + y) <= 15) || (i == 5)
!((i >= 10) || (j <= 0))
(i < 10) && 0
```

Note that in the last example an actual truth value (0 - false) was used as one of the operands of `&&`, this means that whatever the value of `i` this logical expression evaluates to **false** (Why?). In these examples brackets have been used to make the order of application of operators clear. However, in the main, they are not strictly necessary if the precedence rules already considered for arithmetic operators are extended to include relational and logical operators. The consequent extended **Operator Precedence Table** for C++ is:

```
highest - evaluate first

()              brackets
!  +  -         logical not, unary plus, unary minus
*  /  %         multiply, divide, modulus
+  -            add, subtract
< <= > >=       less than, less than or equal,
                greater than, greater than or equal
== !=           equal, not equal
&&              logical and
||              logical or
=               assignment

lowest - evaluate last
```

Be careful not to confuse the assignment operator = with the logical equality operator ==.

Using this table with the following expression

```
x + y < 10 && x/y == 3 || z != 10
```

shows that the operators are evaluated in the order `/`, `+`, `<`, `==`, `!=`, `&&` and `||`. This is equivalent to bracketting the expression as follows:

```
((((x + y) < 10) && ((x/y) == 3)) || (z != 10))
```

Similarly the expressions written in bracketted form above could be written without brackets as:

```
i < 10  &&  j > 0
x + y <= 15 || i == 5
!(i >= 10 || j <= 0)
i < 10 && 0
```

Now that logical expressions (or conditions) in C++ have been covered it is possible to move on and look at the conditional control structures in C++.

10.5 Summary

- A condition or *logical expression* is an expression that can only take the values **false** or **true**.
- A relational expression is constructed from arithmetic expressions combined by the relational operators less than, greater than, equal, not equal, greater than or equal and less than or equal.
- A logical expression is constructed from relational expressions by use of the logical operators **not**, **and** and **or**.
- C++ evaluates only as many operands as necessary to find the value of a logical expression involving **and** or **or**.

10.6 Multiple Choice Questions

1. What values can a relational expression take?
   (a) **true** and **false**
   (b) Any numerical value
2. What values does C++ use to represent **true** and **false**?
   (a) 1 and 0
   (b) Any numerical value

3. If `a` is 5, `b` is 10, `c` is 15 and `d` is 0 what are the truth values of the following expressions?

   (a) `c == a+b`
   (b) `a != 7`
   (c) `b <=a`
   (d) `a > 5`
   (e) `a+d >= c-b`
   (f) `d/a < c*b`

4. If `a` is 5, `b` is 10, `c` is 15 and `d` is 0 what are the truth values of the following expressions?

   (a) `c == a+b || c == d`
   (b) `a != 7 && c >= 6 || a+c <= 20`
   (c) `!(b <= 12) && a % 2 == 0`
   (d) `!(a >5) || c < a+b`

10.7 Review Questions

1. What are the relational operators?

2. Write a relational expression which would evaluate to **true** if the sum of variables `x` and `y` was equal to the value of a variable `z`.

3. Bracket the following logical expressions to show the order of evaluation of the operators. Hence if `a` is 5, `b` is 10, `c` is 15 and `d` is 0 what are the truth values of the expressions?

```
c == a+b
a != 7
b <= a
a > 5
a+d >= c-b
d/a < c*b
```

4. Bracket the following logical expressions to show the order of evaluation of the operators. Hence if `a` is 5, `b` is 10, `c` is 15 and `d` is 0 what are the truth values of the expressions?

```
c == a+b || c == d
a != 7 && c >= 6 || a+c <= 20
!(b <= 12) && a % 2 == 0
!(a >5) || c < a+b
```

5. Write a logical expression which returns **true** if a `float` variable `x` lies between -10.0 and 10.0.

Lesson 11

The `if` statement

As remarked in section 5.2, in order to produce algorithms, it must be possible to select the next statement to execute on the basis of some condition. Simple conditions have been covered in Lesson 10. The `if` statement is the simplest form of conditional or selection statement in C++. The following `if` statement

```
if (x > 0.0)
   cout << "The value of x is positive";
```

will print out the message ‘ `The value of x is positive`’ if `x` is positive.

The general form of the `if` statement is:

```
if (condition)
  statement
```

where *condition* is any valid logical expression as described in Lesson 10.1 or a `bool` variable. The *statement* can be a single C++ statement of any kind and must be terminated by a semi-colon. It can also be a *compound statement*, which is a sequence of statements enclosed in left and right braces and acts as a single statement. The closing right brace is not followed by a semi-colon.

11.1 Examples of `if` statements

The following `if` statement adds `x` to a variable `sum` if `x` is positive:

```
if (x > 0.0)
  sum += x;
```

The following `if` statement also adds `x` to `sum` but in addition it adds 1 to a count of positive numbers held in the variable `poscount`:

```
if (x >= 0.0)
  {
    sum += x;
    poscount++;
  }
```

Note the use of the addition/assignment operator, and of the increment operator. Note how in the second example a compound statement has been used to carry out more than one operation if the condition is **true**. If this had been written as follows:

```
if (x >= 0.0)
   sum += x;
   poscount++;
```

then if `x` was greater than zero the next statement would be executed, that is `x` would be added to `sum`. However the statement incrementing `poscount` would then be treated as the next statement in the program, and not as part of the `if` statement. The effect of this would be that `poscount` would be incremented every time, whether `x` was positive or negative.

The statements within a compound statement can be any C++ statements. In particular, another `if` statement could be included. For example, to print a message if a quantity is negative, and a further message if no overdraft has been arranged:

```
if ( account_balance < 0 )
  {
    cout << "Your account is overdrawn.  Balance "
         << account_balance << endl;
    if ( overdraft_limit == 0 )
      cout << "You have exceeded your limit. << endl;
  }
```

In this case, the same effect could have been achieved using two `if` statements, and a more complex set of conditions:

```
if ( account_balance < 0 )
  cout << "Your account is overdrawn.  Balance "
       << account_balance << endl;
if ( account_balance < 0 && overdraft_limit == 0 )
  cout << "You have exceeded your limit. << endl;
```

11.2 Summary

- An `if` statement is used to execute a statement only if a condition is **true**.

- Compound statements executed because an `if` condition is **`true`** can contain *any* other C++ statement, including other `if` statements.

11.3 Multiple Choice Questions

1. If `y` has the value 5 what will be the value of the variable `y` after the following piece of C++ is executed?

   ```
   if (y > 0)
       y += 2;
   ```

   (a) 5

   (b) 7

   (c) 2

2. If `p` has the value 3 and `max` has the value 5, what will be the value of the variable `max` after the following piece of C++ is executed?

   ```
   if (p > max)
       max = p;
   ```

   (a) 5

   (b) 3

3. If `x` has the value 5.0 what will be the value of the variable `countneg` after the following piece of C++ is executed?

   ```
   countneg = 0;
   if (x < 0.0)
       negsum = negsum + x;
       countneg = countneg + 1;
   ```

   (a) 0

   (b) 1

   (c) 5

4. If this is changed as follows what would be the value of `countneg` after execution of the code?

   ```
   countneg = 0;
   if (x < 0.0)
      {
       negsum = negsum + x;
       countneg = countneg + 1;
      }
   ```

(a) 0

(b) 1

(c) 5

11.4 Exercises

In each of these exercises produce an algorithmic description before proceeding to write the program.

1. Modify your solution to question 2 from Lesson 7 so that the message

   ```
   Trip too short for accurate results
   ```

   is printed if the mileage used to calculate the fuel consumption is less than 100 miles.

2. Write a C++ program which when two integers `x` and `y` are input will output the absolute difference of the two integers. That is whichever one of `(x-y)` or `(y-x)` is positive. Think of all the cases that can arise and consequently plan a set of test data to confirm the correctness of your program.

3. Percentage marks attained by a student in three exams are to be entered to a computer. An indication of Pass or Fail is given out after the three marks are entered. The criteria for passing are as follows:

   A student passes if all three examinations are passed. Additionally a student may pass if only one subject is failed and the overall average is greater than or equal to 50. The pass mark for an individual subject is 40.

   Write a C++ program to implement this task.

Lesson 12

The `if-else` Statement

A simple `if` statement only allows selection of a statement (simple or compound) when a condition holds. If there are alternative statements, some which need to be executed when the condition holds, and some which are to be executed when the condition does not hold. This can be done with simple `if` statements as follows:

```
if (disc >= 0.0)
  cout << "Roots are real";
if (disc < 0.0 )
  cout << "Roots are complex";
```

This technique will work so long as the statements which are executed as a result of the first `if` statement do not alter the conditions under which the second `if` statement will be executed. C++ provides a direct means of expressing this selection. The `if-else` statement specifies statements to be executed for both possible logical values of the condition in an `if` statement.

The following example of an `if-else` statement writes out one message if the variable `disc` is positive and another message if `disc` is negative:

```
if (disc >= 0.0)
  cout << "Roots are real";
else
  cout << "Roots are complex";
```

The general form of the `if-else` statement is:

```
if ( condition )
  statementT
else
  statementF
```

If the *condition* is **true** then *statementT* is executed, otherwise *statementF* is executed. Both *statementF* and *statementT* may be single statements or compound statements. Single statements must be terminated with a semi-colon.

12.1 Examples of `if-else` statements

The following `if-else` statement adds `x` to a sum of positive numbers and increments a count of positive numbers if it is positive. Similarly if `x` is negative it is added to a sum of negative numbers and a count of negative numbers is incremented.

```
if (x >= 0.0)
  {
    sumpos += x;
    poscount++;
  }
else
  {
    sumneg += x;
    negcount++;
  }
```

12.2 Example Program: Wages Calculation

In Lesson 5 an algorithm was developed to calculate wages depending on hours worked and on whether any overtime had been worked. This can now be written in C++. The program is listed below:

```
// IEA 1996
// Program to evaluate a wage

#include <iostream.h>

void main()
{
  const float limit = 40.0,
              overtime_factor = 1.5;
  float hourly_rate,   // hourly rate of pay
        hours_worked,  // hours worked
        wage;          // final wage
    // Enter hours worked and hourly rate
  cout << "Enter hours worked: ";
  cin >> hours_worked;
  cout << "Enter hourly_rate: ";
  cin >> hourly_rate;
    // calculate wage
  if (hours_worked <= limit)
    wage = hours_worked * hourly_rate;
  else
    wage = (limit + (hours_worked - limit) * overtime_factor)
             * hourly_rate;
    // Output wage
  cout << "Wage for " << hours_worked
       << " hours at " << hourly_rate
       << " is " << wage
       << endl;
}
```

wages.cpp

Note that this program contains the minimal amount of comment that a program should contain. Comments have been used to:

- indicate who wrote the program, when it was written and what it does.
- describe the main steps of the computation.
- indicate what the program variables represent.

Also note how constants have been used for the number of hours at which the overtime weighting factor applies and the weighting factor itself. Hence if subsequent negotiations change these quantities the program is easily changed.

12.3 Example Program: Pythagorean Triples

In exercise 7 of Lesson 7 it was required to write a program to input two integer values m and n, where $m > n$ and to output the corresponding Pythagorean triple $m^2 - n^2$, $2mn$ and $m^2 + n^2$. This is now extended so that the values of m and n entered by the user are **validated** to ensure that m is greater than n. A suitable algorithmic description is:

```
enter values for m and n.
if m is greater than m
   then
       {
        calculate the pythagorean numbers
                             from m and n.
        output the pythagorean numbers.
       }
   otherwise output a warning message.
```

This algorithmic description is now easily converted into the following C++ program:

```
// IEA 1996
// Program to produce pythagorean triples
// with input validation.

#include <iostream.h>

void main()
{
  int m, n;       // entered by user to generate triple
  int t1, t2, t3; // The values of the triple
     // input from user
  cout << "Input values for m and n, m > n : ";
  cin >> m >> n;
     // now validate m and n
  if (m > n)
    {
      t1 = m*m-n*n;
      t2 = 2*m*n;
      t3 = m*m+n*n;
      cout << "The triple corresponding to "
           << m << " and " << n << " is "
           << t1 << " " << t2 << " " << t3
           << endl;
    }
  else
    cout << "m must be greater than n!"
         << endl
         << "you entered m as " << m
         << " and n as " << n
         << endl;
}
```

pyth_1.cpp

Note that the values of `m` and `n` entered by the user are printed as part of the output. This is good practice. Programs should not only display results but should give some indication of the data that produced the results. In this case the input data set was produced in full since it was small. In situations where the input data set was large it might not be realistic to reproduce it all but an indication such as

```
Results produced from Data Set No 23
```

might be output. This is vital if a listing of results is to mean anything at a future date.

12.4 Example Program: Area and Perimeter of Rectangle

Exercise 2 of Lesson 5 required an algorithm which given two values for the breadth and height of a rectangle would output the area and perimeter of the rectangle. However depending on whether the breadth and height were equal or not different messages would be output indicating whether it was a rectangle or a square. A suitable algorithm for this would be

```
enter values for breadth and height.
evaluate perimeter.
evaluate area.
if breadth is equal to height
    then
        output 'area and perimeter of square are '
    otherwise
        output 'area and perimeter of rectangle are'.
output area and perimeter.
```

This algorithm is then easily converted into a C++ program as follows:

```
// IEA 1996
// Calculates area and perimeter of a rectangle
// after input of breadth and height. Distinguishes
// a square from a rectangle.

#include <iostream.h>

void main()
{
  int breadth, height; // of rectangle
  int perimeter, area; // of rectangle
    // input breadth and height
  cout << "Enter breadth and height: ";
  cin >> breadth >> height;
    // calculate perimeter and area
  perimeter = 2*(breadth+height);
  area = breadth*height;
  if (breadth == height)
    cout << "Area and perimeter of square are ";
  else
    cout << "Area and perimeter of rectangle are ";
   // output area and perimeter
  cout << area << " " << perimeter
       << endl;
}
```

rect_2.cpp

Note how portions of the algorithmic description have been used as comments within the program. Remember that successive values sent to the output stream `cout` will each be printed immediately after the previous output value. Hence in the program above the printing of the actual values for the area and perimeter will be printed directly after the information string on the same line. If a new line is required then send the end of line marker `endl` to `cout`.

12.5 Summary

- An `if-else` statement is used to choose which of two alternative statements to execute depending on the truth value of a condition.

12.6 Multiple Choice Questions

1. If `y` has the value 5 what will be the value of the variable `y` after the following piece of C++ is executed?

   ```
   if (y > 0)
       y += 2;
   else
       y = 3;
   ```

   (a) 5
   (b) 7
   (c) 3
   (d) 2

2. Consider the following section of code:

   ```
   if ( base < height )
       cout << "Top heavy structure" << endl;
   else
       cout << "Stable structure" << endl;
   ```

   Which of the following possible values for `base` and `height` cause the message `Stable structure` to be printed?

   (a) `base` = 5, `height` = 3
   (b) `base` = 8, `height` = 8
   (c) `base` = 2, `height` = 7

12.7 Review Questions

1. If `x` has the value 3.5 when the following statement is executed what value would be assigned to `y`?

   ```
   if (x + 1 <= 3.6)
      y = 1.0;
   else
      y = 2.0;
   ```

2. In words what do you think the effect of the following statement is intended to be? Why is the statement syntactically incorrect? How could it be changed to be syntactically correct? Has the change that you have made ensured that the statement actually carries out the logical effect you stated at the beginning?

   ```
   if (x >= y)
      sum += x;
      cout << "x is bigger" << endl;
   else
      sum += y;
      cout << "y is bigger" << endl;
   ```

3. Write an `if-else` statement which would add a variable `x` to a variable `possum` if `x` is positive and would add `x` to `negsum` if the variable `x` was negative.

4. Expand the solution to the previous question so that if `x` is positive then a variable `poscount` is incremented and if `x` is negative a variable `negcount` is incremented. If this was part of a program what would be sensible initialisations to carry out?

12.8 Exercises

In each of these exercises produce an algorithmic description before proceeding to write the program.

1. Write a C++ program which when two integers `x` and `y` are input will output the absolute difference of the two integers. That is whichever one of `(x-y)` or `(y-x)` is positive. Think of all the cases that can arise and consequently plan a set of test data to confirm the correctness of your program.

2. Write a C++ program which will compute the area of a square ($area = side^2$) or a triangle ($area = \frac{base * height}{2}$) after prompting the user to type the first character of the figure name (`t` or `s`).

3. Percentage marks attained by a student in three exams are to be entered to a computer. An indication of Pass or Fail is given out after the three marks are entered. The criteria for passing are as follows:

   A student passes if all three examinations are passed. Additionally a student may pass if only one subject is failed and the overall average is greater than or equal to 50. The pass mark for an individual subject is 40.

   Write a C++ program to implement this task.

Lesson 13

Nested `if` and `if-else` statements

The `if-else` statement allows a choice to be made between two possible alternatives. Sometimes a choice must be made between more than two possibilities. For example the sign function in mathematics returns -1 if the argument is less than zero, returns +1 if the argument is greater than zero and returns zero if the argument is zero. The following C++ statement implements this function:

```
if (x < 0)
   sign = -1;
else
   if (x == 0)
      sign = 0;
   else
      sign = 1;
```

This is an `if-else` statement in which the statement following the `else` is itself an `if-else` statement. If `x` is less than zero then `sign` is set to -1, however if it is not less than zero the statement following the `else` is executed. In that case if `x` is equal to zero then `sign` is set to zero and otherwise it is set to 1.

Novice programmers often use a sequence of `if` statements rather than use a nested `if-else` statement. That is they write the above in the logically equivalent form:

```
if (x < 0)
   sign = -1;
if (x == 0)
   sign = 0;
if (x > 0)
   sign = 1;
```

This version is not recommended since it does not make it clear that only one of the assignment statements will be executed for a given value of `x`. Also it is inefficient since all three conditions are always tested.

If nesting is carried out to too deep a level and indenting is not consistent then deeply nested `if` or `if-else` statements can be confusing to read and interpret. It is important to note that an `else` always belongs to the closest `if` without an `else`.

When writing nested `if-else` statements to choose between several alternatives use some consistent layout such as the following:

```
if ( condition1 )
  statement1 ;
else if ( condition2 )
  statement2 ;
   . . .
else if ( condition-n )
  statement-n ;
else
  statement-e ;
```

Assume that a real variable `x` is known to be greater than or equal to zero and less than one. The following multiple choice decision increments `count1` if $0 \leq$ `x` < 0.25, increments `count2` if $0.25 \leq$ `x` < 0.5, increments `count3` if $0.5 \leq$ `x` < 0.75 and increments `count4` if $0.75 \leq$ `x` < 1.

```
if (x < 0.25)
  count1++;
else if (x < 0.5)
  count2++;
else if (x < 0.75)
  count3++;
else
  count4++;
```

Note how the ordering of the tests here has allowed the simplification of the conditions. For example when checking that `x` lies between 0.25 and 0.50 the test `x < 0.50` is only carried out if the test `x < 0.25` has already failed hence `x` is greater than 0.25. This shows that if `x` is less than 0.50 then `x` must be between 0.25 and 0.5.

Compare the above with the following clumsy version using more complex conditions:

```
if (x < 0.25)
  count1++;
else if (x >= 0.25 && x < 0.5)
  count2++;
```

```
else if (x >= 0.5 && x < 0.75)
  count3++;
else
  count4++;
```

13.1 Summary

- Nested `if` and `if-else` statements can be used to implement decisions which have more than two outcomes.
- In nested `if-else` statements each `else` is associated with the nearest preceding `if` which has no `else` already associated with it.

13.2 Multiple Choice Questions

1. What is the final value of `x` if initially `x` has the value 1?

   ```
   if (x >= 0)
      x += 5;
   else if (x >=5)
           x += 2;
   ```

   (a) 8
   (b) 6
   (c) 1

2. What is the final value of `x` if initially `x` has the value 0?

   ```
   if (x >= 0)
     x += 5;
   if (x >= 5)
     x += 2;
   ```

   (a) 7
   (b) 5
   (c) 0

13.3 Review Questions

1. It is decided to base the fine for speeding in a built up area as follows - 50 pounds if speed is between 31 and 40 mph, 75 pounds if the speed is between 41 and 50 mph and 100 pounds if he speed is above 50 mph. A programmer writing a program to automate the assessment of fines produces the following statement:

```
if (speed > 30)
   fine = 50;
else if (speed > 40)
   fine = 75;
else if (speed > 50)
   fine = 100;
```

Is this correct? What fine would it assign to a speed of 60 mph? If incorrect how should it be written?

2. Write a nested `if-else` statement that will assign a character grade to a percentage mark as follows - 70 or over A, 60-69 B, 50-59 C, 40-49 D, 30-39 E, less than 30 F.

13.4 Exercises

1. Extend the mark-processing exercise 3 in Lesson 12 as follows. Print out the student's average mark (to the nearest integer) and also include a grade A to F alongside the average mark as defined in review question 2 above.

Lesson 14

The switch statement

In the last Lesson it was shown how a choice could be made from more than two possibilities by using nested `if-else` statements. However a less unwieldy method in some cases is to use a `switch` statement. For example the following `switch` statement will set the variable `grade` to the character A, B or C depending on whether the variable `i` has the value 1, 2, or 3. If `i` has none of the values 1, 2, or 3 then a warning message is output.

```
switch (i)
  {
    case 1 :  grade = 'A';
              break;
    case 2 :  grade = 'B';
              break;
    case 3 :  grade = 'c';
              break;
    default : cout << i
                   << " not in range";
              break;
  }
```

The general form of a switch statement is:

```
switch ( selector )
  {
    case label1  : statement1 ;
                   break;
    case label2  : statement2 ;
                   break;
          . . .
    case labeln  : statementn ;
                   break;
    default : statementd ;      // optional
```

```
                      break;
      }
```

The *selector* may be an *integer* or *character* variable or an expression that evaluates to an integer or a character. The selector is evaluated and the value compared with each of the *case labels*. The case labels must have the *same type* as the selector and they must all be *different*. If a match is found between the selector and one of the case labels, say *labeli* , then the statements from the statement *statementi* until the next `break` statement will be executed. If the value of the selector cannot be matched with any of the case labels then the statement associated with `default` is executed. The `default` is optional but it should only be left out if it is certain that the selector will always take the value of one of the case labels. Note that the statement associated with a case label can be a single statement or a sequence of statements (without being enclosed in curly brackets).

14.1 Examples of switch statements

The following statement writes out the day of the week depending on the value of an integer variable `day`. It assumes that day 1 is Sunday.

```
switch (day)
  {
   case 1 : cout << "Sunday";
            break;
   case 2 : cout << "Monday";
            break;
   case 3 : cout << "Tuesday";
            break;
   case 4 : cout << "Wednesday";
            break;
   case 5 : cout << "Thursday";
            break;
   case 6 : cout << "Friday";
            break;
   case 7 : cout << "Saturday";
            break;
  default : cout << "Not an allowable day number";
            break;
  }
```

If it has already been ensured that `day` takes a value between 1 and 7 then the `default` case may be missed out. It is allowable to associate several case labels with one statement. For example if the above example is amended to write out whether `day` is a weekday or is part of the weekend:

```
switch (day)
  {
    case 1 :
    case 7 : cout << "This is a weekend day";
             break;
    case 2 :
    case 3 :
    case 4 :
    case 5 :
    case 6 : cout << "This is a weekday";
             break;
    default : cout << "Not a legal day";
              break;
  }
```

Remember that missing out a `break` statement causes control to fall through to the next case label — this is why for each of the days 2–6 ‘This is a weekday’ will be output. Switches can always be replaced by nested `if-else` statements, but in some cases this may be more clumsy. For example the weekday/weekend example above could be written:

```
if (1 <= day && day <= 7)
  {
    if (day == 1 || day == 7)
      cout << "This is a weekend day";
    else
      cout << "This is a weekday";
  }
else
  cout << "Not a legal day";
```

However the first example becomes very tedious—there are eight alternatives! Consider the following:

```
if (day == 1)
    cout << "Sunday";
else if (day == 2)
    cout << "Monday";
else if (day == 3)
    cout << "Tuesday";
        .
        .
else if (day == 7)
    cout << "Saturday";
else
    cout << "Not a legal day";
```

14.2 Summary

- A switch statement selects the next statement to be executed from many possible statements. The selection is made depending on the value of a selector variable which can only be an integer or a character.
- If the selector variable does not match any of the `case` labels then the statements associated with the `default` label will be executed.
- The `default` label is optional but if it is not included then a selector which does not match any of the `case` labels causes the whole `switch` statement to be ignored.

14.3 Multiple Choice Questions

1. What is the value of the variable `c` after the `switch` statement below?

```
x = 3;
switch ( x ) {
  case 1: c = 'A'; break;
  case 2: c = 'B'; break;
  case 3: c = 'C'; break;
  default: c = 'F'; break;
}
```

(a) `A`
(b) `B`
(c) `C`
(d) `F`

14.4 Review Questions

1. A student is given a grade from 'A' to 'F' in a test. So averages can be calculated it is required to assign marks to these grades as follows, 'A' is 10, 'B' is 8, 'C' is 6, 'D' is 4, 'E' is 2 and 'F' is 0. In C++ what two methods are there of doing this? Which of these two methods would be better in this case? Write the appropriate statement for the method you have chosen.

2. A student has a percentage mark for an examination, these marks are to be converted to grades as follows:

```
>=70   'A'
60-69  'B'
```

50-59	'C'
40-49	'D'
30-39	'E'
<30	'F'

Could a switch statement be used to directly assign the appropriate grade given a percentage mark? If not how could you do this? Write a statement to carry out the assignment of grades.

3. Write a switch statement to assign grades as described in the previous question. Use the fact that mark/10 gives the first digit of the mark.

14.5 Exercises

1. Amend the program you wrote for Exercise 1 in Lesson 13 so that it uses a switch statement to assign the grade. Use the idea suggested in Review question 3 above.

Lesson 15

Further Structured Design

15.1 Repetition Control Structures

In Section 5.2 it was shown that algorithms can be described using a few concepts, namely, sequence, conditional execution (or selection) and repetition. The idea of assignment was also used.

In Section 9.1 algorithm design using top-down structured methods was considered. In that section only conditional structures were covered. This section considers repetition control structures which have already been illustrated in Lesson 5. They are further expanded below.

15.2 Example One: Using a `while` loop

The following algorithm illustrates several techniques. The requirement is for a program which given a set of positive data values will output the minimum value, the maximum value and the average value of the data values.

Before starting on the algorithmic description it must be decided how termination of the data is to be signalled. It would be irritating to the user if after each data entry a yes or no reply had to be given to a question asking whether there was any more data. Equally it might be difficult (and error-prone) for the user to give a count of the number of data elements before processing starts. In this case a better way is to make use of the fact that all the numbers are positive, if the user then enters a negative number when there are no more numbers to process then the program can easily recognise this entry as not being a data entry and hence terminate the entry of data. This technique of placing a **sentinel** to terminate an input list is commonplace.

It must also be decided what should happen when there is no input, that is the user enters a negative number initially. Also it must be known what type of numbers are entered, are they whole numbers or real numbers? Hence the following improved requirement specification:

A user is to enter positive real values from the keyboard when prompted by the program. To signal end of input the user enters a negative number. When data entry has terminated the program should output the minimum positive value entered, the maximum positive value entered and the average of the positive values entered. If there is no data entry (the user enters a negative number initially) then the program should output a message indicating that no data has been entered.

In this program as each number is entered it must be compared with zero to check if it is the sentinel value that terminates input. Each positive number must be added to an accumulated sum and a count of the number of numbers entered must be incremented (so the average can be found). Also each positive number entered must be compared with the minimum/maximum entered so far and if necessary these values are updated. This means that **while** the entered number is positive it must be processed. Thus a first attempt at an algorithm might be:

```
initialise.
enter first value.
while (value is positive)
   {
    process value.
    enter a value.
   }
if no data entry
   then output ‘no data entry’
   otherwise
      {
       evaluate average.
       output results.
      }
```

As usual the first thing done in this algorithm is an `initialise` step. This will be expanded later once the details of the rest of the algorithm have been finalised. The `process value` statement must carry out various tasks. The sum of the input data values must be accumulated and a count of the number of values entered must be incremented. The data value read in must be compared with the minimum/maximum so far input and if necessary these values are updated. Hence the expansion of `process value` is:

```
process value:
  add value to accumulated sum.
  add one to count of number of values.
  if value is bigger than saved maximum then
```

```
         put value in saved maximum.
      if value is less than saved minimum then
         put value in saved minimum.
```

Looking at this expansion it is obvious that prior to the first entry to the loop the following variables require initialisation:

1. a variable for the accumulated sum — this must start at zero.
2. a variable for the number of values — again this starts at zero.
3. variables for the saved maximum and minimum — at the first execution of `process value` the only previous value is the first value, hence the initialisation is to set the saved maximum and the saved minimum to this value.

Hence the beginning of the program can be expanded to:

```
set sum to zero.
set count to zero.
enter first value.
set saved minimum and saved maximum
                      to this value.
```

If no data is entered then this can be recognised by the fact that count will be zero after execution of the `while` statement. Finding the average merely requires dividing the sum by the count of values and output is fairly obvious. Thus the final version is:

```
set sum to zero.
set count to zero.
enter first value.
set minimum and maximum to this value.
while (value is positive)
    {
     add value to sum.
     add one to count.
     if value is bigger than maximum then
                     set maximum to value.
     if value is smaller than minimum then
                      set minimum to value.
     read a value.
    }
if count is zero
    then output ‘no data entry’
    else {
          set average to sum/count.
```

```
             output count, average, maximum and minimum.
            }
```

Note that if no data is entered then the terminating negative value will be assigned to minimum and maximum. This does not matter because in this case no use is made of these variables.

15.3 Example Two: Using a `while` loop

A set of marks are available for a class of students. For each student the following details are available:

```
a candidate number - 4 digits
a percentage examination mark for subject 1
a percentage coursework mark for subject 1
a percentage examination mark for subject 2
a percentage coursework mark for subject 2
```

A program is required which will read in the marks as above for each student and will output for each student, in one line, the above information together with a final mark for each subject. The final mark in each subject is 70% of the examination mark plus 30% of the coursework mark. The average final mark for each subject should also be output. End of input is to be signalled by input of a negative candidate number. A first algorithmic description of the required program is:

```
initialise.
enter candidate number.
while candidate number is positive
   {
    enter marks for candidate.
    process marks.
    output results for candidate.
    enter candidate number.
   }
calculate subject averages.
output subject averages.
```

A little thought shows that two accumulation variables will be required to sum the two final subject marks, and since an average has to be found the number of students must be counted. Initially these sums and the count must be initialised to zero and in `process marks` the marks must be added to the sums and the count incremented. Including these features the algorithmic description is expanded to:

```
set sum1 and sum2 to zero.
set count to zero.
enter candidate number.
while candidate number is positive
  {
    enter s1, cw1, s2, cw2.   // the four candidate marks
    increment count.
    set final1 to 0.7*s1+0.3*cw1.
    set final2 to 0.7*s2+0.3*cw2.
    add final1 to sum1.
    add final2 to sum2.
    output candidate number, s1, cw1, s2, cw2, final1, final2.
    enter candidate number.
  }
set subject1 average to sum1/count.
set subject2 average to sum2/count.
output subject1 average.
output subject2 average.
```

Note that this algorithm would fail if the first candidate number entered was negative. The statement loop in the `while` statement would not be executed and hence `count` would retain its initial value of zero. This would lead to a division by zero when an attempt was made to calculate the averages. It is often difficult to decide whether a program should cope gracefully with non-existent data, after all why would a user use the program if they had no data to enter? However a typing error may lead to the entry of unsuitable data and it is preferable that the program should recover gracefully from this situation rather than fail with an obscure run-time error later. This algorithm could do this by either checking the sign of the first candidate number entered and terminating with a suitable message if it is negative or could recognise the situation by checking that count is non-zero before proceeding to calculate the average.

This algorithm could be extended by adding various facilities to it, for example indicating for each student whether they have passed judged by some criteria. See the exercises for possible extensions for you to try.

15.4 Other forms of Repetition Control Structures

The while statement used above executes a statement while some condition remains true. Sometimes it is more natural to repeat something until some condition becomes true. The repetition portion of the first version of the algorithm in Section 15.2 could be written in the alternative form:

```
initialise.
```

```
Enter first value.
repeat
   {
    process value.
    enter a value.
   }
until value is negative.
```

If this version is executed using some positive values terminated by a negative value then it has exactly the same effect as the version using the while statement. However its behaviour is different if a negative number is entered first. In the repeat-until statement the termination condition is tested after the body of the loop has been executed once. Hence in this example the negative value entered first would be processed, giving a result when none was expected and then another entry value would be expected. Hence the repeat-until statement cannot be easily used in situations where it might not be necessary to execute the body of the loop. In this case it is better to use the while statement which tests a continuation condition before executing the body of the loop.

Another form of repetition control is often required, that is to repeat some statement a known number of times. This can be done with a while statement. For example to execute a statement n times could be implemented as follows:

```
set count to zero.
while count is less than n
  {
   statement.
   increment count.
  }
```

There are several things that the programmer has to get right here, the initial value of count, the condition and remembering to increment count inside the loop body. It would be simpler to use a construct like:

```
repeat n times
  {
   statement.
  }
```

Many loops fall into this simple category. The following problem identifies a related form of repetition control structure.

> Write a program to enter a value for n and output a table of the squares of the numbers from 1 to n.

An algorithm to solve this problem could be written as follows:

```
for i taking values from 1 to n do
  {
   output i and i squared.
   take a new line.
  }
```

It is normal in the forms 'repeat n times' and 'for i taking values from start to finish' to not execute the body of the loop if n is zero or if the start value is greater than the finish value.

The most basic of repetition control structures is the while control structure since all the others can be simulated by it. However the others are useful in some cases, particularly the for control structure. They all have equivalents in C++.

15.5 Summary

- An important facility in designing algorithms is the ability to specify the repetition of a group of operations.
- The repetition of a group of statements is called a **loop** and the statements that are repeated are called the **body** of the loop.
- The most basic repetition control structure is the `while` loop. This loop repeats the body of a loop while some condition remains true. If the condition is initially **false** then the loop body is never executed.
- The **repeat-until** repetition control structure repeats the body of the loop until some condition becomes **true**. Since the body of the loop is executed before the test of the condition this means that the body of the loop is always executed at least once.
- The `for` repetition control structure repeats the body of the loop a fixed number of times, possibly while some control variable sequences through a specified set of values.

15.6 Review questions

1. What does the following algorithm do?

```
set sum to zero.
set i to 1.
input n.
while i is less than or equal to n do
   {
    add i to sum.
```

```
      increment i.
     }
  output sum.
```

What would be output if the value entered for `n` was 0? What would be the effect of reversing the order of the statements in the loop body on the general result produced by the algorithm?

15.7 Exercises

1. In example 2 in this lesson an algorithm was produced to process examination marks. Extend the algorithm so that an indication of pass or fail is printed for each student. A student passes if his/her average mark over the two subjects is greater than 40 and neither mark is below 35. Also extend the algorithm so that it will output a suitable message if no student details are entered.

2. Design an algorithm to solve the following problem:

    > A set of numbers is to be entered to the computer and the number of negative and the number of positive values entered are to be output. All the numbers lie between -100.0 and 100.0.

    Use a sentinel controlled loop and choose a suitable value for the sentinel.

3. Rewrite your solution to Exercise 1 for the case where the number of students in the class is entered initially. Write one version using a **while** loop and another using a **repeat** loop.

4. Write an algorithm which will print out a table of ascending powers of two from two to the power zero (1) and terminate printing at the greatest value less than 10,000. Give two versions of the algorithm, one using a **while** loop and the other using a **repeat-until** loop.

Lesson 16

The `while` statement

The following piece of C++ illustrates a `while` statement. It takes a value entered by the user and as long as the user enters positive values it accumulates their sum. When the user enters a negative value the execution of the `while` statement is terminated.

```
sum = 0.0;
cin >> x;
while (x > 0.0)
   {
    sum += x;
    cin >> x;
   }
```

The variable `sum` which is to hold the accumulated sum is initialised to zero. Then a value for `x` is entered so that `x` has a value before being tested in the condition `x > 0.0`. Note that the value of `x` is updated in the body of the `while` loop before returning to test the condition `x > 0.0` again.

The general form of a `while` statement is:

```
while ( condition )
   statement
```

While the *condition* is true the *statement* is repeatedly executed. The *statement* may be a single statement (terminated by a semi-colon) or a compound statement. Note the following points:

1. It must be possible to evaluate the *condition* on the first entry to the `while` statement. Thus all variables etc. used in the condition must have been given values before the `while` statement is executed. In the above example the variable `x` was given a value by entering a value from the user.

2. At least one of the variables referenced in the condition must be changed in value in the statement that constitutes the body of the

loop. Otherwise there would be no way of changing the truth value of the condition, which would mean that the loop would become an infinite loop once it had been entered. In the above example `x` was given a new value inside the body of the `while` statement by entering the next value from the user.

3. The *condition* is evaluated before the statement is executed. Thus if the condition is initially false then the statement is never executed. In the above example if the user entered a negative number initially then no execution of the body of the `while` loop would take place.

16.1 Example `while` loop: Printing integers

The following `while` statement prints out the numbers 1 to 10, each on a new line.

```
int i;
i = 1;
while (i <= 10)
  {
    cout << i << endl;
    i++;
  }
```

nl_while.cpp

16.2 Example `while` loop: Summing Arithmetic Progression

The following portion of C++ uses a `while` statement to produce the sum `1+2+3+` ...`+n`, where a value for `n` is entered by the user. It assumes that integer variables `i`, `n` and `sum` have been declared:

```
cout << "Enter a value for n: ";
cin >> n;
sum = 0;
i = 1;
while (i <= n)
  {
    sum += i;
    i++;
  }
cout << "The sum of the first " << n
     << " numbers is " << sum << endl;
```

sumnat_1.cpp

There are several important points to note here:

1. The condition `i <= n` requires that `i` and `n` must have values before the `while` loop is executed. Hence the **initialisation** of `i` to 1 and the entry of a value for `n` before the `while` statement.

2. It is possible that a `while` loop may not be executed at all. For example if the user entered a value 0 for `n` then the condition `i <= n` would be false initially and the statement part of the `while` loop would never be entered.

3. When accumulating the sum of a sequence the variable in which we accumulate the sum must be **initialised** to zero before commencing the summation. Note also that if the user entered a value for `n` that was less than 1 then the initialisation of `sum` would mean that the program would return zero as the accumulated total — if `n` is zero this is certainly a sensible value.

4. There is no unique way to write a `while` statement for a particular loop. For example the loop in this example could have been written as follows:

   ```
   i = 0;
   while (i < n)
     {
       i = i + 1;
       sum = sum + i;
     }
   ```

   without changing the result.

16.3 Example `while` loop: Table of sine function

The following program produces a table of `x` and `sin(x)` as `x` takes values 0 to 90 degrees in steps of 5 degrees. Note that the C++ `sin` function takes an argument in radians, not in degrees. In the program below `sin(radian)` returns the sine of an angle `radian` expressed in radians. The mathematical function library will be covered in more detail in section 20.2.

```
// IEA Oct. 95
// Tabulates x and sin(x)

#include <iostream.h>
#include <math.h>      // because we are going to use
```

```
                       // a mathematical function
void main()
{
  const float degtorad = M_PI/180;  // convert degrees
                                    // to radians
  int degree = 0;      // initialise degrees to zero
  float radian;        // radian equivalent of degree
      // Output headings
  cout << endl << "Degrees" << "   sin(degrees)"
       << endl;
      // Now loop
  while (degree <= 90)
    {
      radian = degree * degtorad;  // convert degrees to
                                   // radians
      cout << endl << "   " << degree << "        "
           << sin(radian);
      degree = degree + 5;  // increment degrees
    }
  cout << endl;
}
```

sin_2.cpp

Note that the mathematical constant π is defined in the mathematical library as `M_PI`.

16.4 Example `while` loop: Average, Minimum and Maximum Calculation

In section 15.2 an algorithm was designed given the following requirement specification:

> A user is to enter positive float values from the keyboard when prompted by the program. To signal end of input the user enters a negative integer. When data entry has terminated the program should output the minimum value entered, the maximum value entered and the average of the positive values entered. If there is no data entry (the user enters a negative number initially) then the program should output a message indicating that no data has been entered.

The final version of the algorithm was:

```
set sum to zero.
set count to zero.
enter first value.
set minimum and maximum to this value.
while (value is positive)
    {
     add value to sum.
     add one to count.
     if value is bigger than maximum then
                     set maximum to value.
     if value is smaller than minimum then
                      set minimum to value.
     read a value.
    }
if count is zero
    then output ‘no data entry’
    else {
          set average to sum/count.
          output count, average, maximum and minimum.
         }
```

The above algorithm can be written in C++ as follows:

```
// IEA Aug 96
// Reads in positive data until a negative number
// is entered and calculates the average and the
// maximum and minimum of the positive entries.

#include <iostream.h>

void main()
{
  float value, sum;
  float average, minimum, maximum;
  int count;
     // initialise
  sum = 0.0;
  count = 0;
  cout << "Enter a value: ";
  cin >> value;
  minimum = value;
  maximum = value;
  while (value >= 0.0)
    {
         // process value
```

```
        sum += value;
        count++;
        if (value > maximum)
           maximum = value;
        else if (value < minimum)
          minimum = value;
           // get next value
        cout << "Enter a value: ";
        cin >> value;
      }
    if (count == 0)
      cout << "No data entry" << endl;
    else
      {
        average = sum / count;
        cout << "There were " << count << " numbers" << endl;
        cout << "Average was " << average << endl;
        cout << "Minimum was " << minimum << endl;
        cout << "Maximum was " << maximum << endl;
      }
}
```

maxminav.cpp

16.5 Example Program: Student mark processing

A set of marks are available for a class of students. For each student the following details are available:

```
a candidate number - 4 digits
a percentage examination mark for subject 1
a percentage coursework mark for subject 1
a percentage examination mark for subject 2
a percentage coursework mark for subject 2
```

A program is required which will read in the marks as above for each student and will output for each student, in one line, the above information together with a final mark for each subject. The final mark in each subject is 70% of the examination mark plus 30% of the coursework mark. The average final mark for each subject should also be output. End of input is to be signalled by input of a negative candidate number. A first algorithmic description of the required program is:

```
initialise.
```

```
enter candidate number.
while candidate number is positive
   {
    enter marks for candidate.
    process marks.
    output results for candidate.
    enter candidate number.
   }
calculate subject averages.
output subject averages.
```

A little thought shows that two accumulation variables will be required to sum the two final subject marks, and since an average has to be found the number of students must be counted. Initially these sums and the count must be initialised to zero and in `process marks` the marks must be added to the sums and the count incremented. Including these features the algorithmic description is expanded to:

```
set sum1 and sum2 to zero.
set count to zero.
enter candidate number.
while candidate number is positive
  {
    enter s1, cw1, s2, cw2.   // the four candidate marks
    increment count.
    set final1 to 0.7*s1+0.3*cw1.
    set final2 to 0.7*s2+0.3*cw2.
    add final1 to sum1.
    add final2 to sum2.
    output candidate number, s1, cw1, s2, cw2, final1, final2.
    enter candidate number.
  }
set subject1 average to sum1/count.
set subject2 average to sum2/count.
output subject1 average.
output subject2 average.
```

This is then easily translated into the following program - the main function only is listed.

```
void main()
{
  int candno;           // candidate number
  int s1, cw1, s2, cw2; // candidate marks
  int final1, final2;   // final subject marks
```

```
    int count;            // number of students
    int sum1, sum2;       // sum accumulators
    int subav1, subav2;   // subject averages
    const float EXAMPC = 0.7, // mark weightings
                CWPC = 0.3;
      // initialise
    sum1 = 0;
    sum2 = 0;
    count = 0;
      // enter candidate number
    cout << "Input candidate number: ";
    cin >> candno;
    while (candno >= 0)
      {
          // enter marks
        cout << "Input candidate marks: ";
        cin >> s1 >> cw1 >> s2 >> cw2;
          // process marks
        count++;
        final1 = int(EXAMPC*s1 + CWPC*cw1);
        final2 = int(EXAMPC*s2 + CWPC*cw2);
        sum1 += final1;
        sum2 += final2;
          // output candidate number and marks
        cout << candno << " "
             << s1 << " " << cw1 << " "
             << s2 << " " << cw2 << " "
             << final1 << " " << final2
             << endl;
          // enter next candidate number
        cout << "Input candidate number (negative to finish): ";
        cin >> candno;
      }
      // evaluate averages
    subav1 = sum1/count;
    subav2 = sum2/count;
       // output averages
    cout << endl << "Subject1 average is " << subav1
         << endl << "Subject2 average is " << subav2
         << endl;
  }
```

exam_1.cpp

Note that this program would fail if the first candidate number entered was negative. The statement loop in the `while` statement would not be executed and hence `count` would retain its initial value of zero. This would lead to a division by zero when an attempt was made to calculate the averages. It is often difficult to decide whether a program should cope gracefully with non-existent data, after all why would a user use the program if they had no data to enter? However a typing error may lead to the entry of unsuitable data and it is preferable that the program should recover gracefully from this situation rather than fail with an obscure run-time error later. This program could do this by either checking the sign of the first candidate number entered and terminating with a suitable message if it is negative or could recognise the situation by checking that count is non-zero before proceeding to calculate the average.

This program could be extended by adding various facilities to it, for example indicating for each student whether they have passed judged by some criteria. See the exercises for possible extensions for you to try.

16.6 Example Program: Iterative evaluation of a square root

Frequently in solving scientific problems **iterative** methods are used. In an iterative method a first approximation to a solution of the problem is produced, then some method which improves the accuracy of the solution is used repeatedly until two successive approximations agree to the accuracy required. This process could be described algorithmically as follows:

```
produce first approximation in a variable old_app.
produce a better approximation as a function
                     of old_app in a variable new_app.
while old_app and new_app are not close enough
   {
    set old_app equal to new_app.
    produce a better approximation as a function
                      of old_app in the variable new_app.
   }
```

A simple problem that can be solved in this way is that of finding the square root of a positive number. If *old* is an approximation to the square root of a number `x` then a better approximation, *new*, is given by:

$$new = (old + \frac{x}{old})/2$$

For example taking 3 as an approximation to the square root of 10 gives the following sequence of approximations:

old	*new*
3	$(3 + 10/3)/2 \rightarrow 3.17$
3.17	$(3.17 + 10/3.17)/2 \rightarrow 3.1623$
3.1623	$(3.1623 + 10/3.1623)/2 \rightarrow 3.162278$

It can be seen that the sequence of approximations is rapidly converging to the correct value, which is 3.16227766. This may not always be true in every application of iterative methods, but in the case of square roots it will always happen as long as the first approximation chosen is positive.

In the algorithmic description above the phrase 'while `old_app` and `new_app` are not close enough' was used. How do we test this? In the square root example it might be decided that the new approximation would be accepted as the correct value of the root when it differed by less than 0.0005 from the previous approximation. This would mean that the estimate of the root was correct to three decimal places. It might be tempting to write the test as

```
while ((new_app-old_app) > 0.0005)
```

but in the second iteration this test would give `3.1623-3.17 > 0.0005` which is equivalent to `-0.0077 > 0.0005` which is false, causing the iteration process would stop prematurely. The problem is that if `new_app-old_app` is ever negative then since a negative number can never be greater than a positive number the condition will become false however large the difference between `new_app` and `old_app`! The solution to this problem is to test the absolute magnitude, without regard to sign, of `new_app` and `old_app`. C++ has a function `fabs(x)` which returns the absolute value of `x` i.e. `x` if `x` is positive and `-x` if `x` is negative. Using this function the test could be written:

```
while (fabs(new_app-old_app) > 0.0005)
```

which solves the problem above. In general if trying to find if two quantities are within some tolerance of each other then **test the absolute value of the difference** between them against the tolerance. However there is another difficulty with the above. Consider the following approximations:

exact value	approximation
100	100.1
0.1	0.2

Which of these approximations is the most accurate? They both have an absolute error of 0.1 but in one case the error is 0.1% of the quantity being approximated and in the other it is 100% of the quantity being approximated. Thus if these approximations were used as a multiplying factor in the next stage of a computation then in one case the answer would be a small fraction larger than it should be whereas in the other case the answer would be twice what it should be! Thus the relative size of the error with respect to the quantity being approximated is important. In the square root problem if x was 0.000001, with square root 0.001, then two successive approximations 0.0015 and 0.00108 would have a difference less than 0.0005 and hence 0.00108 would be accepted as a result. However this result would be 8% in error. Hence it is always safer to test the relative error against the tolerance, this ensures that results of different sizes have the same number of significant figures correct. Hence if three significant figures were required in the result the test should be:

```
while (fabs((new_app-old_app)/new_app) > 0.0005)
```

Since the square root of a negative number is not defined (in real arithmetic) no attempt should be made to use this algorithm if the value input is negative. Also if the input is zero, which has square root zero, the use of this algorithm will lead to a division by something approaching zero. This should be avoided by making zero a special case. Hence the program:

```
void main()
{
  const float tol = 0.000005; // relative error tolerance
  float value;
  float old_app, new_app;
  cout << "Square root of a number"
       << endl << endl;
  cout << "Enter a positive number: ";
  cin >> value;
  if (value < 0.0)
    cout << "Cannot find square root of negative number"
         << endl;
  else
     if (value == 0.0)
        cout << "square root of "
             << value
             << " is 0.0"
             << endl;
     else
       {
         old_app = value; // take value as first approximation
         new_app = (old_app + value/old_app)/2;
         while (fabs((new_app-old_app)/new_app) > tol)
           {
             old_app = new_app;
             new_app = (old_app + value/old_app)/2;
```

```
            }
           cout << "square root of "
                << value
                << " is " << new_app
                << endl;
        }
}
```

sqrt.cpp

16.7 Summary

- The `while` statement in C++ allows the body of a loop to be executed repeatedly until a condition is not satisfied. For as long as the condition is **`true`** the loop body is executed.
- The `while` statement is the most fundamental of the iteration statements. Because the condition is tested before executing the loop statement the loop statement may be executed zero or more times.
- A `while` statement requires initialisation of any variables in the condition prior to entering the `while` statement . The loop statement must include statements to update at least one of the variables that occurs in the loop condition.
- In testing for convergence of successive values in an iterative sequence always compare the absolute difference of successive values with the required tolerance in testing for termination. It is almost always better to test the relative error between successive values rather than the absolute error.

16.8 Multiple Choice Questions

1. What is the value of `i` after the `while` statement below?

   ```
   n = 10;
   i = 0;
   while ( i < n ) {
    ...
    i++;
   }
   ```

   (a) `9`
   (b) `10`

(c) `11`

2. What are the first and last values of `i` output by this loop?

```
n = 10;
i = 0;
while ( ++i < n ) {
  cout << i << endl;
}
```

(a) 1 and 9

(b) 0 and 9

(c) 0 and 10

(d) 1 and 10

3. What are the first and last values of `i` output by this loop?

```
n = 10;
i = 0;
while ( i++ < n ) {
  cout << i << endl;
}
```

(a) 1 and 10

(b) 0 and 10

(c) 1 and 9

(d) 0 and 9

16.9 Review Questions

1. What would be output by the following segment of C++?

```
int fact, i;
fact = 1;
i = 2;
while (i <= 6)
   fact *= i;
   i++;
cout << fact;
```

If the purpose of this segment of C++ was to output the value of `1*2*3*4*5*6` how should it be changed?

2. Consider the following piece of C++

```
int i;
while (i < 10)
   {
     cout << i << endl;
     i++;
   }
```

To what should `i` be initialised so that the loop would be traversed 10 times? In this case what would be printed out? How would you change the body of the loop so that with the same initialisation the numbers 1 to 10 would be printed? If the body of the loop was kept as it is above how should the initialisation and the condition be changed so that the numbers 1 to 10 are printed out?

3. Write C++ statements using a `while` statement to print `n` asterisks at the beginning of a new line.

4. Write C++ statements using a `while` statement to evaluate `n!`, i.e. 1*2*3*. . . *`n`, 0! is 1.

16.10 Exercises

1. In exercise 2 of Lesson 15 you should have developed an algorithm for the following problem:

   > A set of numbers is to be entered to the computer and the number of negative and the number of positive values entered are to be output. All the numbers lie between -100.0 and 100.0.

   Now implement your algorithm as a C++ program using a `while` loop and a sentinel value to terminate input. Make up a suitable small set of data and choose a suitable sentinel value to terminate input and test your program.

2. In exercise 3 of Lesson 15 you should have developed an algorithm to generate all powers of two up to the largest power less than 10,000. Now implement this as a C++ program. Once you have this working change the program so that the user can enter a value for the stopping value (i.e. a value to replace the 10,000). If you try to enter a value that would cause a number greater than the largest `int` variable possible to be generated then errors will occur! The form the error will take will depend on the compiler being used, it might generate erroneous results and/or get stuck in an infinite loop. The onset of this problem can be delayed by declaring the integer variables holding the limit and the power of two as being of type `long int` which will allow these

variables to take values up to a 10 digit integer. It is important while programming to remember that there are limitations on the size and precision that numerical values can take, if these limitations are not kept in mind nonsensical results can be produced.

3. The standard C++ library contains functions to generate random numbers. Consider the following C++ program which will print out ten random integers:

```
#include <stdlib.h>
#include <time.h>
#include <iostream.h>

void main()
{
 int r;                  // random integer;
 int i = 0;              // control variable
 srand(time(0));                // initialise random number generator
 while (i < 10)
   {
    r = rand();          // gets random int in 0-RAND_MAX
    cout << "Random integer was " << r;
    i++;
   }
}
```

Note that the function `srand` is used **once** at the beginning of `main`, this ensures that the random number sequence will start at a different value each time that the program is run. If the call of `srand` was left out then the program would deliver exactly the same results each time it was run, not very random! The function `rand()` returns a random integer in the range 0 to `RAND_MAX`. `RAND_MAX` is a `const int` defined in the include file `stdlib.h`. The use of `time(0)` as an argument to `srand` ensures that a new starting value is used each time the program is run. Note that the inclusion of the files `stdlib.h` and `time.h` is required before these functions can be used.

Now extend the above program so that it generates `n` random numbers, where `n` is entered by the user, and counts how many of the random integers generated are less than a half of `RAND_MAX`. If the random number generator is producing a uniform spread of random numbers in the interval then this should converge towards a half of `n`. Execute your program with increasing values of `n` (but not bigger than 32767).

Lesson 17

The `do-while` statement

The following example is a version using a `do-while` statement of the problem considered at the beginning of the Lesson on the `while` statement. The program has to accept positive numbers entered by a user and to accumulate their sum, terminating when a negative value is entered.

```
sum = 0.0;
cin >> x;
do {
    sum += x;
    cin >> x;
  }
while (x > 0.0);
```

Again the accumulator variable `sum` is initialised to zero and the first value is entered from the user before the do-while statement is entered for the first time. The statement between the `do` and the `while` is then executed *before* the condition `x > 0.0` is tested. This of course is different from the while statement in which the `condition` is tested before the *statement* is executed. This means that the compound statement between the `do` and the `while` would be executed at least once, even if the user entered a negative value initially. This value would then be added to `sum` and the computer would await entry of another value from the user! Thus `do-while` statements are not used where there is a possibility that the statement inside the loop should not be executed.

The general form of the `do-while` statement is:

```
do
  statement
while ( condition );  // note the brackets!
```

In the `do-while` statement the body of the loop is executed *before* the first test of the condition. The loop is terminated when the condition becomes **false**. As noted above the *loop statement is always executed at least* *once*, unlike the `while` statement where the body of the loop is not executed at all if the condition is initially **false**. The statement may be a single statement or a compound statement. The effect of a `do-while` statement can always be simulated by a `while` statement so it is not strictly necessary. However in some situations it can be more convenient than a `while` statement.

17.1 Example Program: Sum of Arithmetic Progression

The following loop produces the sum `1+2+3+` ...`+n`, where a value for `n` is entered by the user:

```
cout << "Enter a value for n: ";
cin >> n;
sum = 0;
i = 1;
do
  {
    sum += i;
    i++;
  }
while (i <= n);
```

sumnat_2.cpp

If the user entered a value of 0 for `n` then the value of 1 would be returned as the value of `sum`. This is obviously incorrect and, as noted above, is because the loop statement of a `do-while` loop is always executed at least once. In this case if the entered value of `n` is zero then the loop statement should not be entered at all! Thus if there is any possibility that some valid data may require that a loop be executed zero times then a `while` statement should be used rather than a `do-while` statement.

17.2 Example Program: Valid Input Checking

The `do-while` statement is useful for checking that input from a user lies in a valid range and repeatedly requesting input until it is within range. This is illustrated in the following portion of C++ program:

```
bool accept;      // indicates if value in range
float x;          // value entered
float low, high;  // bounds for x

// assume low and high have suitable values
do {
    cout << "Enter a value (" << low <<" to "
         << high << "):";
    cin >> x;
    if (low <= x && x <= high)
      accept = true;
    else
      accept = false;
  }
while (!accept);
```

valid.cpp

Note the use of the logical operator not (`!`) operating on the boolean value, to invert its truth value.

Another way of controlling the loop is to assign the value of the condition directly to `accept`. At first sight, this may appear strange, but the condition is already being evaluated as either `true` or `false`, so it makes sense to replace the `if-else` statement with

```
accept = low <= x && x <= high;
```

17.3 Example Program: Student Mark Processing (2)

The program in section 16.5 could have equally well have been written using a `do-while` loop. The `do-while` loop would be:

```
cin >> candno;
do {
    // enter marks
  cout << "Input candidate marks: ";
  cin >> s1 >> cw1 >> s2 >> cw2;
    // process marks
  count = count+1;
  final1 = int(EXAMPC*s1+CWPC*cw1);
  final2 = int(EXAMPC*s2+CWPC*cw2);
  sum1 = sum1+final1;
  sum2 = sum2+final2;
    // output marks
  cout << candno << " "
```

```
      << s1 << " " << cw1 << " "
      << s2 << " " << cw2 << " "
      << final1 << " " << final2
      << endl;
    // enter candidate number
  cout << "Input candidate number (negative to finish): ";
  cin >> candno;
  } while (candno >= 0);
```

exam_2.cpp

This is not completely equivalent to the `while` statement version. Consider what happens if the user initially enters a negative candidate number—they would be surprised to then be asked for further data and another candidate number! If there is at least one candidate then the two versions are equivalent.

17.4 Summary

- The `do-while` loop statement will always be executed at least once since the condition is not tested until after the first execution of the loop statement.

17.5 Multiple Choice Questions

1. What is the value of `i` after the `do-while` statement below?

   ```
   n = 10;
   i = 0;
   do {
     ...
     i++;
   } while ( i < n );
   ```

   (a) `9`

   (b) `10`

   (c) `11`

2. What are the first and last values of `i` output by this loop?

   ```
   n = 10;
   i = 0;
   do {
   ```

```
  cout << i << endl;
} while ( ++i < n );
```

(a) 1 and 9
(b) 0 and 9
(c) 0 and 10
(d) 1 and 10

3. What are the first and last values of `i` output by this loop?

```
n = 10;
i = 0;
do {
  cout << i << endl;
} while ( i++ < n );
```

(a) 1 and 10
(b) 0 and 10
(c) 1 and 9
(d) 0 and 9

17.6 Review Questions

1. What is the major difference between a `while` statement and a `do-while` statement?

2. What would be output by the following segment of C++?

```
int i;
i = -12;
do
  {
    cout << i << endl;
    i = i - 1;
  }
while (i > 0)
```

3. Do questions 3 and 4 of Lesson 16 again using a `do-while` statement. What would be the difference from your solution using a `while` statement in the case where `n` is zero?

17.7 Exercises

1. Rewrite the program that you produced for exercise 1 in Lesson 16 using a `do-while` statement rather than a `while` statement.

2. Rewrite the program that you produced for exercise 2 in Lesson 16 using a `do-while` statement rather than a `while` statement.

3. Write a C++ program to implement the following requirement:

   > A set of examination marks expressed as percentages are to be processed by computer. Data entry is terminated by entering a negative percentage value. As each examination mark is entered it should be validated as being either a valid percentage or as being a negative number. The program should ask for the data entry to be repeated until a valid value is entered. The program should then output the number of percentage marks entered, the average mark and how many marks were above 40

   The program should implement the loop by using a `do-while` loop.

Lesson 18

The `for` statement

Frequently in programming it is necessary to execute a statement a fixed number of times or as a control variable takes a sequence of values. For example consider the following use of a `while` statement to output the numbers 1 to 10. In this case the integer variable `i` is used to control the number of times the loop is executed.

```
i = 1;
while (i <= 10)
  {
    cout << i << endl;
    i++;
  }
```

In such a while loop three processes may be distinguished:

1. **Initialisation** - initialise the control variable `i` (`i = 1`).
2. **Test expression** - evaluate the truth value of an expression (`i <= 10`).
3. **Update expression** - update the value of the control variable before executing the loop again (`i++`).

These concepts are used in the **for statement** which is designed for the case where a loop is to be executed starting from an initial value of some control variable and looping until the control variable satisfies some condition, meanwhile updating the value of the control variable each time round the loop.

The general form of the `for` statement is:

`for (` *initialise* `;` *test* `;` *update* `)`
  statement

which executes the *initialise* statement when the `for` statement is first entered, the *test* expression is then evaluated and if **true** the loop *statement* is executed followed by the *update* statement. The cycle of (test;execute-statement;update) is then continued until the test expression evaluates to **false**, control then passes to the next statement in the program.

18.1 Example `for` statement: Print 10 integers

The equivalent `for` statement to the `while` statement in section 16.1 is

```
for (i = 1; i <= 10; i++)
   cout << i << endl;
```

nl_for.cpp

which initially sets `i` to 1, `i` is then compared with 10, if it is less than or equal to 10 then the statement to output `i` is executed, `i` is then incremented by 1 and the condition `i <= 10` is again tested. Eventually `i` reaches the value 10, this value is printed and `i` is incremented to 11. Consequently on the next test of the condition the condition evaluates to **false** and hence exit is made from the loop.

18.2 Example `for` statement: Print table of sine function

The following loop tabulates the sin function from `x = 0.0` to `x = 1.6` in steps of `0.1`.

```
int i;
float x;
for (i = 0; i <= 16; i++)
  {
    x = 0.1 * i;
    cout << x << "  " << sin(x) << endl;
  }
```

sin_3.cpp

Note how an integer variable `i` is used to control the loop while inside the loop the corresponding value of `x` is calculated as a function of `i`. This is preferable to the following:

```
float x;
for (x = 0.0; x <= 1.6; x += 0.1)
   cout << x << "  " << sin(x) << endl;
```

The problem with the above is that floating point variables are not held exactly, thus 0.1 might be represented by something slightly larger than 0.1. Then after continually adding 0.1 to `x` the final value that should be 1.6 may actually be something like 1.60001. Thus the test `x <= 1.6` would fail prematurely and the last line of the table would not be printed. This could be corrected by making the test `x <= 1.605` say.

In general it is probably best to use integer variables as control variables in for loops.

18.3 Example Program: Student Mark Processing (3)

The program from section 16.5 could also be modified to use the `for` statement to control the loop. This cannot be done without changing the specification of the input data. If the specification was changed so that the number of students was entered before the examination marks then a `for` statement could be used as follows:

```
void main()
{
  int candno;           // candidate number
  int s1, cw1, s2, cw2; // candidate marks
  int final1, final2;   // final subject marks
  int count;            // number of students
  int sum1, sum2;       // sum accumulators
  int subav1, subav2;   // subject averages
  int i;                // for loop control variable
  const float EXAMPC = 0.7,
              CWPC = 0.3;
    // initialise
  sum1 = 0;
  sum2 = 0;
    // enter number of students
  cout << "Enter number of students: ";
  cin >> count;
  for (i=0; i<count; i=i+1)
    {
        // enter candidate number and marks
      cout << "Input candidate number and marks: ";
      cin >> candno >> s1 >> cw1 >> s2 >> cw2;
        // process marks
      final1 = int(EXAMPC*s1+CWPC*cw1);
      final2 = int(EXAMPC*s2+CWPC*cw2);
      sum1 = sum1+final1;
```

```
        sum2 = sum2+final2;
          // output marks
        cout << candno << " "
             << s1 << " " << cw1 << " "
             << s2 << " " << cw2 << " "
             << final1 << " " << final2
             << endl;
      }
      // evaluate averages
    subav1 = sum1/count;
    subav2 = sum2/count;
       // output averages
    cout << endl << "Subject1 average is " << subav1
         << endl << "Subject2 average is " << subav2
         << endl;
}
```

exam_3.cpp

Note that if zero or a negative number is entered for the number of students then this program will fail, an attempted division by zero will be encountered (why?). This could be fixed by putting in a validation check on the number of students when this is entered by the user.

18.4 Example Program: Generation of Pythagorean Triples

In exercise 7 at the end of Lesson 7 Pythagorean triples were introduced. A Pythagorean triple is a set of three integers a, b and c such that $a^2+b^2=c^2$. Such a triple can be produced from two integers m and n, $m > n$ as follows:

$$\begin{aligned} a &= m^2 - n^2 \\ b &= 2mn \\ c &= m^2 + n^2 \end{aligned}$$

To generate a list of Pythagorean triples `m` could be allowed to take values from its minimum possible value (2) up to some maximum value entered by the user. For each value of `n` from 1 up to `m-1` the corresponding triple could then be evaluated and output. An algorithmic description for this is:

```
enter and validate a maximum value for m.
for each value of m from 2 to maximum do
```

```
{
 for each value of n from 1 to m-1 do
    {
     evaluate and print a triple.
    }
}
```

This description uses a `for` loop, a looping construct that repeats the loop statement as some control variable takes a sequence of values. Also note that one for loop is **nested** inside another, this is very common. From this description the following C++ program is easily produced:

```
void main()
{
  int max,        // maximum value of m to be used
      m, n,       // control variables for loops
      t1, t2, t3; // pythagorean triple
    // enter and validate max
  do {
      cout << "Enter a maximum value for m ( >1 ): ";
      cin >> max;
     }
  while (max < 2);
    // loop on m from 2 to max
  for (m=2; m <= max; m++)
    {
        // now loop on n from 1 to m-1
      for (n=1; n < m; n++)
        {
            // evaluate and print triple
          t1 = m*m-n*n;
          t2 = 2*m*n;
          t3 = m*m+n*n;
          cout << t1
               << " "  << t2
               << " "  << t3
               << endl;
        }
     }
}
```

pyth_2.cpp

18.5 Summary

- The `for` statement is is used to implement loops which execute a fixed number of times. This number of times must be known before the `for` statement is entered.
- If the test expression in a `for` statement is initially **false** then the loop statement will not be executed. Thus a `for` statement may iterate zero or more times.

18.6 Multiple Choice Questions

1. What is the value of `i` after the `for` statement below?

   ```
   n = 100;
   for ( i = 0; i < n; i++ ) {
     ...
   }
   ```

   (a) `99`
   (b) `100`
   (c) `101`

2. What are the first and last values of `i` output by this loop?

   ```
   n = 20;
   for ( i = 0; i < n; i++ ) {
     cout << i << endl;
   }
   ```

   (a) 0 and 19
   (b) 1 and 19
   (c) 0 and 20
   (d) 1 and 20

3. What are the first and last values of `i` output by this loop?

   ```
   n = 15;
   i = 0;
   for( i = 0; i <= n; i++ ) {
     cout << i << endl;
   }
   ```

   (a) 0 and 15

(b) 1 and 14

(c) 1 and 15

4. What is the last value of `i` output by this loop?

```
n = 27;
i = 0;
for( i = 0; i <= n; i+= 2 ) {
  cout << i << endl;
}
```

(a) 25

(b) 28

(c) 27

(d) 26

18.7 Review Questions

1. What would be output by the following segment of C++?

```
int i;
for ( i = 1; i <= 12; i *= 2 )
  cout << i << endl;
```

2. Do questions 3 and 4 of Lesson 16 again using a `for` statement.

3. What is printed by the following segment of C++?

```
int i;
for (i=1; i<20; i = i+3)
    cout << i << endl;
```

What would happen if the `i+3` in the update expression was changed to `i-3`?

4. Write a `for` statement to output the numbers 1 to 20, each on a new line.

5. Write a segment of C++ using a `for` statement which accepts `n` real values entered by a user and outputs their average value. Assume `n` will be greater than zero.

6. Write a segment of C++ which enters a value for n, $n \geq 0$ and outputs the value of 2^n. Use a `while` statement first, then repeat using a `do-while` statement and finally repeat using a `for` statement.

18.8 Exercises

1. Write an algorithm for a program to produce the sum of the series

   $$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

   where `n` is entered by the user. Then write and test the program.

   Change your algorithm so that the sum of the series

   $$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots \frac{1}{n}$$

   is computed. Then write and test the program. As `n` gets larger the results should tend towards 0.7854 ($\pi/4$).

2. In exercise 1 of Lesson 16 you wrote a program to enter a set of numbers and to output counts of the number of negative and positive numbers. Change that program so that it asks the user how many numbers are to be entered and uses a `for` loop.

3. In exercise 3 of Lesson 16 it was described how random integers could be generated by the `rand` function. This routine can also be used to generate random fractions in the range 0-1. Since `rand()` generates an integer between 0 and `RAND_MAX-1` then `rand()/float(RAND_MAX)` generates a fraction in the range (0,1). If x is a random number in the range (0,1) then $1 - 2x$ is a random fraction in the range (-1,+1). Start by writing a program that produces `n` (entered by user) pairs of random fractions in the range (-1,+1). Use a `for` loop to implement the loop.

   Once this is working extend your program as follows. Treat each pair of random values as the co-ordinates of a point (x, y) and output a count of how many of the generated points lie inside a circle of radius 1 and centre at (0,0) i.e. $x^2 + y^2 < 1$.

   If the random number generator used is reasonably uniform in its distribution of numbers then the probability that a point lands in the circle is the ratio of the area of the circle to the area of the 2×2 square centred on the origin. This ratio is $\pi/4$. Thus an estimate of the value of π can be made from the ratio of the count of points inside the circle to the total number of points. Hence extend your program to output an estimate of π. This is known as a Monte Carlo method. Run your program a few times with increasing values of `n` to see how the estimate of π improves with the number of trials carried out.

4. Write an algorithm to produce an `n` times multiplication table (`n` less than or equal to 10). For example, if `n` is equal to four the table should appear as follows:

```
    1    2    3    4
1   1    2    3    4
2   2    4    6    8
3   3    6    9   12
4   4    8   12   16
```

Convert your algorithm into a program. Use nested `for` loops and pay attention to formatting your output so that it is displayed neatly.

Lesson 19

Streams and External Files

So far all input and output has been done by obtaining data from the input stream `cin` and sending output to the output stream `cout`. These streams have been connected to the keyboard and screen respectively. There are many situations when these facilities are not sufficient. For example:

1. When the amount of data to be entered to a program is very large or when a program is to be executed at a later date with the same data. This may happen while developing and testing a program with a set of test data.

2. When the data for a program has been produced by another computer program.

3. When a permanent record of the output of a program is required.

In all these cases **External Files** are required. Thus if a program requires a significant amount of data to be entered then the data can be entered on a file using a text editor and this file used as the input medium while developing and testing the program. Similarly output can be sent to a file so that a permanent record of results is obtained.

19.1 Streams

In C++ data is written to and from **streams**. A stream is :

- a sequence of characters
- connected to a device or to a (disk) file
- referred to by name

The streams `cin` and `cout` which are connected to the keyboard and the screen respectively have already been considered. When using these streams the file `iostream.h` must be included in the program.

C++ allows other streams to be set up. These streams must be declared just as identifiers are declared. Thus streams that are to be used for input are declared as having the type `ifstream` and those that are used for output are declared as having the type `ofstream`. Once a stream has been declared it must be **connected** to an external file.

If streams are going to be used the file `fstream.h` must be included in the program.

19.2 Connecting Streams to External Files

The stream must first be declared. The names for streams follow the same rules of formation as identifiers. Thus the following declaration declares the stream `ins` for input and the stream `outs` for output:

```
ifstream ins;  // input stream
ofstream outs; // output stream
```

To connect these streams to the appropriate external files the `open` function is used. The `open` function is a member function of the stream class and to connect the stream `ins` to a file `indata.dat` the following statement is used:

```
ins.open("indata.dat");
```

The general form for a stream with identifier *streamname* and a file with name *filename* is:

streamname`.open(`*filename*`);`

If the file `indata.dat` did not exist or could not be found then this `open` function will fail. Failure of the `open` function is tested by using the the function *streamname*`.fail()` which returns `true` if the `open` function failed. When using the `open` function failure should always be tested for as in the following example:

```
ifstream ins;
ins.open("indata.dat");
if (ins.fail())
  {
    cout << "Error opening file indata.dat"
         << endl;
    return 1;
  }
```

In this example it is assumed that the failure occurs within the `main()` program. This means that the `return` statement exits the `main` program and hence the program is terminated. The value 1 that is returned indicates to the operating system that the program has terminated with an error condition. As the `main` program now returns a value it must be declared as having the type `int` rather than `void`.

Having connected the external files to the streams any input/output from/to these streams will then be from/to the connected files.

In the following example two streams are declared, one for input called `ins` and another for output called `outs`. These streams are then connected to files with names `indata.dat` and `results.dat` respectively by:

```
int main()
{
 ifstream ins;
 ofstream outs;
 ins.open("indata.dat");
 if (ins.fail())
   {
     cout << "Error opening indata.dat"
          << endl;
     return 1;
   }
 outs.open("results.dat");
 if (outs.fail())
   {
     cout << "Error opening results.dat"
          << endl;
     return 1;
   }
      .
      .
```

All access to these files now uses the declared stream names `ins` and `outs` and the input/output operators `<<` and `>>`. Input/output manipulators can also be used on the output stream.

Thus a data item is entered from the stream `ins` by:

```
ins >> x;
```

and results are output to the stream `outs` by:

```
outs << "Result is " << setw(6) << count << endl;
```

19.3 Testing for end-of-file

In the above the functions `open` and `fail` have been attached to a stream name. Another function that can be attached to a stream name is the end-of-file condition, `eof`. This condition is set true when an an attempt is made to read beyond the end of a file, otherwise it is set to false. Unlike some other languages the end-of-file character is actually entered and the `eof` function returns true when it is entered. For example to read values from a file and evaluate their average then the following code could be used:

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>

int main()
{
  int n;
  float x, sum, average;
  ifstream ins;          // input stream
  ofstream outs;         // output stream
  ins.open("indata.dat");

     // open files, exit program if fail
  if (ins.fail())
    {
      cout << "Can't open indata.dat" <<endl;
      return 1; //exit with code 1 for failure
    }
  outs.open("results.dat");
  if (outs.fail())
    {
      cout << "Can't open results.dat" << endl;
      return 1; //exit with code 1 for failure
    }

  // Initialise and let user know something is happening
  sum = 0.0;
  n = 0;
  cout << "Reading input file " << endl;
      // read from file, accumulate sum and output average
      // to output file.
  ins >> x;  // if file was empty then eof would now be true
  while (!ins.eof())
    {
      sum += x;
      n++;
      ins >> x;
    }
  average = sum / n;
  cout << "Writing results to file " << endl;
  outs << "The average of " << n << " numbers is "
       << setprecision(2)
       << average << endl;
  ins.close();     // Close all files - GOOD PRACTICE
  outs.close();
  return 0;        // indicate success
}
```

sumfile.cpp

Note that at the end the streams `ins` and `outs` were closed. While some operating systems will do this automatically when your program terminates it is good practice to always close files before exiting a program. If this is not done then sometimes not all of the output is written to the file.

If the program was to be run again with data from a different file then the only way this could be done would be to edit the program above and replace each occurrence of `indata.dat` with the name of the other input file. This is obviously inconvenient and it is much better to allow the user to enter the file name at run-time. However this topic is delayed until Lesson 23 when further consideration has been given to the representation of arbitrary strings within a program.

19.4 Summary

- To send data to/from an external file the file must be connected to a stream.
- A stream is a sequence of characters and is connected to a device or a file.
- If streams other than the standard input streams `cin` and `cout` are used then the file `fstream.h` must be included in the program.
- Streams must be declared. Input streams are declared as having type `ifstream` and output streams as having type `ofstream`.
- A file is connected to a stream using the `open(`*filename*`)` member function of the stream. Failure of the `open(`*filename*`)` function is tested by the `fail()` member function of the stream. End of file condition is tested by the `eof()` member function of the stream. A stream (and the associated file) is closed by using the `close()` member function of the stream.

19.5 Review Questions

1. What is the purpose of the `open` function of a stream?
2. Write a declaration for an input stream and connect it to a file called '`datain.txt`'. How would you read a real value from this stream into a `float` variable?
3. Write a declaration for an output stream and connect it to a file called '`results`'. Assuming that there existed float variables `x` and `y` how would you write the values of `x` and `y` to the file `results` together with a message '`The values of x and y are `'?

19.6 Exercises

1. In question 2 of Lesson 18 you wrote a program which input a series of numbers and output counts of the number of positive and negative numbers. Amend this program so that it takes data from a file (make up a sample file) and terminates on the end of file condition. Write the output from the program to another file.

Lesson 20

Top-down design using Functions

In Lesson 5 a program was produced which entered the hours worked and an hourly rate of pay for an employee and output the employee's total wage. This could be expanded so that a user could enter this data for several employees in turn and get their wages output. A suitable algorithmic description for such a program might be:

```
repeat
  enter hours worked and hourly rate.
  produce wage.
  prompt user ‘any more data?’
  read reply
until reply is no
```

Since an algorithm has already been produced which outputs the wage given the hours worked and the hourly rate the description of this could now be used as the expansion of `produce wage`. This re-use of an algorithm from a previous program saves time in developing an algorithm again and if the algorithm has been previously tested and verified to be correct also reduces the chances of error. The best mechanism for this re-use of an algorithm is to incorporate it into a **function**. The function is given a name and is supplied with the input **parameters** (or **arguments**) of the problem and returns the results as output parameters. Thus the description for the calculation of the wage could be placed in a function called, say, `calcwage`. This function would take the hours worked and the hourly rate as input parameters and would return the wage as output. This function is then **called** to produce the wage when values are available for the hours worked and the hourly rate. The algorithm above could then be written:

```
repeat
  Enter hours worked and hourly rate.
```

```
    Call the function calcwage(hours,rate,wage).
    print out wage.
    prompt user 'any more data?'
    read reply.
  until reply is no.
```

Apart from its use in this program the function `calcwage` might possibly be used in other programs in the future which required the calculation of wages. Another advantage of the above approach is that if the rules for calculating wages are changed then only the function `calcwage` need be changed. Thus if the solution of a problem has been neatly encapsulated into a function which has been comprehensively tested and debugged then it can be incorporated into subsequent programs. This obviously saves much work and removes some sources of error. Ultimately everyone in an organisation could use this function in their programs without even having to understand how to actually solve the problem themselves.

20.1 The need for functions

A rationale for the use of functions has been given above. Basically they save work in that having solved a problem once it need not be solved again if there exists a function to solve the problem given the particular parameters for this instance of the problem. In addition the function can be comprehensively tested and hence a possible source of error is eliminated in future programs. These reasons are now expanded upon:

1. When solving large problems it is usually necessary to split the problem down into a series of sub-problems, which in turn may be split into further sub-problems etc. This is usually called a **top-down** approach. This process continues until problems become of such a size that they can be solved by a single programmer. This top-down approach is essential if the work has to be shared out between a team of programmers, each programmer ending up with a specification for a part of the system which is to be written as a function (or functions). While writing a single function the programmer is able to concentrate on the solution of this one problem only and is thus more likely to be able to solve the problem and make less errors. This function can now be tested on its own for correctness.

2. In a particular organisation or industry it may be found that in carrying out the top-down approach in 1 some tasks occur very frequently. For example the operation of sorting a file of data into some order occurs frequently in data-processing applications. Thus a **library** of such commonly used functions can be built up and re-used in many

different programs. This obviously saves much work and cuts down errors if such functions have already been well tested.

3. There are many very specialised problem areas, not every programmer can know every area. For example many programmers working in scientific applications will frequently use mathematical function routines like sine and cosine, but would have no idea how to write such routines. Similarly a programmer working in commercial applications might know very little about how an efficient sorting routine can be implemented. However a specialist can write such routines, place them in a public library of functions and all programmers can benefit from this expertise by being able to use these efficient and well tested functions.

Before looking at how functions are implemented in C++ the use of the mathematical function routines supplied in C++ is considered.

20.2 The mathematical function library in C++

The functions `sin` and `fabs` have already been used in programs in previous Lessons. For example in section 16.3 the statement

```
cout << endl << "  " << degree << "        "
     << sin(radian);
```

occurred. This statement used `sin(radian)` as a call of the C++ function with the name `sin` which returns as its value the sine of the angle (in radians) which is given as its input parameter (in this case the variable `radian`). In this use of a function there are no output parameters, the single result that the function produces is returned to the calling program via the name of the function.

Some of the mathematical functions available in the C++ mathematics library are listed below.

`acos(x)`	inverse cosine, -1 $\leq$ `x` $\leq$ +1, returns value in radians in range 0 to π
`asin(x)`	inverse sine, -1 $\leq$ `x` $\leq$ +1, returns value in radians in range 0 to π
`atan(x)`	inverse tangent, returns value in radians in range -$\pi/2$ to $\pi/2$
`cos(x)`	returns cosine of `x`, `x` in radians
`sin(x)`	returns sine of `x`, `x` in radians
`tan(x)`	returns tangent of `x`, `x` in radians
`exp(x)`	exponential function, e to power `x`
`log(x)`	natural log of `x` (base e), `x` > 0
`sqrt(x)`	square root of `x`, `x` ≥ 0
`fabs(x)`	absolute value of `x`
`floor(x)`	largest integer not greater than `x`
`ceil(x)`	smallest integer not less than `x`

In all these functions the parameter `x` is a floating point value. The `x` is used as a **formal parameter**, that is it is used to denote that a parameter is required and to allow the effect of the function to be described. When the function is called then this formal parameter is replaced by an **actual parameter**. The actual parameter can be a constant, a variable or an expression. An expression may include a call of another function.

These functions are called by quoting their name followed by the actual parameter enclosed in rounded brackets, for example, `exp(x+1)`. The function call can then be used anywhere in an expression that an ordinary variable may be used. Hence the following examples:

```
y = sin(3.14159);
z = cos(a) + sin(a);
factor = sin(theta)/(sin(delta) - sin(delta-theta));
theta = acos(1.0/sqrt(1 - x*x));
if (sin(x) > 0.7071)
  cout << "Angle is greater than 45 degrees";
cout << "The value is " << exp(-a*t)*sin(a*t);
```

The file `math.h` must be included in any program that is going to use any functions from this library. `math.h` also defines some constants which may be used. For example `M_PI` can be used for π and `M_E` can be used for e.

20.3 Summary

- The **top-down** approach to program design splits the initial problem into several sub-problems which in turn can be further sub-divided. Once a sub-problem is simple enough to solve then it can be implemented as a function.

- A function should be completely self-contained. That is it should communicate with the calling program only via supplied input parameters and output parameters. The user of a function should not have to know any details of how the function is implemented.

- Functions encourage the re-use of code and can encapsulate knowledge and techniques of which the user has no knowledge.

20.4 Review Questions

1. Why are functions used?

2. What role do the parameters of a function play?

20.5 Exercises

1. Each competitor in a race is started separately. The start and finish time for each competitor is noted in hours, minutes and seconds. The number of competitors is available as is the distance of the race in miles. Design an algorithm which will initially enter the number of competitors and the distance of the race and will then enter for each competitor a competitor number together with the competitor's start and finish times. The competitor's elapsed time should then be output in minutes and seconds together with the average speed in miles per hour. It is desirable that the start and finish times should be validated as being legal times in the twenty four hour clock. In designing your algorithm identify at the initial stage which tasks could be carried out by using functions. Do not expand the functions any further but specify what they should do and also specify their input parameters and output parameters.

Lesson 21

Introduction to User-defined functions in C++

C++ allows programmers to define their own functions. For example the following is a definition of a function which given the co-ordinates of a point `(x,y)` will return its distance from the origin.

```
float distance(float x, float y)
      // Returns the distance of (x, y) from origin
   {
    float dist;  //local variable
    dist = sqrt(x * x + y * y);
    return dist;
   }
```

dist.cpp

This function has two input parameters, real values `x` and `y`, and returns the distance of the point `(x,y)` from the origin. In the function a **local variable** `dist` is used to temporarily hold the calculated value inside the function.

The general form of a function definition in C++ is as follows:

```
function-type function-name( parameter-list )
   {
     local-definitions;
     function-implementation;
   }
```

- If the function returns a value then the type of that value must be specified in *function-type*. For the moment this could be `int`, `float` or `char`. If the function does not return a value then the *function-type* must be `void`.

- The *function-name* follows the same rules of composition as identifiers.
- The *parameter-list* lists the formal parameters of the function together with their types.
- The *local-definitions* are definitions of variables that are used in the *function-implementation*. These variables have no meaning outside the function.
- The *function-implementation* consists of C++ executable statements that implement the effect of the function.

21.1 Functions with no parameters

Functions with no parameters are of limited use. Usually they will not return a value but carry out some operation. For example consider the following function which skips three lines on output.

```
void skipthree(void)
   // skips three lines on output
   {
    cout << endl << endl << endl;
   }
```

skip3.cpp

Note that the *function-type* has been given as `void`, this tells the compiler that this function does not return any value. Because the function does not take any parameters the *parameter-list* is empty, this is indicated by the `void` *parameter-list*. No local variables are required by this function and the *function implementation* only requires the sending of three successive end of line characters to the output stream `cout`. Note the introductory comment that describes what the function does. All functions should include this information as minimal comment.

Since this function does not return a value it cannot be used in an expression and is **called** by treating it as a statement as follows:

```
skipthree();
```

Even though there are no parameters the empty parameter list `()` *must* be inserted.

When a function is called the C++ compiler must insert appropriate instructions into the object code to arrange to pass the actual parameter values to the function code and to obtain any values returned by the function. To do this correctly the compiler must know the types of all parameters and the type of any return value. Thus before processing the call of a function it must already know how the function is defined. This can be done by defining any functions that are used in the main program before the main program, for example the function `skipthree` could be incorporated in a program as follows:

```
#include <iostream.h>

void skipthree(void)
   // Function to skip three lines
  {
    cout << endl << endl << endl;
  }

void main()
{
  int ....;
  float ....;
  cout << "Title Line 1";
  skipthree();
  cout << "Title Line 2";
      .
      .
}
```

However this has disadvantages, namely:

- The main program tends to convey much more information of use in understanding the program than do individual functions. So it is better if the main program comes first. However this means that the compiler meets the call of a function before it meets the definition of the function.
- If using functions from a library of functions then the main program is linked with the pre-compiled object code of the functions. Thus while compiling the main program on its own the compiler has no knowledge of the function definitions.

The way round both the problems above is to use **Function prototypes**. A function prototype supplies information about the return type of a function and the types of its parameters. This function prototype is then placed before the main program that uses the function. The full function definition is then placed after the main program or may be contained in a separate file that is compiled separately and linked to the main program later. The function prototype is merely a copy of the function heading. Thus the function prototype for the function `skipthree` is:

```
void skipthree(void);
```

which would be included in the program file as follows:

```
#include <iostream.h>

void skipthree(void);  // function prototype

void main()
{
  int ....;
  float ....;
  cout << "Title Line 1";
  skipthree();
  cout << "Title Line 2";
      .
      .
}

// Now the function definition
void skipthree(void)
   // Function to skip three lines
  {
    cout << endl << endl << endl;
  }
```

skip3.cpp

In fact when using functions from the stream libraries and the mathematical libraries prototypes are required for these functions. This is handled by including the files `iostream.h` and `math.h` which, among other things, contain the function prototypes.

21.2 Functions with parameters and no return value

The function of the previous section is not very useful, what if four lines were to be skipped, or two lines? It would be much more useful if it was possible to tell the function how many lines to skip. That is the function should have an input parameter which indicates how many lines should be skipped.

The function `skipthree()` is now changed to the function `skip` which has a parameter `n` indicating how many lines have to be skipped as follows:

```
void skip(int n)
```

```
 // Function skips n lines on output
{
  int i;   // a local variable to this function
    // now loop n times
  for (i = 0; i < n; i++)
    cout << endl;
}
```

skipn.cpp

As before this function does not return a value hence it is declared as having type `void`. It now takes an integer parameter `n` which indicates the number of lines to be skipped. The parameter list then consists of a type and a name for this formal parameter. Inside the body of the function (enclosed in `{}`) a loop control variable `i` is declared. This variable is a **local variable** to the function. A local variable defined within the body of the function has no meaning, or value, except within the body of the function. It can use an identifier name that is used elsewhere in the program without there being any confusion with that variable. Thus changing the value of the local variable `i` in the function skip will not affect the value of any other variable `i` used elsewhere in the program. Similarly changing the value of a variable `i` used elsewhere in the program will not affect the value of the local variable `i` in `skip`.

The function is called in the same manner as `skipthree()` above, but a value must be given for the parameter `n`. Thus all the following calls are acceptable:

```
void main()
{
  int m = 6, n = 3;
  ...............;
  skip(m);
  .......;
  skip(m + n);
  ............;
  skip(4);
  .......;
}
```

however the call:

```
skip (4.0);
```

would not be acceptable because the **actual parameter type** must match the **formal parameter type** given in the definition of the function.

In writing the function prototype for a function with parameters it is not necessary to detail the formal names given to the parameters of the function, only their types. Thus a suitable function prototype for the parameterised version of `skip` would be:

```
void skip(int); // function prototype
```

21.3 Functions that return values

One of the most useful forms of function is one that returns a value that is a function of its parameters. In this case the type given to the function is that of the value to be returned. Thus consider the function, previously considered, which given the co-ordinates of a point `(x,y)` will return its distance from the origin:

```
float distance(float x, float y)
      // Returns the distance of (x, y) from origin
  {
    float dist;  //local variable
    dist = sqrt(x * x + y * y);
    return dist;
  }
```

dist.cpp

The function prototype for this function is:

```
float distance(float, float); // function prototype
```

This function introduces several new features. Note the following:

- The function has been given the type `float` because it is going to return a `float` value.
- The *parameter-list* now has two parameters, namely, `x` and `y`. Each parameter is declared by giving its type and name and successive parameter declarations are separated by a comma.
- A local variable `dist` has been declared to temporarily hold the calculated distance.
- Because this function returns a value it includes a `return` statement which returns the value. In a statement `return` *value* the *value* may be a constant, a variable or an expression. Hence the use of the local variable `dist` was not essential since the `return` statement could have been written:

```
    return sqrt(x*x + y*y);
```

When the function is called the formal parameters x and y are replaced by actual parameters of type float and in the same order, i.e. the x co-ordinate first. Since the function returns a value it can only be used in an expression.

Hence the following examples of the use of the above function in a program in which it is declared:

```
float a, b, c, d, x, y;
a = 3.0;
b = 4.4;
c = 5.1;
d = 2.6;

x = distance(a, b);
y = distance(c, d);

if (distance(4.1, 6.7) > distance(x, y))
   cout << "Message 1" << endl;
```

A function may have several return statements. This is illustrated in the following function which implements the algorithm for evaluating the square root previously considered.

```
float mysqrt(float x)
      // Function returns square root of x.
      // If x is negative it returns zero.
  {
    const float tol = 1.0e-7;  // 7 significant figures
    float xold, xnew;          // local variables
    if (x <= 0.0)
      return 0.0;             // covers -ve and zero case
    else
      {
        xold = x;               // x as first approx
        xnew = 0.5 * (xold + x / xold); // better approx
        while (fabs((xold-xnew)/xnew) > tol)
          {
            xold = xnew;
            xnew = 0.5 * (xold + x / xold);
          }
        return xnew;   // must return float value
       }
  } // end mysqrt
```

mysqrt.cpp

If the function has type `void` then it must *not* return a value. If a `void` function does return a value then most compilers will issue some form of warning message that a return value is not expected.

21.4 Example function: sum of squares of integers

The following function returns the sum of the squares of the first `n` integers when it is called with parameter `n`.

```
// This function returns the sum of squares of the
// first n integers
int sumsq(int n)
  {
    int sum = 0;
    int i;
    for (i = 1; i <= n; i++)
        sum += i * i;

    return sum;
  } // End of sumsq
```

sumsq.cpp

A typical use of `sumsq` is:

```
float sumsquare;
int number;
cout << "Enter number (>= 0): ";
cin >> number;
sumsquare = sumsq(number);
```

21.5 Example Function: Raising to the power

This function returns the value of its first parameter raised to the power of its second parameter. The second parameter is an integer, but may be 0 or negative.

```
float power(float x, int n)
  {
    float product = 1.0;
    int absn;
    int i;
    if ( n == 0)
      return 1.0;
    else
      {
       absn = int(fabs(n));
       for (i = 1; i <= absn; i++)
         product *= x;
       if (n < 0)
         return 1.0 / product;
       else
         return product;
      }
  } // end of power
```

power.cpp

A typical use of the power function is shown below

```
float x, y;
int p;
cout << "Enter a float and an integer: ";
cin >> x >> p;
y = power(x, p);
y = power(x + y, 3);
```

functest.cpp

21.6 Call-by-value parameters

Suppose the function power above is now amended to include the statement

```
n++;
```

just before the final closing } and the following statements are executed:

```
p = 4;
y = power(x, p);
cout << p;
```

What would be printed out for the value of p? In fact instead of the value 5 that you might expect p would still have the value 4. This is because the parameter has been **passed by value**. This means that when the function is called a **copy of the value** of the actual parameter used in the call is passed across to the memory space of the function. Anything that happens inside the function to this copy of the value of the parameter cannot affect the original actual parameter. All the examples that have been considered have used call-by-value parameters. This is because all the parameters used have been input parameters. To make a parameter call-by-value it is specified in the parameter list by giving its type followed by its name.

Thus if a parameter is only to be used for passing information into a function and does not have to be returned or passed back from the function then the formal parameter representing that parameter should be call-by-value. Note also that since the function cannot change the value of a call-by-value parameter in the calling program strange side effects of calling a function are avoided.

21.7 Summary

- A C++ function can return a value. The function must be declared to have the same type as this value. If the function does not return a value then it must be given the type `void`.
- Information is passed into a function via the *parameter-list*. Each parameter must be given a type. If not declared to be otherwise then parameters are treated as *value parameters*
- If a parameter is a value parameter then the function operates on a copy of the value of the actual parameter hence the value of the actual parameter cannot be changed by the function.
- A function prototype provides information to the compiler about the return type of a function and the types of its parameters. The function prototype must appear in the program before the function is used.
- Any variable declared inside a function is *local* to that function and has existence and meaning only inside the function. Hence it can use an identifier already used in the main program or in any other function without any confusion with that identifier.

21.8 Review Questions

1. How is information supplied as input to a function? How can information be conveyed back to the calling program?
2. What would the following function do?

```
void example(int n)
  {
    int i;
    for (i=0; i<n; i++)
```

```
        cout << '*';
      cout << endl;
    }
```

How would you call this function in a program? How would you use this function in producing the following output on the screen?

```
*
**
***
****
```

3. What would be the output from the following programs?

   a)

```
void change(void)
 {
   int x;
   x = 1;
 }
void main()
 {
   int x;
   x = 0;
   change();
   cout << x << endl;
 }
```

   b)

```
void change(int x)
  {
    x = 1;
  }
void main()
  {
    int x;
    x = 0;
    change(x);
    cout << x << endl;
  }
```

4. Write a function prototype for a function that takes two parameters of type `float` and returns **true** (1) if the first parameter is greater than the second and otherwise returns **false** (0).

5. Write a function prototype for a function that takes two parameters of type `int` and returns **true** if these two integers are a valid value for a sum of money in pounds and pence. If not valid then **false** should be returned.

6. A function named `ex1` has a local variable named `i` and another function `ex2` has a local variable named `i`. These two functions are used together with a main program which has a variable named `i`. Assuming that there are no other errors in the program will this program compile correctly? Will it execute correctly without any run-time errors?

21.9 Exercises

1. Write a function which draws a line of `n` asterisks, `n` being passed as a parameter to the function. Write a **driver** program (a program that calls and tests the function) which uses the function to output an `m`$\times$ `n` block of asterisks, `m` and `n` entered by the user.

2. Extend the function of the previous exercise so that it prints a line of `n` asterisks starting in column `m`. It should take two parameters `m` and `n`. If the values of `m` and `n` are such that the line of asterisks would extend beyond column 80 then the function should return **false** and print nothing, otherwise it should output **true** and print the line of asterisks. Amend your driver program so that it uses the function return value to terminate execution with an error message if `m` and `n` are such that there would be line overflow.

   Think carefully about the test data you would use to test the function.

3. Write a function which converts a sum of money given as an integer number of pence into a floating point value representing the equivalent number of pounds. For example 365 pence would be 3.65 pounds.

Lesson 22

Further User-defined functions in C++

In Lesson 21 all function parameters were input-only parameters and thus were implemented as **call-by value** parameters. The only method used to return information to the calling program was by the function returning a single value. Frequently it is necessary to write functions that return more than one value. For example a function that took a sum of money in pence might have to return the equivalent sum in pounds and pence.

To allow information to be returned to the calling program C++ allows information to be returned by parameters. As explained in Lesson 21 this cannot be done by the use of **call-by-value** parameters. To allow a parameter to return a value it must be declared to be a **call-by-reference** parameter.

22.1 Call-by-reference parameters

Values cannot be returned to the calling program via call-by-value parameters because the function only operates on a copy of the value of the parameters, not on the actual parameter itself. If it is required to return a value by a parameter then the address of the actual parameter used in the function call must be passed to the function. The function can then use this address to access the actual parameter in its own space in the calling program and change it if required. Thus what we are passing is a **reference** to the parameter. Hence **call-by-reference** parameters.

To indicate that a parameter is **called by reference** an `&` is placed after the type in the parameter list. Any change that is made to that parameter in the function body will then be reflected in its value in the calling program.

For example consider the following function to evaluate the solution of a quadratic equation:

```
    // solves the quadratic equation a*x*x+b*x+c = 0.
```

```
// If the roots are real then the roots are
// returned in two parameters root1 and root2 and
// the function returns true, if they are complex
// then the function returns false.

bool quadsolve(float a,        // IN coefficient
               float b,        // IN coefficient
               float c,        // IN coefficient
               float& root1,   // OUT root
               float& root2)   // OUT root

  {
    float disc;      // local variable
    disc = b * b - 4 * a * c;
    if (disc < 0.0)
      return false;
    else
      {
        root1 = (-b + sqrt(disc))/(2 * a);
        root2 = (-b - sqrt(disc))/(2 * a);
        return true;
      }
  }
```

quadrat.cpp

Note that the roots, which are output parameters, have been declared to be reference parameters, while the coefficients are input parameters and hence are declared to be value parameters. The function prototype would have the following form:

```
int quadsolve(float, float, float, float&, float&);
```

This might be called in a program as follows:

```
float c1, c2, c3;
float r1, r2;
      .
      .
if (quadsolve(c1, c2, c3, r1, r2))
    cout << "Roots are " << r1 << " and " << r2 << endl;
else
    cout << "Complex Roots" << endl;
```

quadrat.cpp

Note how the return value has been used to discriminate between the situation where the roots are real and are found in the two output parameters and the case where the roots are complex and no values for the roots are returned.

22.2 Example Program: Invoice Processing

Details of an invoice are available as follows:

```
The number of items on the invoice - n
For each item
   an item code (6 digits), a quantity and
                a unit cost (pounds,pence)
```

Thus a typical set of input values for an invoice might be:

```
3
161432 5 6 50
543289 10 2 25
876234 2  10 75
```

indicating that there are three items on the invoice, the first item having an item code of 161432, an order quantity of 5 and a unit price of six pounds fifty pence. Assume that the days date will also be entered.

Write a C++ program to enter details of such an invoice and to output an invoice which indicates the total cost of each item and the total cost of the invoice together with full details. The above details might produce an invoice as follows:

```
Invoice date: 10/6/96

 Item    quantity  unit price  total price

161432       5         6.50        32.50
543289      10         2.25        22.50
876234       2        10.75        21.50

                          Total    76.50
```

A first attempt at an algorithm might be:

```
initialise.
enter date.
enter number of items into n.
output headings.
for n items do
```

```
    {
      enter a record.
      calculate total cost of item.
      accumulate total invoice cost.
      write line of invoice.
    }
  output total cost.
```

Assume that there are four programmers available to implement this program. There are four major operations inside the `for` loop hence each programmer could be given one operation to implement. The best way to do this is to use a function for each operation. Each programmer can then be given a precise definition of a function.

For example consider the operation `enter a record`. This function must read in the integer quantities item number, quantity and unit price in pounds and pence. Hence it has no input parameters and four output parameters. A definition of the function could then be written as follows:

```
Function name: dataentry
Operation:     Enter a record
Description:   Enters four integers from the current
               input line and returns their values.
Parameters:    Output parameter  int  itemno
               Output parameter  int  quantity
               Output parameter  int  unitpounds
               Output parameter  int  unitpence
```

Similarly the other functions could be specified as follows:

```
Function name : calccost
Operation     : Calculates the cost for a single item.
Description   : Given the unit price of an item in
                pounds and pence and the quantity of
                the item calculates the total cost in
                pounds and pence.
Parameters    : Input parameter int quantity
                input parameter int unitpounds
                input parameter int unitpence
                output parameter int totalpound
                output parameter int totalpence

Function name : acctotal
Operation     : Accumulates the total cost of invoice
Description   : Given current total invoice cost and
                the total cost of an invoice item
                calculates the new total invoice cost.
```

```
Parameters    : input parameter int totalpound
                input parameter int totalpence
                input & output parameter int invpound
                input & output parameter int invpence

Function name : writeline
Operation     : Writes a line of the invoice.
Description   : Given the item reference number, the
                quantity, the unit price and total
                price of an item outputs a line of
                the invoice.
Parameters    : input parameter int itemno
                input parameter int quantity
                input parameter int unitpounds
                input parameter int unitpence
                input parameter int totalpound
                input parameter int totalpence
```

In terms of these functions the main program could be written as follows:

```
void main()
{
  int i,          // control variable
      n,          // number of items
      itemno,     // item reference number
      quantity,   // quantity of item
      unitpounds,
      unitpence,  // unit item price
      totalpound,
      totalpence, // total item price
      invpound,
      invpence;   // total invoice price
      // initialise
  invpound = 0; // total value of invoice has to be
  invpence = 0; // set to zero initially
      // Enter number of items
  cout << "Enter number of items on invoice: ";
  cin  >> n;
      // Headings
  cout << " Item   quantity  unit price  total price"
       << endl << endl;
      // For n items
  for (i=1; i<=n; i++)
    {
      dataentry(itemno, quantity, unitpounds, unitpence);
```

```
        calccost(quantity, unitpounds, unitpence, totalpound,
                 totalpence);
        acctotal(totalpound, totalpence, invpound, invpence);
        writeline(itemno, quantity, unitpounds, unitpence,
                  totalpound, totalpence);
      }
        // write total line
    cout << "                         Total    "
          << invpound
          << "."
          << invpence << endl;
  }
```

invoice.cpp

Using the function specifications above the functions can now be written and tested separately. For example `calccost` could be written as follows:

```
void calccost(int q, int ul, int up,
              int& totl, int& totp)
     // Calculates the quantity q times the unit cost in
     // pounds and pence in ul and up and places the
     // result in pounds and pence in totl and totp

  {
    int p;
    p = q * up;
    totp = p % 100;
    totl = q * ul + p/100;
  }
```

calccost.cpp

To test this function on its own a **driver program** would have to be written. A driver program is a simple program that allows the programmer to enter values for the parameters of the function to be tested and outputs the results. A suitable driver program to test the above function could be:

```
// IEA 1996
// Driver program to test calccost

#include <iostream.h>

// function prototype
```

```
void calccost(int, int, int, int&, int&);

void main()
{
  int quant, unitl, unitp, totall, totalp;
    // stop on negative quantity
  cout << "Enter quantity: ";
  cin >> quant;
  while (quant >= 0)
    {
      cout << "Enter unit cost (pounds pence): ";
      cin >> unitl >> unitp;
      calccost(quant, unitl, unitp, totall, totalp);
      cout << endl
           << quant << " times "
           << unitl << " pounds "
           << unitp << " pence "
           << " is "
           << totall << " pounds "
           << totalp << " pence ";
      cout << endl << "Enter quantity: ";
      cin >> quant;
    }
}

// function definition here
```

calccost.cpp

When testing functions try to use one example of each of the cases that can occur. For example using the above driver program to show that `calccost` works for 7 times 1.10 is not a complete test since it does not generate any ‘carry’ from the pence to the pounds. An additional test on say 6 times 1.73 checks that the carry works. Choosing a set of test data to adequately validate a function requires much thought.

22.3 Summary

- Information is passed back from the function via *reference* parameters in the *parameter-list*. A parameter is declared to be a reference parameter by appending & to its type. In this case the address of the actual parameter is passed to the function and the function uses this address to access the actual parameter value and can change the value. Hence information can be passed back to the calling program.

22.4 Review Questions

1. How is information supplied as input to a function? How can information be conveyed back to the calling program?

2. What would be the output from the following program?

   ```
   void change(int& y)
     {
       y = 1;
     }
   void main()
     {
       int x;
       x = 0;
       change(x);
       cout << x << endl;
     }
   ```

3. Write a function prototype for a function that takes two parameters of type `int` and returns **true** if these two integers are a valid value for a sum of money in pounds and pence. If they are valid then the value of the sum of money should also be returned in pence (without affecting the input value of pence). If not valid then **false** should be returned.

22.5 Exercises

1. Write a function

   ```
   void floattopp(float q, int& L, int& P)
   ```

   which converts the sum of money `q` in pounds into `L` pounds and `P` pence where the pence are correctly rounded. Thus if `q` was 24.5678 then `L` should be set to 24 and `P` should be set to 57. Remember that when assigning a real to an integer the real is truncated. Thus to round a real to the nearest integer add 0.5 before assigning to the integer.

   Write a simple driver program to test the function. Think carefully about the boundary conditions.

2. It is required to print out a table of mortgage repayments for a range of years of repayment and a range of rates of interest. Write an algorithm to produce a table of the monthly repayments on a mortgage of 40,000 pounds repayable over 15, 16, 17,..., 30 years with interest

rates of 3, 4, 5,. . . , 10 per cent. Assume that the actual repayment will be produced by a function which will take as input parameters the principle `P`, the repayment time `n` in years and the rate of interest `r` as a percentage rate and will return the monthly repayment in pounds and pence.

Transform your algorithm into a C++ program. Write a function prototype for the repayment calculation function and write the function itself as a **function stub**. That is a function that does not perform the correct calculation but delivers values that are sufficient to test the rest of the program. In this case it should return values of zero for the pounds and pence of the monthly repayment. This will allow you to test that your program lays the table out properly.

Once you have the table layout working you can now write the function itself. The repayment each month on a mortgage of `P` pounds, taken out over `n` years at an interest rate of `r`% is given by:

$$repayment = \frac{rPk^{12n}}{1200(k^{12n}-1)}$$

where

$$k = 1 + \frac{r}{1200}$$

In writing this function use the function `power` from Section 21.5. This means you must add this function, and a suitable prototype, to your program. Also use the function of exercise 1 above.

Lesson 23

Arrays

Variables in a program have values associated with them. During program execution these values are accessed by using the identifier associated with the variable in expressions etc. In none of the programs written so far have very many variables been used to represent the values that were required. Thus even though programs have been written that could handle large lists of numbers it has not been necessary to use a separate identifier for each number in the list. This is because in all these programs it has never been necessary to keep a note of each number individually for later processing. For example in summing the numbers in a list only one variable was used to hold the current entered number which was added to the accumulated sum and was then overwritten by the next number entered. If that value was required again later in the program there would be no way of accessing it because the value has now been overwritten by the later input.

If only a few values were involved a different identifier could be declared for each variable, but now a loop could not be used to enter the values. Using a loop and assuming that after a value has been entered and used no further use will be made of it allows the following code to be written. This code enters six numbers and outputs their sum:

```
sum = 0.0;
for (i = 0; i < 6; i++)
  {
    cin >> x;
    sum += x;
  }
```

This of course is easily extended to `n` values where `n` can be as large as required. However if it was required to access the values later the above would not be suitable. It would be possible to do it as follows by setting up six individual variables:

```
float a, b, c, d, e, f;
```

and then handling each value individually as follows:

```
sum = 0;
cin >> a;  sum += a;
cin >> b;  sum += b;
cin >> c;  sum += c;
cin >> d;  sum += d;
cin >> e;  sum += e;
cin >> f;  sum += f;
```

which is obviously a very tedious way to program. To extend this solution so that it would work with more than six values then more declarations would have to be added, extra assignment statements added and the program re-compiled. If there were 10000 values imagine the tedium of typing the program (and making up variable names and remembering which is which)!

To get round this difficulty all high-level programming languages use the concept of a data structure called an **Array**

23.1 Arrays in C++

An **array** is a data structure which allows a collective name to be given to a group of elements which **all have the same type**. An individual element of an array is identified by its own unique **index** (or **subscript**).

An array can be thought of as a collection of numbered boxes each containing one data item. The number associated with the box is the index of the item. To access a particular item the index of the box associated with the item is used to access the appropriate box. The index **must** be an integer and indicates the position of the element in the array. Thus the elements of an array are **ordered** by the index.

23.1.1 Declaration of Arrays

An array declaration is very similar to a variable declaration. First a type is given for the elements of the array, then an identifier for the array and, within square brackets, the number of elements in the array. The number of elements **must be an integer**.

For example data on the average temperature over the year in Britain for each of the last 100 years could be stored in an array declared as follows:

```
float annual_temp[100];
```

This declaration will cause the compiler to allocate space for 100 consecutive `float` variables in memory. The number of elements in an array must be fixed at compile time. It is best to make the array size a constant and then, if required, the program can be changed to handle a different size of array by changing the value of the constant,

```
const int NE = 100;
float annual_temp[NE];
```

then if more records come to light it is easy to amend the program to cope with more values by changing the value of NE. This works because the compiler knows the value of the constant NE at compile time and can allocate an appropriate amount of space for the array. It would not work if an ordinary variable was used for the size in the array declaration since at compile time the compiler would not know a value for it.

23.1.2 Accessing Array Elements

Given the declaration above of a 100 element array the compiler reserves space for 100 consecutive floating point values and accesses these values using an index/subscript that takes values from 0 to 99. The **first element** in an array in C++ always has the **index 0**, and if the array has `n` elements the last element will have the index `n-1`.

An **array element** is accessed by writing the identifier of the array followed by the subscript in square brackets. Thus to set the 15th element of the array above to 1.5 the following assignment is used:

```
annual_temp[14] = 1.5;
```

Note that since the first element is at index 0, then the `i`th element is at index `i-1`. Hence in the above the 15th element has index 14.

An array element can be used anywhere an identifier may be used. Here are some examples assuming the following declarations:

```
const int NE = 100,
          N = 50;
int i, j, count[N];
float annual_temp[NE];
float sum, av1, av2;
```

A value can be read into an array element directly, using `cin`

```
cin >> count[i];
```

The element can be increased by 5,

```
count[i] = count[i] + 5;
```

or, using the shorthand form of the assignment

```
count[i] += 5;
```

Array elements can form part of the condition for an `if` statement, or indeed, for any other logical expression:

```
if (annual_temp[j] < 10.0)
   cout << "It was cold this year "
        << endl;
```

`for` statements are the usual means of accessing every element in an array. Here, the first `NE` elements of the array `annual_temp` are given values from the input stream `cin`.

```
for (i = 0; i < NE; i++)
  cin >> annual_temp[i];
```

The following code finds the average temperature recorded in the first ten elements of the array.

```
sum = 0.0;
for (i = 0; i <10; i++)
    sum += annual_temp[i];
av1 = sum / 10;
```

Notice that it is good practice to use named constants, rather than literal numbers such as 10. If the program is changed to take the average of the first 20 entries, then it all too easy to forget to change a 10 to 20. If a `const` is used consistently, then changing its value will be all that is necessary.

For example, the following example finds the average of the *last* `k` entries in the array. `k` could either be a variable, or a declared constant. Observe that a change in the value of `k` will still calculate the correct average (provided `k<=NE`).

```
sum = 0.0;
for (i = NE - k; i < NE; i++)
  sum += annual_temp[i];
av2 = sum / k;
```

Important - C++ does not check that the subscript that is used to reference an array element actually lies in the subscript range of the array. Thus C++ will allow the assignment of a value to `annual_temp[200]`, however the effect of this assignment is unpredictable. For example it could lead to the program attempting to assign a value to a memory element that is outside the program's allocated memory space. This would lead to the program being terminated by the operating system. Alternatively it might actually access a memory location that is within the allocated memory space of the program and assign a value to that location, changing the value of the variable in your program which is actually associated with that memory location, or overwriting the machine code of your program. Similarly reading a value from `annual_temp[200]` might access a value that has not been set by the program or might be the value of another variable. It is the programmer's responsibility to ensure that if an array is declared with `n` elements then no attempt is made to reference any element with a subscript outside the range 0 to `n-1`. Using an index, or subscript, that is out of range is called **Subscript Overflow**. Subscript overflow is one of the commonest causes of erroneous results and can frequently cause very strange and hard to spot errors in programs.

23.1.3 Initialisation of arrays

The initialisation of simple variables in their declaration has already been covered. An array can be initialised in a similar manner. In this case the initial values are given as a list enclosed in curly brackets. For example initialising an array to hold the first few prime numbers could be written as follows:

```
int primes[] = {1, 2, 3, 5, 7, 11, 13};
```

Note that the array has not been given a size, the compiler will make it large enough to hold the number of elements in the list. In this case `primes` would be allocated space for seven elements. If the array is given a size then this size must be greater than or equal to the number of elements in the initialisation list. For example:

```
int primes[10] = {1, 2, 3, 5, 7};
```

would reserve space for a ten element array but would only initialise the first five elements.

23.2 Example Program: Printing Outliers in Data

The requirement specification for a program is:

> A set of positive data values (200) are available. It is required to find the average value of these values and to count the number of values that are more than 10% above the average value.

Since the data values are all positive a negative value can be used as a sentinel to signal the end of data entry. Obviously this is a problem in which an array must be used since the values must first be entered to find the average and then each value must be compared with this average. Hence the use of an array to store the entered values for later re-use.

An initial algorithmic description is:

```
initialise.
enter elements into array and sum elements.
evaluate average.
scan array and count number greater than
                       10% above average.
output results.
```

This can be expanded to the complete algorithmic description:

```
set sum to zero.
set count to zero.
set nogt10 to zero.
enter first value.
while value is positive
   {
    put value in array element with index count.
    add value to sum.
    increment count.
    enter a value.
   }
average = sum/count.
for index taking values 0 to count-1
    if array[index] greater than 1.1*average
       then increment nogt10.
output average, count and nogt10.
```

In the above the variable `nogt10` is the number greater than 10% above the average value. It is easy to argue that after exiting the `while` loop, `count` is set to the number of positive numbers entered. Before entering the loop `count` is set to zero and the first number is entered, that is `count` is one less than the number of numbers entered. Each time round the loop another number is entered and `count` is incremented hence `count` remains one less than the number of numbers entered. But the number of numbers entered is one greater than the number of positive numbers so `count` is therefore equal to the number of positive numbers.

A `main()` program written from the above algorithmic description is given below:

```
void main()
{
  const int NE = 200;  // maximum no of elements in array
  float sum = 0.0;     // accumulates sum
  int count = 0;       // number of elements entered
  int nogt10 = 0;      // counts no greater than 10%
                       // above average
  float x;             // holds each no as input
  float indata[NE];    // array to hold input
  float average;       // average value of input values
  int i;               // control variable

      // Data entry, accumulate sum and count
      // number of +ve numbers entered
```

```
    cout << "Enter numbers, -ve no to terminate: " << endl;
    cin >> x;
    while (x >= 0.0)
      {
        sum = sum + x;
        indata[count] = x;
        count = count + 1;
        cin >> x;
      }

        // calculate average
    average = sum/count;

        // Now compare input elements with average
    for (i = 0; i < count; i++)
      {
        if (indata[i] > 1.1 * average)
          nogt10++;
      }

        // Output results
    cout << "Number of values input is " << n;
    cout << endl
         << "Number more than 10% above average is "
         << nogt10 << endl;
  }
```

outliers.cpp

Since it was assumed in the specification that there would be less than 200 values the array size is set at 200. In running the program less than 200 elements may be entered, if `n` elements where `n < 200` elements are entered then they will occupy the first `n` places in the array `indata`. It is common to set an array size to a value that is the maximum we think will occur in practice, though often not all this space will be used.

23.3 Example Program: Test of Random Numbers

The following program simulates the throwing of a dice by using a random number generator to generate integers in the range 0 to 5. The user is asked to enter the number of trials and the program outputs how many times each possible number occurred.

An array has been used to hold the six counts. This allows the program to increment the correct count using one statement inside the loop rather than using a switch statement with six cases to choose between variables if separate variables had been used for each count. Also it is easy to change the number of sides on the dice by changing a constant. Because C++ arrays start at subscript 0 the count for an `i` occurring on a throw is held in the `i-1`th element of this `count` array. By changing the value of the constant `die_sides` the program could be used to simulate a `die_sides`-sided die without any further change.

```
#include <iostream.h>
#include <stdlib.h>    // time.h and stdlib.h required for
#include <time.h>      // random number generation

void main()
{
  const int die_sides = 6;    // maxr-sided die
  int count[die_sides];       // holds count of each
                         // possible value
  int no_trials,         // number of trials
      roll,              // random integer
      i;                 // control variable
  float sample;          // random fraction 0 .. 1

      // initialise random number generation and count
      // array and input no of trials
  srand(time(0));
  for (i=0; i < die_sides; i++)
    count[i] = 0;
  cout << "How many trials? ";
  cin >> no_trials;

      // carry out trials
  for (i = 0; i < no_trials; i++)
    {
      sample = rand()/float(RAND_MAX);
      roll = int ( die_sides * sample);
           // returns a random integer in 0 to die_sides-1
      count[roll]++;        // increment count
    }

      // Now output results
  for (i = 0; i < die_sides; i++)
    {
      cout << endl << "Number of occurrences of "
           << (i+1) << " was " << count[i];
    }
  cout << endl;
}
```

throwdie.cpp

23.4 Arrays as parameters of functions

In passing an array as a parameter to a function it is passed as a reference parameter. What is actually passed is the address of its first element. Since arrays are passed by reference this means that if the function changes the value of an element in an array that is a parameter of the function then the corresponding actual array of the call will have that element changed.

Though an array is passed as a reference parameter an `&` is not used to denote a reference parameter. However it must be indicated to the compiler that this parameter is an array by appending `[]` to the formal parameter name. Thus to declare an array of real values as a parameter requires the parameter to be specified as follows:

```
..., float A[],...
```

This is the same as a normal array declaration but the size of the array is not specified. This is illustrated in the following example which returns the average value of the first `n` elements in a real array.

```
float meanarray(int n,        // IN no of elements
                float A[])    // IN array parameter
     // This function returns the average value of
     // the first n elements in the array A which
     // is assumed to have >= n elements.
  {
    float sum = 0.0;     // local variable to
                         // accumulate sum
    int i;               // local loop control
    for (i = 0; i < n; i++)
        sum += A[i];

    return sum/n;
  }  // end meanarray
```

meanarry.cpp

If when this function was called the value given for the parameter `n` was greater than the number of elements in the actual array replacing the parameter `A` then an incorrect result would be returned.
The function `meanarray` could be used as follows:

```
const int NE = 100;
float average, data[NE];
int i, m;

cout << "Enter no of data items (no more than "
     << NE << "): ";
cin >> m;

for (i = 0; i < m; i++)
   cin >> data[i];

average = meanarray(m, data);
```

An array can also be an output parameter, consider the following example in which the function `addarray` adds two arrays together to produce a third array whose elements are the sum of the corresponding elements in the original two arrays.

```
void addarray(int size,          // IN size of arrays
              const float A[],   // IN input array
              const float B[],   // IN input array
              float C[])         // OUT result array

     // Takes two arrays of the same size as input
     // parameters and outputs an array whose elements
     // are the sum of the corresponding elements in
     // the two input arrays.

   {
     int i;     // local control variable
     for (i = 0; i < size; i++)
          C[i] = A[i] + B[i];

   } // End of addarray
```

The function `addarray` could be used as follows:

```
float one[50], two[50], three[50];
           .
           .
addarray(20, one, two, three);
```

Note that the parameter size could have been replaced with any value up to the size that was declared for the arrays that were used as actual parameters. In the example above the value of 20 was used which means that only the first 20 elements of the array `three` are set.

Also note that the input parameters `A` and `B` have been declared in the function head as being of type `const float`. Since they are input parameters they should not be changed by the function and declaring them as constant arrays prevents the function from changing them.

23.5 Strings in C++

So far the only form of character information used has been single characters which are defined as being of type `char`. Character strings have also been used in output.

A new data type is now considered, namely, the **character string**, which is used to represent a sequence of characters regarded as a single data item. In C++ strings of characters are held as an **array of characters**, one character held in each array element. In addition a special **null character**, represented by '`\0`', is appended to the end of the string to indicate the end of the string. Hence if a string has `n` characters then it requires an `n+1` element array (at least) to store it. Thus the character '`a`' is stored in a single byte, whereas the single-character string `"a"` is stored in two consecutive bytes holding the character 'a' and the null character.

A string variable `s1` could be declared as follows:

```
char s1[10];
```

The string variable `s1` could hold strings of length up to nine characters since space is needed for the final null character. Strings can be initialised at the time of declaration just as other variables are initialised. For example:

```
char s1[] = "example";
char s2[20] = "another example"
```

would store the two strings as follows:

```
s1  |e|x|a|m|p|l|e|\0|

s2  |a|n|o|t|h|e|r| |e|x|a|m|p|l|e|\0|?|?|?|?|
```

In the first case the array would be allocated space for eight characters, that is space for the seven characters of the string and the null character. In the second case the string is set by the declaration to be twenty characters long but only sixteen of these characters are set, i.e. the fifteen characters of the string and the null character. Note that the length of a string does not include the terminating null character.

23.5.1 String Output

A string is output by sending it to an output stream, for example:

```
cout << "The string s1 is " << s1 << endl;
```

would print

```
The string s1 is example
```

The `setw(`*width*`)` I/O manipulator can be used before outputting a string, the string will then be output right-justified in the field width. If the field width is less than the length of the string then the field width will be expanded to fit the string exactly. If the string is to be left-justified in the field then the `setiosflags` manipulator with the argument `ios::left` can be used.

23.5.2 String Input

When the input stream `cin` is used space characters, newline etc. are used as separators and terminators. Thus when inputting numeric data `cin` skips over any leading spaces and terminates reading a value when it finds a white-space character (space, tab, newline etc.). This same system is used for the input of strings, hence a string to be input cannot start with leading spaces, also if it has a space character in the middle then input will be terminated on that space character. The null character will be appended to the end of the string in the character array by the stream functions. If the string `s1` was initialised as in the previous section, then the statement

```
cin << s1;
```

would set the string `s1` as follows when the string `"first"` is entered (without the double quotes)

```
|f|i|r|s|t|\0|e|\0|
```

Note that the last two elements are a relic of the initialisation at declaration time. If the string that is entered is longer than the space available for it in the character array then C++ will just write over whatever space comes next in memory. This can cause some very strange errors when some of your other variables reside in that space!

To read a string with several words in it using `cin` we have to call `cin` once for each word. For example to read in a name in the form of a Christian name followed by a surname we might use code as follows:

```
char christian[12], surname[12];
cout << "Enter name ";
cin >> christian;
cin >> surname;
cout << "The name entered was "
     << christian << " "
     << surname;
```

The name would just be typed by the user as, for example,

```
Ian Aitchison
```

and the output would then be

```
The name entered was Ian Aitchison
```

In Lesson 19 it was noted that it would be useful if the user of a program could enter the name of the data file that was to be used for input during that run of the program. The following example illustrates how this may be done. It assumes that a file name for an input file must be entered and also a file name for an output file.

```
// IEA 1996
// Example program which copies a specified
// input file to a specified output file.
// It is assumed that the input file holds a
// sequence of integer values.

#include <iostream.h>
#include <fstream.h>

int main()
{
  ifstream ins;                  // declare input and output
  ofstream outs;                 // file streams
  char infile[20], outfile[20];  // strings for file names
  int i;

     // ask user for file names
  cout << "Enter input file name: ";
  cin >> infile;
  cout << "Enter output file name: ";
  cin >> outfile;

     // Associate file names with streams
  ins.open(infile);
  if (ins.fail())
    {
      cout << "Could not open file " << infile
```

```
            << " for input" << endl;
        return 1; // exit with code 1 for failure
      }
    outs.open(outfile);
    if (outs.fail())
      {
        cout << "Could not open file " << outfile
             << " for output" << endl;
        return 1; // exit with code 1 for failure
     }

       // input from input file and copy to output file
    ins >> i;
    while (!ins.eof())
      {
        outs << i << " ";
        ins >> i;
      }
    outs << endl;

        // close files
    ins.close();
    outs.close();
    return 0; //return success indication.
  }
```

This program assumes that the file names entered by the user do not contain more than 19 characters. Note how a space character was output after each integer to separate the individual values in the output file.

23.6 Summary

- An array is used to store a collection of data items which are all of the same type.
- An individual element of an array is accessed by its index, which must be an integer. The first element in the array has index 0.
- When arrays are declared they must be given an integer number of elements. This number of elements must be known to the compiler at the time the array is declared.
- If an attempt is made to access an element with an out of range index then an unpredictable error can occur. Always ensure that array in-

dices for elements of arrays are in range when verifying the correctness of your programs.

- Arrays can be initialised when they are declared.
- When an array is passed as a parameter to a function then it is passed as a reference parameter. Hence any changes made to the array inside the function will change the actual array that is passed as a parameter.
- Character strings are represented by arrays of characters. The string is terminated by the null character. In declaring a string you must set the size of the array to at least one longer than required to hold the characters of the string to allow for this null character.
- Strings can be input and output via the input and output streams, `cin` and `cout`.

23.7 Review Questions

1. If an array has a 100 elements what is the allowable range of subscripts?
2. What is the difference between the expressions `a4` and `a[4]`?
3. Write a declaration for a 100 element array of floats. Include an initialisation of the first four elements to 1.0, 2.0, 3.0 and 4.0.
4. An array `day` is declared as follows:

   ```
   int day[] = {mon, tue, wed, thu, fri};
   ```

   How many elements has the array `day`? If the declaration is changed to

   ```
   int day[7] = {mon, tue, wed, thu, fri};
   ```

   how many elements does `day` have?
5. What would be output by the following section of C++?

   ```
   int A[5] = {1 , 2, 3, 4};
   int i;
   for (i=0; i<5; i++)
    {
     A[i] = 2*A[i];
     cout << A[i] << "  ";
    }
   ```
6. What is wrong with the following section of program?

```
int A[10], i;
for (i=1; i<=10; i++)
   cin >> A[i];
```

7. Write a function heading for a function which will double the first `n` elements of an array. If the function was amended so that it would return **false** if `n` was larger than the size of the array how should the function heading be written? If the function was to be changed so that a new array was produced each of whose elements were double those of the input array how would the heading be written?

23.8 Exercises

1. To familiarise yourself with using arrays write a program that declares two `float` arrays, say with 5 elements each, and carries out the following:

   (a) Input some data from the user into the two arrays.

   (b) Output the sum of the elements in each of the two arrays.

   (c) Output the inner product of the two arrays - that is the sum of the products of corresponding elements `A[0]*B[0] + A[1]*B[1]+` ....etc.

   (d) Produce an estimate of how different the values in the two arrays are by evaluating the sum of squares of the differences between corresponding elements of the two arrays divided by the number of elements.

   Start by only entering and printing the values in the arrays to ensure you are capturing the data correctly. Then add each of the facilities above in turn.

2. A popular method of displaying data is in a Histogram. A histogram counts how many items of data fall in each of `n` equally sized intervals and displays the results as a bar chart in which each bar is proportional in length to the number of data items falling in that interval.

   Write a program that generates `n` random integers in the range 0-99 (see exercise 3 of Lesson 16) and produces a Histogram from the data. Assume that we wish to count the number of numbers that lie in each of the intervals 0-9, 10-19, 20-29, ..., 90-99. This requires that we hold 10 counts, use an array to hold the 10 counts. While it would be possible to check which range a value `x` lies in by using `if-else` statements this would be pretty tedious. A much better way is to note that the value of `x/10` returns the index of the count array element to increment. Having calculated the interval counts draw the Histogram by printing each bar of the Histogram as an appropriately sized line of X's across the screen as below

```
 0 -  9  16  XXXXXXXXXXXXXXXX
10 - 19  13  XXXXXXXXXXXXX
20 - 29  17  XXXXXXXXXXXXXXXXX
    etc.
```

3. In question 1 above you should have written C++ statements to enter numbers into an array. Convert these statements into a general function for array input. Your function should indicate the number of elements to be entered and should signal an error situation if this is greater than the size of the array—think about the required parameters. Also write a function to output `n` elements of a given array five to a line.

   Write a driver program to test these functions and once you are satisfied they are working correctly write functions:

   (a) To return the minimum element in the first `n` elements of an array.

   (b) To return a count of the number of corresponding elements which differ in the first `n` elements of two arrays of the same size.

   (c) Which searches the first `n` elements of an array for an element with a given value. If the value is found then the function should return **true** and also return the index of the element in the array. If not found then the function should return **false**.

   In these functions incorporate error testing for a number of elements greater than the array size.