PART A:
FUNDAMENTAL CONCEPTS OF THE PROGRAMMING PROCESS

Chapter 1: Concept of an algorithm

Chapter 2: Development and semi-formal specification of algorithms, based on a simplified computer model.

Chapter 3: The Structure Theorem
Chapter 1a: Concept of an algorithm

A program must be systematically and properly designed before coding begins. This design process results in the construction of an algorithm. An algorithm is a formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. The concept of an algorithm is often illustrated by the example of a recipe, although many algorithms are much more complex. Basically, an algorithm is a method, like a recipe, in that by following the steps of an algorithm you are guaranteed to find the solution or the answer, if there is one. Algorithms often have steps that repeat (iterate) or require decisions (such as logic or comparison). Algorithms can be composed to create more complex algorithms. The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as finding the common divisor of two numbers or multiplying two numbers. The concept was formalized in 1936 through Alan Turing's Turing machines and Alonzo Church's lambda calculus, which in turn formed the foundation of computer science. Most algorithms can be directly implemented by computer programs; any other algorithms can at least in theory be simulated by computer programs. In many programming languages, algorithms are implemented as functions or procedures.

1. Preparing a Computer Program

The steps in preparing a computer program involve:

1- Study the requirement specification for the application.

2- Analyse the problem and decide how to solve it: what algorithm to use?

3- Translate the algorithm produced at the previous step into a suitable language which is the source program or source code.

4- Compile the program into machine-language or object code. The compiler may find Syntax errors in the program. A syntax error is a mistake in the grammar of a language. Before proceeding, syntax errors must be corrected and compilation is repeated until the compiler produces an executable program free from syntax errors. A run-time error will cause the program to halt during execution because it cannot carry out an instruction.

5- Run the program with test data to detect Logical errors. Logical errors are caused by errors in the method of solution, thus while the incorrect statement is syntactically correct but it is asking the computer to do something which is incorrect in the context of the application.

6- The program can now be put into general use.
2. Definition of some fundamental concept of software development

Before we go further in the design and analysis of algorithms, there are a number of related concepts that need to be defined.

2.1. Program:

A program is an organized list of instructions that, when executed, causes the computer to behave in a predetermined manner. Without programs, computers are useless.

A program is like a recipe. It contains a list of ingredients (called variables) and a list of directions (called statements) that tell the computer what to do with the variables. The variables can represent numeric data, text, or graphical images.

There are many programming languages -- C, C++, Pascal, BASIC, FORTRAN, COBOL, and LISP are just a few. These are all high-level languages. One can also write programs in low-level languages called assembly languages, although this is more difficult. Low-level languages are closer to the language used by a computer, while high-level languages are closer to human languages.

Eventually, every program must be translated into a machine language that the computer can understand. This translation is performed by compilers, interpreters, and assemblers.

When you buy software, you normally buy an executable version of a program. This means that the program is already in machine language -- it has already been compiled and assembled and is ready to execute.

2.2 Statement:

A statement is an instruction written in a high-level language. A statement directs the computer to perform a specified action. A single statement in a high-level language can represent several machine-language instructions. Programs consist of statements and expressions. An expression is a group of symbols that represent a value.

2.3. Expression:

In programming, an expression is any legal combination of symbols that represents a value. Each programming language and application has its own rules for what is legal and illegal. For example, in the C++ language $x + 5$ is an expression, as is the character string "MONKEYS."

Every expression consists of at least one operand and can have one or more operators. Operands are values, whereas operators are symbols that represent particular actions. In the expression $x + 5$, $x$ and 5 are operands, and + is an operator.
Expressions are used in programming languages, database systems, and spreadsheet applications. For example, in database systems, you use expressions to specify which information you want to see. These types of expressions are called queries.

Expressions are often classified by the type of value that they represent. For example:

- **Boolean expressions**: Evaluate to either TRUE or FALSE
- **Integer expressions**: Evaluate to whole numbers, like 3 or 100
- **Floating-point expressions**: Evaluate to real numbers, like 3.141 or -0.005
- **String expressions**: Evaluate to character strings

2.4. Operands and operators:

In all computer languages, expressions consist of two types of components: *operands* and *operators*. Operands are the objects that are manipulated and operators are the symbols that represent specific actions. For example, in the expression $5 + x$, $x$ and $5$ are operands and $+$ is an operator. All expressions have at least one operand.

2.5. Data and Data Structure:

*Data* represent distinct pieces of information, usually formatted in a special way. Data can exist in a variety of forms -- as numbers or text on pieces of paper, as bits and bytes stored in electronic memory.

In programming, the term *data structure* refers to a scheme for organizing related pieces of information. The basic types of data structures include:

- **files**: A collection of data or information that has a name, called the *filename*. Almost all information stored in a computer must be in a file. There are many different types of files: *data files*, *text files*, *program files*, *directory files*, and so on. Different types of files store different types of information. For example, program files store programs, whereas text files store text.
- **lists**: Any ordered set of data.
- **arrays**: A series of objects all of which are the same size and type. Each object in an array is called an *array element*. For example, you could have an array of integers or an array of characters or an array of anything that has a defined data type. Each element has the same data type (although they may have different values). The entire array is stored contiguously in memory (that is, there are no gaps between elements). Arrays can have more than one dimension. A one-dimensional array is called a *vector*; a two-dimensional array is called a *matrix*.
- **records**: A combination of other data objects. For example, a record might contain three integers, a floating-point number, and a character string.
- **trees**: A type of data structure in which each element is attached to one or more elements directly beneath it. The connections between elements are called branches. Trees are often called *inverted trees* because they are normally drawn with the root at the top. The elements at the very bottom of an inverted tree (that is, those that have no elements below them) are called *leaves*. Inverted trees are
the data structures used to represent hierarchical file structures. In this case, the leaves are files and the other elements above the leaves are directories. A binary tree is a special type of inverted tree in which each element has only two branches below it.

- **tables**: Refers to data arranged in rows and columns. A spreadsheet, for example, is a table. In relational database management systems, all information is stored in the form of tables.
Chapter 2a: Semi-formal Specification of Algorithms, Based on a Simplified Computer Model

Semi-formal specifications are used to simplify the formalization of a problem. A popular and intuitive way of representing algorithms is the use of pseudocode. Pseudocode is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of programming languages, but omits detailed subroutines, variable declarations or language-specific syntax.

When designing a solution algorithm, you need to keep in mind that a computer will eventually perform the set of instructions written. That is, if you use words and phrases in the pseudocode which are in line with basic computer operations, the translation from the pseudocode algorithm to a specific programming language becomes quite simple.

This chapter establishes six basic computer operations and introduces common words and keywords used to represent these operations in pseudocode. Each operation can be represented as a straightforward English instruction, with keywords and indentation to signify a particular control structure.

1. A computer can receive information

When a computer is required to receive information or input from a particular source, whether it be a terminal, a disk or any other device, the verbs Read and Get are used in pseudocode. Read is usually used when the algorithm is to receive input from a record on a file, while Get is used when the algorithm is to receive input from the keyboard. For example, typical pseudocode instructions to receive information are:

Read student name
Get system date
Read number-1, number 2
Get tax-code

Each example uses a single verb, Read or Get, followed by one or more nouns to indicate what data is to be obtained.

2. A computer can put out information

When a computer is required to supply information or output to a device, the verbs Print, Write, Put, Output or Display are used in pseudocode. Print is usually used when the output is to be sent to the printer, while Write is used when the output is to be written to a file. If the output is to be written to the screen, the words Put, Output or Display are used in pseudocode. Typical pseudocode examples are:

Print 'Program Completed'
Write customer record to master file
Put out name, address and postcode
Output total-tax
Display 'End of data'

Usually an output Prompt instruction is required before an input Get instruction. The Prompt verb causes a message to be sent to the screen, which requires the user to respond, usually by providing input - for example:

Prompt for student-mark
Get student-mark

3. A computer can perform arithmetic

Most programs require the computer to perform some sort of mathematical calculation, or formula, and for these, a programmer may use either actual mathematical symbols or the words for those symbols. For instance, the same pseudocode instruction can be expressed as either of the following:

Add number to total
total = total + number

Both expressions clearly instruct the computer to add one value to another, so either is acceptable in pseudocode. The equal symbol ‘=’ has been used to indicate assignment of a value as a result of some processing.

The verbs Compute and Calculate are also available. Some pseudocode examples to perform a calculation are:

Divide total_marks by student_count
Sales_tax = cost_price*0.10
Compute C = (F - 32)*5/9

When writing mathematical calculations for the computer standard mathematical ‘order of operations’ applies to pseudocode and most computer languages. The first operation carried out will be any calculation contained within parentheses. Next, any multiplication or division, as it occurs from left to right, will be performed. Then, any addition or subtraction, as it occurs from left to right, will be performed.

4. A computer can assign a value to a variable or memory location

There are three cases where you may write pseudocode to assign a value to a variable or memory location:

- To give data an initial value in pseudocode, the verbs Initialise or Set are used.
- To assign a value as a result of some processing, the symbols '=' or'←' are written.

- To keep a variable for later use, the verbs *Save* or *Store* are used.

Some typical pseudocode examples are:

Initialise total_price to zero
Set student_count to 0
Total_price = cost-price + sales-tax
total_price ← cost_price + sales_tax
Store customer_num in last_customer_num

Note that the '=' symbol is used to assign a value to a variable as a result of some processing and is not equivalent to the mathematical '=' symbol. For this reason, some programmers prefer to use the '←' symbol to represent the assign operation.

5. A computer can compare two variables and select one of two alternate actions

An important computer operation available to the programmer is the ability to compare two variables and then, as a result of the comparison, select one of two alternate actions. To represent this operation in pseudocode, special keywords are used: *IF*, *THEN* and *ELSE*. The comparison of data is established in the *IF* clause, and the choice of alternatives is determined by the *THEN* or *ELSE* options. Only one of these alternatives will be performed. A typical pseudocode example to illustrate this operation is:

IF student_attendance_status is part-time THEN
  add 1 to part_time_count
ELSE
  add 1 to full_time_count
ENDIF

In this example the attendance status of the student is investigated, with the result that either the part-time-count or the full_time_count accumulator is incremented. Note the use of indentation to emphasise the THEN and ELSE options, and the use of the delimiter ENDIF to close the operation.

6. A computer can repeat a group of actions:

When there is a sequence of processing steps that need to be repeated, two special keywords, DOWHILE and ENDIDO, are used in pseudocode. The condition for the repetition of a group of actions is established in the DOWHILE clause, and the actions to be repeated are listed beneath it. For example:

DOWHILE student_total < 50
  Read student record
  Print student name, address to report
\textit{Add 1 to student-total} \\
\textit{ENDDO}

In this example it is easy to see the statements that are to be repeated, as they immediately follow the \textit{DOWHILE} statement and are indented for added emphasis. The condition that controls and eventually terminates the repetition is established in the \textit{DOWHILE} clause, and the keyword \textit{ENDDO} acts as a delimiter. As soon as the condition for repetition is found to be false, control passes to the next statement after the \textit{ENDDO}.

\textbf{Conclusion:}

There are six basic computer operations which are: to receive information, put out information, perform arithmetic, assign a value to a variable, decide between two alternative actions, and repeat a group of actions. Typical pseudocode keywords have been presented to illustrate some semi-formal specifications of algorithms based on the simple computer model presented earlier.
Chapter 3a: The Structure Theorem

The very first step in the development of a computer program is defining the problem. This involves carefully reading and rereading the problem until you understand completely what is required. To help with this initial analysis, the problem should be divided into three separate components:

- **Input:** a list of the source data provided to the problem.
- **Output:** a list of the outputs required.
- **Processing:** a list of actions needed to produce the required outputs.

1. **Problem definition:**

When reading the problem statement, the input and output components are easily identified, because they use descriptive words such as nouns and adjectives. The processing component is also identified easily. The problem statement usually describes the processing steps as actions, using verbs and adverbs.

When dividing a problem into its three different components, you should simply analyse the actual words used in the specification, and divide them into those that are descriptive and those that imply actions. It may help to underline the nouns, adjectives and verbs used in the specification.

In some programming problems, the inputs, processes and outputs may not be clearly defined. In such cases, it is best to concentrate on the outputs required. Doing this will then decide most inputs, and the way will be set for determining the processing steps required to produce the desired output.

At this stage, the processing section should be a list of what actions need to be performed, not how they will be accomplished.

2. **The Structure Theorem:**

The Structure Theorem revolutionised program design by establishing a structured framework for representing a solution algorithm. The Structure Theorem states that it is possible to write any computer program by using only three basic control structures that are easily represented in pseudocode: sequence, selection and repetition. The three basic control structures:

2.1 **Sequence:**

The sequence control structure is the straightforward execution of one processing step after another. In pseudocode, we represent this construct as a sequence of pseudocode statements.

```
statement a
statement b
statement c
```
The sequence control structure can be used to represent the first four basic computer operations listed previously: to receive information, put out information, perform arithmetic, and assign values. For example, a typical sequence of statements in an algorithm might read:

```
Add 1 to pageCount
Print heading line1
Print heading line2
Set LineCount to zero
Read customer record
```

These instructions illustrate the sequence control structure as a straightforward list of steps written one after the other, in a top-to-bottom fashion. Each instruction will be executed in the order in which it appears.

### 2.2 Selection
The selection control structure is the presentation of a condition and the choice between two actions, the choice depending on whether the condition is true or false. This construct represents the decision-making abilities of the computer and is used to illustrate the fifth basic computer operation, namely to compare two variables and select one of two alternate actions.

In pseudocode, selection is represented by the keywords `IF, THEN, ELSE` and `ENDIF`:

```
IF condition p is true THEN
    statement(s) in true case
ELSE statement(s) in false case
ENDIF
```

If condition p is true, then the statement or statements in the true case will be executed, and the statements in the false case will be skipped. Otherwise (the ELSE statement) the statements in the true case will be skipped and statements in the false case will be executed. In either case, control then passes to the next processing step after the delimiter `ENDIF`. A typical pseudocode example might read:

```
IF student_gender is female THEN
    add 1 to female_count
ELSE
    add 1 to male_count
ENDIF
```

### 2.3 Repetition
The repetition control structure can be defined as the presentation of a set of instructions to be performed repeatedly, as long as a condition is true. The basic idea of repetitive code is that a block of statements is executed again and again, until a terminating condition occurs. This construct represents the sixth basic computer operation, namely to repeat a group of actions. It is written in pseudocode as:
**DOWHILE** condition $p$ is true

statement block

**ENDDO**

The **DOWHILE** loop is a leading decision loop - that is, the condition is tested before any statements are executed. If the condition in the **DOWHILE** statement is found to be true, the block of statements following that statement is executed once. The delimiter **ENDDO** then triggers a return of control to the retesting of the condition. If the condition is still true, the statements are repeated, and so the repetition process continues until the condition is found to be false. Control then passes to the statement that follows the **ENDDO** statement. It is imperative that at least one statement within the statement block alters the condition and eventually renders it false, because otherwise the logic may result in an endless loop.

Here is a pseudocode example that represents the repetition control structure:

```
Set student total to zero
DOWHILE student_total < 50
   Read student record
   Print student name, address to report
   Add 1 to student_total
ENDDO
```

This example illustrates a number of points:

- The variable student-total is initialised before the **DOWHILE** condition is executed.
- As long as student - total is less than 50 (that is, the **DOWHILE** condition is true), the statement block will be repeated.
- Each time the statement block is executed, one instruction within that block will cause the variable student total to be incremented.
- After 50 iterations, student-total will equal 50, which causes the **DOWHILE** condition to become false and the repetition to cease.

It is important to realise that the initialising and subsequent incrementing of the variable tested in the condition is an essential feature of the **DOWHILE** construct.

**Conclusion:**

The Structure Theorem states that it is possible to write any computer program by using only three basic control structures: sequence, selection and repetition. Each control structure was defined, and its association with each of the six basic computer operations was indicated.
## PART B

**INTRODUCTION TO PROGRAMMING CONCEPTS**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Introduction to C++</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Variables, Constants and Types.</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Input and Output.</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Assignment Statements.</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Control structures: selection.</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Control structures: iteration.</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Subprograms: procedures and functions.</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Data structure: Arrays.</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>structures in C++</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction to C++ Programming Language

This chapter is intended as a first introduction to programming computers using the C++ programming language.

1. A simple C++ program

// Sample program : 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

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.

The line #include <iostream.h> 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 is required to use stream input or output. This statement is a compiler directive -- that is it gives information to the compiler but does not cause any executable code to be produced.

The actual program consists of the procedure 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 procedure, while the closing brace (}) indicates the end. The word void indicates that main does not return a value.

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

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.
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.

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.

2. General form of a C++ Program

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

#include <iostream.h>

void main()
{
    constant declarations
    variable declarations
    executable statements
}
```
1. Variables and Identifiers:

1.1 What is a variable?
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.

Global variable and local variable: the answer should focus on the difference in scope between the two types of variables. The answer should describe instances where each type of variable would be suitable and also use an example to illustrate the points being made. Global variables are available for use anywhere in the program whereas local variables can only be used by the module/procedure/function in which they have been defined. Local variables are used in control constructs such as for loops. Global variables can be used for declaring constants.

1.2 Identifiers:
In order to distinguish between different variables, they must be given identifiers, names which distinguish them from all other variables.
An identifier must start with a letter and consist only of letters, the digits 0-9, or the underscore symbol _ (but the use of two consecutive underscore symbols, __, is forbidden.)
An identifier must not be a reserved word. Reserved words are valid identifiers that have special significance to C++.
The following are valid identifiers:

```
length days_in_year
DataSet1
Profit95
Int_Pressure
first_one
first_1
```

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. 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.
1.3: C++ Reserved words:

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

2. Types:

In C++ 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.
- 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.

There are four elementary variable types to considered, namely, int, float, bool and char. These types hold values as follows:

- **int**: represents 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**: represents 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 a range of about \(-10^{38}\) to \(10^{38}\). It is important to note that\textbf{ float values are only represented approximately}.

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

- **char**: represents 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;
```

The above statements declare 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 variable-identifier
```

3. Constants and Declaration

Often in programming numerical constants are used, e.g. the value of \textit{vat=21.0}. It is well worthwhile to associate meaningful names with constants. These names can be associated with the appropriate numerical value in a \textbf{constant declaration}. The names given to constants must conform to the rules for the formation of identifiers as defined above. The following constant declaration defines an integer constant \textit{days_in_year} which has the value 365:

```
const int days_in_year = 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:

\[
\textit{const type constant-identifier = value};
\]

\textit{type} is the type of the constant, \textit{constant-identifier} is the identifier chosen for the constant, which must be distinct from all identifiers for variables, and \textit{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:

\[
\textit{const float VatRate = 21};
\]

This defines a constant \textit{VatRate} to have the value 21. 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 \textit{VatRate} at the one place in the program. This of course only works if the constant identifier \textit{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 \textbf{of the same type} can be declared in the same constant declaration by separating each declaration by a comma. Thus:

\[
\textit{const int days_in_year = 365, days_in_leap_year = 366};
\]
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.

### 1. Inputs:

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

```cpp
cin >> number;
```

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

```cpp
cin input-list;
```

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

```cpp
cin >> n1 >> 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 and 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.

For example given the following declarations and input statement:

```cpp
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:
2. Outputs:

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;
Chapter 4: The Assignment Statement

1. Introduction

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 \textit{not} carried out.

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

The result of a division operator between two \texttt{int} operands is of type \texttt{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 \texttt{float} \( 1/7 \) would still be evaluated as an integer divided by an integer giving zero, which would then be converted to the equivalent \texttt{float} value, i.e. 0.0, before being assigned to the \texttt{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:

\[ i \% j = i - (\text{IntPart}(i / j)) \times j \]

Then \( 10\% -7 = 10 - (-1)*(-7) = 10 - 7 = 3 \), which means \( 10 = (-7)*(-1) + 3 = 7+3 \)

\[ 2. \text{Priority of Operators} \]

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

\[ a + b \times c \]

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

\[ (a + b) \times c \text{ or as } a + (b \times c) \]?
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
\[ a + b \times c \]
is evaluated as if it had been written as
\[ a + (b \times c) \]
because the * has a higher priority than the +. If the + was to be evaluated first then brackets would need to be used as follows:
\[ (a + b) \times c \]

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.

For example:

Write \( i = a + (-b/2 + c/4); \) Instead of \( i = a + -b/2 + c/4; \)

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:

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

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

3.1 Example Program 1: Temperature Conversion

The following program converts an input value in degrees Fahrenheit to the corresponding value in degrees Centigrade. Note how the constant \texttt{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 would return 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;
}

3.2. Example Program 2: 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 << endl << "Enter the amount in pence: " << endl;
    cin >> pence;
    pounds = pence / 100; // note use of integer division
    pence = pence % 100; // modulus operator -> remainder
    cout << pence << " pence is ";
    cout << pounds << " pounds and "
    << pence << " pence" << endl;
}

4. 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.

### 5. 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 illustrates the use of compound assignment operators.

\[ total \; += \; value; \; \; \text{or} \; total \; = \; total \; + \; value; \]
\[ prod \; *= \; 10; \; \; \text{or} \; prod \; = \; prod \; * \; 10; \]
\[ x \; /= \; y \; + \; 1; \; \; \text{or} \; x \; = \; x \; / \; (y \; + \; 1); \]
\[ n \; %= \; 2; \; \; \text{or} \; n \; = \; n \; \% \; 2; \]

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.
6. Formatting of output

When considering output in the previous sections 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++ using 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 \(6.7453 \times 10^{16}\) and a number like 0.0000000000001245 is better written as \(1.245 \times 10^{-13}\). 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++

```cpp
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:
\begin{verbatim}
cout << setw(7) << x << endl
    << setw(7) << y << endl;
\end{verbatim}
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.

7. 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 repeats the statements after it. In this case, \( i \) takes values 0, 1, 2, ... 16.

\begin{verbatim}
    // 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;
        }
    }
\end{verbatim}

Conclusion / 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.
• 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

\[ \text{variable op = expression} \]

which is identical in operation to

\[ \text{variable = variable op ( expression )} \]

• 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.
Chapter 5: Control structures: selection.

In order to produce algorithms, it must be possible to select the next statement to execute on the basis of some condition

1. The if statement

The if statement is the simplest form of conditional or selection statement in C++. The following if statement

\[
\text{if } (x > 0.0) \\
\quad \text{cout} << \text{"The value of x is positive"}; \\
\text{will print out the message } \text{"The value of x is positive"} \text{ if } x \text{ is positive.}
\]

The general form of the if statement is:

\[
\text{if } (\text{condition}) \\
\quad \text{statement}
\]

where \text{condition} is any valid logical expression or a \text{bool} variable. The \text{statement} can be a single C++ statement of any kind and must be terminated by a semi-colon. It can also be a \text{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.

The following if statement adds \( x \) to a variable \( \text{sum} \) if \( x \) is positive:

\[
\text{if } (x > 0.0) \\
\quad \text{sum} = \text{sum} + x;
\]

The following if statement also adds \( x \) to \( \text{sum} \) but in addition it adds 1 to a count of positive numbers held in the variable \( \text{poscount} \):

\[
\text{if } (x \geq 0.0) \\
\quad \{
\quad \text{sum} = \text{sum} + x; \\
\quad \text{poscount} = \text{poscount} + 1;
\quad \}
\]

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:

\[
\text{if } (x \geq 0.0) \\
\quad \text{sum} = \text{sum} + x; \\
\quad \text{poscount} = \text{poscount} + 1
\]

Then if \( x \) was greater than zero the next statement would be executed, that is \( x \) would be added to \( \text{sum} \). However the statement incrementing \( \text{poscount} \) 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:

```cpp
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:

```cpp
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;
```

2. 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:

```cpp
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:

```cpp
if (disc >= 0.0)
    cout << "Roots are real";
else
    cout << "Roots are complex";
```
The general form of the if-else statement is:

```cpp
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.

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.

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

3. 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:

```cpp
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 $0.25 \leq x < 0.50$, 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++;  

4. 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).

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.

```cpp
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:

```cpp
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";

5. Example Program: Wages Calculation

This program calculates 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
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.

Conclusion / 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.
- An if-else statement is used to choose which of two alternative statements to execute depending on the truth value of a condition.
- 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.
- 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.
Chapter 6: Control structures: iteration.

The iteration control structure can be defined as the presentation of a set of instructions to be performed repeatedly, as long as a condition is true.

1. 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.

```cpp
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:

```cpp
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.

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

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

**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:

```cpp
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;
```

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:

```cpp
i = 0;
while (i < n)
{
    i = i + 1;
```
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.

```cpp
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:

```cpp
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.

The following loop produces the sum $1+2+3+\ldots+n$, 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);

Example while-do loop: 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);

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;

Example Program: Student Mark Processing

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
while (candno >= 0);

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.

3. 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.

```cpp
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.

The equivalent for statement to the while statement is:

for (i = 1; i <= 10; i++)
    cout << i << endl;

It 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.

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.

```cpp
int i;
float x;
for (i = 0; i <= 16; i++)
{
    x = 0.1 * i;
    cout << x << "  " << sin(x) << endl;
}
```

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:

```cpp
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.
Conclusion / 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.

- 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.
- 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.
In a previous chapters we analysed a program 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.

### 1. Top-down design using Functions

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.
1.2. The need for functions

A rationale for the use of functions has been given above. Basically, the function can be comprehensively tested and hence a possible source of error is eliminated in future programs. These reasons are now expanded upon:

- 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 fewer errors. This function can now be tested on its own for correctness.

- In a particular organisation or industry it may be found that in carrying out the top-down approach 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.

- There are many 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.

1.2. The mathematical function library in C++

The following statement uses 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):

```cpp
cout << endl << "" << degree << ""
    << sin(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 PI
- **asin(x)**: inverse sine, \(-1 \leq x \leq +1\), returns value in radians in range 0 to PI
- **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 \(x\), \(x \geq 0\)
- **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, which 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, \(\text{exp}(x+1)\). The function call can then be used anywhere in an expression that an ordinary variable may be used. Hence the following examples:

```c++
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, \(\text{M_PI}\) can be used for \(\pi\) and \(\text{M_E}\) can be used for \(e\).
2. Top down design using functions

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.

```cpp
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;
    }
```

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 \(\text{dist}\) is used to
temporarily hold the calculated value inside the function.

The general form of a function definition in C++ is as follows:

```cpp
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 \(\text{int, float or char}\). If the function does not return a
value then the function-type must be \(\text{void}\).
- 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.

2.1 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.

```cpp
void skipthree(void)
    // skips three lines on output
    {
        cout << endl << endl << endl;
    }
```

Note that the function-type has been given as \(\text{void}\), 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 void functions have disadvantages:

- 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.
- When functions are used from a library of functions 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:

```c
void skipthree(void);
```

which would be included in the program file as follows:

```c
#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;
}
```

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.

### 2.2 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 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;
  }

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

2.3 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;
    }

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

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.

2.4 Example functions:

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

A typical use of sumsq is:
float sumsquare;
int number;
cout << "Enter number (>= 0): ";
cin >> number;
sumsquare = sumsq(number);
**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.

```c
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
```

A typical use of the `power` function is shown below

```c
float x, y;
int p;
cout << "Enter a float and an integer: ";
cin >> x >> p;
y = power(x, p);
y = power(x + y, 3);
```

### 2.5 Call-by-value parameters

Suppose the function `power` above is now amended to include the statement

```c
n++;
```

just before the final closing `)` and the following statements are executed:

```c
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.

2.6 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:

```cpp
// 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;
}
```
• 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.
• 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.
1. Introduction:

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:

```cpp
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 and then handling each value individually as follows:

```cpp
float a, b, c, d, e, f;
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;
```

This 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**

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.

2. 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:

```c
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,

```c
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.

3. 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 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:

```csharp
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:

```csharp
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
```csharp
    cin >> count[i];
```

//The element can be increased by 5,
```csharp
    count[i] = count[i] + 5;
```

//or, using the shorthand form of the assignment
```csharp
    count[i] += 5;
```

Array elements can form part of the condition for an if statement, or indeed, for any other logical expression:

```csharp
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.
```csharp
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.

```csharp
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, you might 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 \( k \leq NE \)).

\[
\text{sum} = 0.0; \\
\text{for (i = NE - k; i < NE; i++)} \\
\quad \text{sum += annual_temp[i];} \\
\text{av2 = sum / k;}
\]

**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 \( \text{annual_temp}[200] \), 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 \( \text{annual_temp}[200] \) 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.

4. 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:

\[
\text{int primes[]} = \{1, 2, 3, 5, 7, 11, 13\};
\]

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.

5. 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 notice 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:

```c
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++;
    }
}```
cout << "Number of values input is " << n;  
cout << endl 
<< "Number more than 10% above average is " 
<< nogt10 << endl;
}

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.

6. 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 the symbol “ & “ 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:
...

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;
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:

```cpp
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.

```cpp
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];
}
```

The function addarray could be used as follows:

```cpp
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.

7. Strings in C++

7.1 Definition:

So far the only form of character information used has been single characters which are defined as being of type char.

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:

```c
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:

```c
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.
7.2 String Output

A string is output by sending it to an output stream, for example:

```cpp
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.

7.3 String Input

When the input stream `cin` is used separators and terminators are required: there can be space characters, a new line or tab. Thus when inputting numeric data `cin` skips over any leading spaces and terminates reading a value when it finds a white-space character. 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:

```cpp
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 especially on streams 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 file streams
    ofstream outs;
    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.

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 indices 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.
A structure in C++ is analogous to a record in Pascal. A record can be thought of as a kind of complex variable declaration.

The structure in C++ and the record in Pascal encapsulate the same idea using a syntax defined for this purpose in the C++ language. C++ uses the keyword struct and the following syntax...

```cpp
struct MyData {
    char phonenum[8];
    bool married;
    int age;
    char name[20];
};
```

This struct statement creates the equivalent of a record in Pascal. This declaration defines the 'type' of the structure. In C++ you can create an instance of the structure by including the name of the variable after the closing curly braces and before the final semi-colon which completes the definition.

```cpp
struct MyData {
    char phonenum[8];
    bool married;
    int age;
    char name[20];
}; Datalist;
```

In the example above we have described the type of the structure and then we created an instance of the structure - a variable given the name Datalist. Note that C++ is case sensitive, so a variable named Datalist is not the same as a variable named datalist or DataList, whereas all three would refer to the same variable in Pascal, which is not case sensitive.

```cpp
MyData DataList;
```

This declares a variable DataList which is the type MyData, which means that the program allocates memory for the structure defined above, creating an instance of this record.

The fields of the record can then be accessed using the dot operator and standard C++ functions.

To copy a string into a field within the structure...
strcpy(Datalist.phonenum, "383-4455");

or to fill in other fields...

Datalist.age = 51;

Datalist.married = true;

And so on...

Instances structures can be declared as arrays of records, for example as part of a database application. Here we will declare an array of 20 records...

MyData Datalist[20];

Each individual record can then be accessed using array syntax combined with the dot operator used to access record fields...

Datalist[0].age = 51;

And so on...

Structures can also be filled in using the following syntax. Let's assume that the a record exists with a string field, and a couple of integer fields, and the structure name is Datalist.

Datalist = { "Some Name", 12, 102};