HARDWARE AND SOFTWARE

Early computers filled large rooms with tall metal racks on which were fixed thousands of vacuum tubes, tanks of mercury and panels of flashing lights. The resemblance to an ironmonger's store was so compelling that the computer engineers of the time wryly talked about their creations as hardware. Nowadays a considerable more powerful computer fits easily in a briefcase, but the principles of its operation remain the same.

The hardware of every digital computer consists of a processor, a store and an assortment of peripheral devices. The processor is the unit which actually performs the calculations. It contains a control unit to direct operations, as well as an arithmetic unit. The latter is equivalent to an electronic calculator, but much faster, being capable of a million or more operations per second. To make use of this speed the processor must be able to access its data equally quickly. Retaining data for rapid access by the processor is the job of the computer's store. Some calculators have a handful of registers or locations in which numbers can be kept. The store of a modest computer contains millions of locations. A calculator's numeric keys and display correspond to the peripherals or input/output devices of a computer. These allow data to be placed in the store and results to be taken out. Though very fast by human standards, peripherals are usually much slower than the processor and store.

A calculator is given instructions by pressing its function keys. However the great speed of a computer would be wasted if it could not be supplied with instructions as quickly as it obeys them. To make this possible the computer's instructions, encoded in numerical form, are held in store along with the data. The computer works in a cycle as follows.

1) The control unit fetches the next instruction from store.
2) The instruction is decoded into electronic signals by the control unit.
3) In response to these signals the arithmetic unit, the store, or a peripheral device carries out the instruction.
4) The whole cycle repeats from step 1.

In this way long sequences of instructions can be obeyed automatically at the full speed of the processor. Such a sequence of instructions is called a program.

Individual computer instructions are very simple in their effect, the following being typical.

a) Read an item of data into store from an input device.
b) Copy an item of data from one location to another.

c) Add the contents of two locations and place the sum in a third.
d) If the content of a location represents a negative number, take the next instruction from a different part of the program; otherwise continue with the next instruction in sequence.
e) Write an item of data from store to an output device.

When you think of a computer, you probably think in terms of electronics and silicon chips. But non-electronic computers are possible. Mechanical computers, made from gears and levers, were designed during the nineteenth century by and English mathematician, Charles Babbage. Whether powered by electricity, air or water, any computer has the same basic structure: processor, store and peripherals. It is the functions performed by these units, and their interconnections, that make a machine a computer rather than a radio or a windmill. In fact the idea of structure is an essential one in computing. You will see it applied to computers themselves, to the programs they run and to the data they process.

Many different programs can be written to perform a given task. They may run faster or slower, use more or less store or be intended to run on different makes of a computer. But, so long as they follow the same general method, we say that they are all implementations of the same algorithm. An algorithm is a method for carrying out a specific calculation that is guaranteed to produce a result in a finite number of steps. These steps may be simple as the machine instructions described above, or even more complex. Even when an algorithm for computing a certain result is known to exist, it may not be feasible to use it in practice. For example, it may be too slow, or need too much store. An example of this would be an algorithm to play a perfect game of chess. Methods that usually work, but are not guaranteed, are called heuristics.

Computer programs are working models of algorithms and heuristics. An important topic in computer science is the invention and study of algorithms and heuristics. The collection of all programs available in a computer system constitutes its software. This word was invented to emphasize that the programs are just as important as the hardware. It also contrasts them effectively. Hardware is visible, solid and substantial; software is somewhat intangible.

One of the most important parts of software is the operating system, a set of control programs which are kept permanently in store. The operating system carries out many of the routine tasks needed to prepare and run a user's program; for example deciding which program to run next, making ready its input, bringing the program into store, allocating it some processor time etc.

PROGRAMMING LANGUAGES

The earliest computers were programmed in machine code: in other words, by giving them instructions directly in numerical form. However the drawbacks were soon recognized.

a) Because of the primitive nature of machine instructions, machine-code programming is both tedious and error prone.
b) For the same reason, machine-code programs are difficult to understand and modify.
c) Programming is a time consuming and expensive business. It would be a great saving to be able to use the same program on different computers, but a machine-code program is specific to one model of computer and will not work on any other.

For these reasons machine code is now seldom used.

Why not write programs in English? Computer programming is not unique in needing to describe sequences of actions precisely and in detail. The same requirement is often met in daily life. However, anyone who has struggled with the sometimes mystifying instructions in car maintenance handbooks or recipe books will agree that English is not ideally suited to the task. In fact the glories of the English language – its vast scope, its subtlety, its potential for ambiguity and metaphor – must be considered severe disadvantages when the aim is literalness, accuracy and completeness. In other words English is the extreme opposite of machine code and precisely for that reason must be rejected as a medium for practical computer programming. What is needed is a middle way: one which combines the readability and generality of English with the directness and precision of machine code. Languages of this sort are called high level programming languages.

Any language can be studied from two points of view: that of its grammar, or syntax, and that of its meaning, or semantics. A good understanding of both is needed to use it properly.

A program written in a high level programming language cannot be directly performed by the hardware of a computer. To make it executable it must be translated from the high level programming language into an equivalent set of machine code instructions. This translation can be specified rigorously enough to make it a suitable task for a computer. Three programs are involved here: The translator program, or compiler; the source program written in a high level programming language and the equivalent machine code or object program. Thus a high level programming language is run in two distinct stages:

a) Compilation. The high level programming language compiler is brought into store and obeyed.

It causes the computer to read the source program, check it for errors, and convert it into the corresponding object program.

b) Execution. The object program resulting from stage (1) above is brought into store. It is obeyed in turn and reads input, performs computations, and writes output, in exactly the manner specified by the High level language source program. This stage can be repeated as often as necessary. Recompilation is needed only when the source program is changed.

Programs often contain errors in the use of the programming language and these are reported by the compiler during stage (1). The report usually takes the form of an error message or a number which refers to a list of error messages. These error messages are often helpful in finding the cause of the trouble. However, the compiler may be misled by an error into taking later, perfectly correct, parts of a program as erroneous. Thus one genuine error can cause a whole group of messages to be output, many of which are spurious.

The compilation phase, during which the program is translated from source code to object code, is often called compile time. The execution phase, when the object program is running, is often called run time or execution time.

PROGRAMMING METHODICALLY

It is one thing to read a simple example program and understand what it does. But the real question is, how do we go about writing a program from scratch, to satisfy a given problem specification? For very small programs, a little careful thought, coupled perhaps with a flash or two of insight, might be sufficient to produce a rough solution. This can be tried out on a computer. The program is likely to contain errors, however, and these have to be discovered and corrected. Often this is tedious and haphazard.

Such an ad hoc approach might be all right for very small programs consisting of few lines of code. But when we attempt to write larger programs, a more methodical approach becomes essential. We now consider how to achieve this.

1) First of all we must be clear about our objectives in writing programs. A program must, above all, be correct. There can be no argument about this: correctness is the overriding, indisputable

objective. A program that produces wrong results is unacceptable.

2) A second objective is efficiency. It is little use having a program that calculates a weekly payroll perfectly correctly if it takes nine days to process the data! Nor would a cash-dispenser program or an airline reservation program be very useful if it took an hour to perform each transaction.
3) A third objective is that program should be easy to modify. Modifiability is important because, during the life of the program, it might have to be adapted to deal with changes in the real world. Associated with this objective is the readability objective. This is dependent on the programming style. A readable program is relatively easy to understand and hence to modify.

In many cases the objectives of efficiency, modifiability and readability must be traded off one against the other, until a working compromise is achieved. However there can be no compromise about correctness.

These objectives are not easy to achieve. You are faced not only with the problem of getting the program to work correctly, which requires thorough testing. You must also bear in mind that, in the future, the program will have to be modified in unforeseen ways. And, as if that is not enough, you must also make sure the program is reasonably efficient. The task is daunting. For that reason, a wise programmer adopts any methods that will simplify the task. Several different techniques of methodical programming have been devised. Really, the particular method chosen matters less than the fact that it is used consistently.

A method I advocate for that is both simple and effective is known as stepwise refinement or top-down program design. The basic idea of stepwise refinement is to break a problem down into a set of smaller sub problems. Assume for the moment that each of these sub problems will eventually be solved. Then we have achieved an outline solution to the original problem. We now examine each of the sub problems in turn. If a sub problem is trivial, then we can write down its solution directly. If a sub problem is non-trivial, then we break it down further, into a set of smaller sub problems. If all the new sub problems can eventually be solved, then we have achieved a more detailed outline solution. We repeat the refinement process as long as necessary to reduce the original problem down to trivial sub problems. This completes the solution of the original problem.

The question we must now examine is how do we choose refinements? Although choosing refinements becomes easier with practice, like many skills, the novice needs some guidelines to get

started. The first point to remember is to look for familiar themes. Some common themes (for example sorting and searching) crop up again and again s sub problems. If you identify a sub problem that you have previously solved, you can use the same solution again. A word of warning, though. Sometimes a small difference in the problem requires a very different solution. Always guard against using a stock solution without carefully thinking about whether it is suitable for the current problem.

Every solution is composed from three basic structures, which will be discussed a little later. They are the sequence, selection (conditional) and repetition. There is no problem that cannot be solved in terms of these structures. When deciding between sequential, selection (conditional) and repetitive structures for a refinement, it is often helpful to take into account the structure of the input that must be accepted and the output that must be generated. Sometimes the structure of the input will determine the control structure (also to be discussed a little later) to be used. Sometimes the structure of the output is the important factor. Whatever its form, a good refinement usually has the following desirable properties:

a) Modularity. A program is said to be modular if its various components are relatively independent of each other. Choose refinements whose components interact as little as possible. This reduces the complexity of the refinements to be made at the next level down.
b) Localization. A program displays good localization when related statements are written close to each other in the text. This improves readability by reducing the need to scan back and forth through a program to see how things connect together.
c) Consistency. When you have to the same sort of thing several times, do it the same way on each occasion (unless there is good reason to vary).
d) Delayed decisions. Choose refinements which incorporate only those design decisions that cannot be postponed. This avoids committing the design prematurely to specific implementation ideas, increases the ease with which modifications can subsequently be made and reduces the distracting effect of minutiae.
e) Simplicity. It is very difficult to keep in mind more than about half a dozen facts or inter-relationships. Avoid overloading yourself with complex refinements and this will reduce the likelihood of confusion and error.

From time to time every programmer finds that ideas stop flowing and a difficult problem blocks all further progress. When this happens, suspect that a design error at a higher level has set you

an impossible task. However, if the design seems to be sound, there are a few things worth trying. First of all look for a solution by someone else. This may be found in the literature of the subject, or there may be a readymade program in a software library. Even if no solution of exactly the current problem turns up, you may still be able to find solutions to closely related problems. These are fertile sources of ideas. Failing that, try to simplify the problem, solve the simpler problem and then work back to the original.

It is important to use a consistent notation for stepwise refinement and to be thorough about writing down design decisions. In this way the written record of the design process acts as an extension of our limited memory, and allows more factors to be taken into account than would otherwise be possible. Each program outline is written in terms of a set of actions and expressions whose implementation has not yet been determined. These actions and expressions should be given descriptive names that will suggest exactly what the program is doing, even to a casual reader of the program outline. Accuracy is much more important than conciseness, because a misleading or vague name is both a symptom and cause of sloppy thinking.

In programming there are many opportunities for errors. Even a modest sized program is a complicated piece of text. We can cause it to fail by omitting or inserting a single punctuation mark. There are two kinds of errors, namely:

1) Syntax errors. A syntax error means that the program or part of it is ill formed; it does not follow the grammatical rules of the programming language. Syntax errors are discovered at compile time by the compiler which enforces the grammatical rules of the programming language.
2) Semantic errors. A semantic error occurs when the program is syntactically correct but the logic or structure of the program is wrong. A popular example is dividing a number by zero, where as the syntax is not wrong, dividing by zero will produce an error that the compiler will not detect. Most Semantic errors are not discoverable at compile time but will be discovered at run time.

Through testing we can ensure our programs do not have mistakes. The traditional jargon for errors in programs is bugs. Therefore testing and removing bugs is also called debugging the program. A programmer will often miss errors by assuming that the program does what he/she wanted it to do rather than what he actually programmed it to do! More comprehensive tests are indispensable. Thus the program must be tested on a computer, with large and varied sets of data. Another way of testing is

to do a trace which simply means to follow the execution of the program by hand. This however is tedious for large and complex programs and is unreliable. Modern software development tools enable a programmatic approach to tracing and this is far more reliable than doing it by hand.

PROGRAM DEVELOPMENT LIFE CYCLE

In the early days of the computer era, there were no well defined procedures for managing the software development process. The use of computers was limited to scientific and engineering applications. The programs were written for well defined problems such as calculating missile trajectories.

The Program Development Life Cycle (PDLC) was introduced to provide an organized plan for breaking down the task of program development into distinct manageable phases, each of which must be successfully completed before moving onto the next phase.

As mentioned, the PDLC is a systematic way of developing quality software. The PDLC is divided into the following phases:

1) Defining the problem.
2) Designing the program.
3) Coding the program.
4) Testing and Debugging the program.
5) Documenting the program.
6) Implementing and Maintaining the program.

Defining the Problem

The first step in developing a program is to define the problem. In major software projects, this is the job of the System Analyst, who then provides the results of his work to programmers in the form of a program specification. The program specification precisely defines the input data, the processing that should take place, the format of the output and the user interface. Depending on the size of the job, the program development might be handled by an individual or a team. Defining the problem entails figuring out exactly what the problem domain is and clarifying any ambiguity in the problem statement.

Designing the Program

An architect does not design a house just by going to the site with a bunch of boards and bricks. After determining the needs of the house's owner, the next step in designing a house is an architectural drawing – a plan drawn on paper that can be discussed and reviewed until everything is right. Likewise programmers create program designs. Like an architect's drawing, the program design specifies the components that make the program work.

Program design begins by focusing on the main goal that the program is trying to achieve and then breaking the program into manageable components, each of which contributes to this goal. This approach as covered under the programming methodically heading is called top-down program design or stepwise refinement. The first step involves identifying the main routine which is the program's major activity. From here the programmer tries to break down the main routine into smaller units called modules, until each module is highly focused. Experience shows that this is the best way to ensure program quality. If an error appears in a program designed this way, it is relatively easy to identify the module that must be causing the error.

For each module, the programmer draws a conceptual plan using an appropriate program design tool to visualize how the module will do its assigned job. The various design tools are:

a) Structure charts. A structure chart, also called a hierarchy chart, shows the top-down design of a program. Each box in the chart indicates a task that the program must accomplish.
b) Algorithms. An algorithm is a step by step description of how to arrive at a solution
c) Flowcharts. A flowchart is a diagram that shows the logic of the program. Programmers create flowcharts either by hand using a flowchart template or on the computer. More on flowcharts a little later.
d) Pseudocode. Pseudocode is an algorithm expressed in a programming language like construct but is not the actual programming language.

Coding the Program

Coding the program involves translating the algorithm into specific program language instructions. The programmer must choose an appropriate programming language and then create the program by typing the code. The programmer must carefully follow the language's rules, which precisely specify how to express certain operations. Violation of language rules results in grammatical errors, more precisely known syntax errors. These syntax errors must be eliminated before moving to

the next phase

Testing and Debugging the Program

After the removal of syntax errors, the program will execute. However the output of the program may not be correct. This is because of the existence of logical errors in the program. A logical error is a mistake that the programmer made while designing the solutions to the problem. A program development tool, such as a compiler, cannot detect these errors. Therefore, the programmer must find and correct logical errors by carefully examining the program output based on a set of input data for which the output is known. This type of data is called test data.

If the software developed is a complex one and consists of a large number of modules, then testing must be performed on these modules separately. This is known as unit testing. Once each module is thoroughly tested, they are then integrated together to form a software package. Testing is then performed on the software package to uncover any sort of error that may have been as a result of integration of different modules. This is known as system testing.

Documenting the Program

After testing, the software project is almost completed. The structure charts, pseudocode and flowcharts developed during the design phase become documentation for others who are associated with the software project. In addition, more documentation in the form of lists of variable names and variable definitions, description of files needed for the program to work and format of output produced by the program. All of this documentation needs to be saved and placed together for future reference.

This phase ends by writing a manual that provides an overview of the program's functionality, tutorials for the beginner, in depth explanations of major program features, reference documentation of all program commands and a thorough description of the error messages generated by the program. This manual is given to the user when the program is installed.

Implementing and Maintaining the Program

In the final phase, the program is installed at the user's site. The program is kept under watch till the user gives the green light to use it. Users may discover errors that were not caught in the testing phase, no matter how exhaustively the program was tested.

Even after the software project is complete, it needs to be maintained and evaluated regularly.

In program maintenance, the programming team fixes program errors that users discover during their day to day use of the program. In periodic evaluations, the team asks whether the program is fulfilling its objectives. This evaluation may lead to updates for the program, to either modify the working of a section of the program or addition of new features to the program. The evaluation can also lead to a decision to abandon the current system and develop a new one, and so the program development life cycle begins afresh.

ALGORITHMS, FLOWCHARTS AND PSEUDOCODE

Algorithms

An algorithm is a finite set of instructions having the following structure:

1) Precision – The steps in the algorithm are precisely stated, are clear and unambiguous.
2) Uniqueness – The intermediate results of each step of execution are uniquely defined and depend only on the inputs and the results of the preceding steps.
3) Finiteness – The algorithm stops after a finite number of steps.
4) Input – The program receives input.
5) Output – The algorithm produces output.

Example: An algorithm that finds the maximum of three numbers a, b, c.

1) *x:=a*
2) *if b>x, then x:=b*
3) *if c>x, then x:=c*

The idea of this algorithm is to inspect the numbers one by one and copy the largest value into a variable 'x'. At the conclusion of the algorithm, x will then be equal to the largest of the three numbers. Note - := is called an assignment operator, i.e. y:=z means copy the value of the z variable into the y variable or replace the current value of y with the value of z.

A simulation of the algorithm is known as a trace. For example, let's assume that a = 1, b = 5 and c = 3. The trace would go as follows:

1) trace started

2) set x to 1
3) if 5 is greater than x set x to 5 (*5 > x, set x to 5)*
4) if 3 is greater than x set x to 3 *(x > 3, do nothing)*
5) trace finished

The example algorithm has the properties of an algorithm:

a) The steps are precisely stated, they are clear and unambiguous.
b) The algorithm produces a unique result from the values of input. e.g. given the values a = 1, b = 5 and c = 3, the value of x at the end of the algorithm will always be set to 5 regardless of what person or machine executes the algorithm.
c) Finiteness – The algorithm stops after 3 steps and produces the largest of the three given values.
d) The algorithm receives input (a, b, c).
e) The algorithm produces output (x).

Pseudocode

Although ordinary language is sometimes adequate to specify an algorithm many mathematicians and computer scientists prefer pseudocode because of its precision, structure and universality. Pseudocode is so named because it resembles the actual code (programs) of high level languages such as Pascal and C.

Pseudocode, which is structured English, describes the algorithm using syntax like a programming language. Pseudocode is not as strict as programming languages which require a strict following of syntax and semantic rules. The only rule for pseudocode is that it is must be clear and unambiguous.

Example: Pseudocode for algorithm used to find the largest of three numbers a, b, c.

Input: a, b, c
Output: x, the largest of a, b, c

```
1) procedure max (a, b, c)
2)     x:=a
3)     if  b > x then // if b is larger than x, update x
4)         x:=b
5)     if c > x then //if c is larger than x, update x
```

```
6)      x:=c
7)    return x
8)  end max
```

Note: The first line contains the word procedure; the name of the procedure is max. In brackets we have what is called parameters for procedure max. Parameters describe the data, variables, arrays etc that are supplied as input into the procedure. In our case we supplied the procedure with three variables containing numbers (a, b, c).

The two forward slash marks denote a one line comment. Comments are used to clarify and make the code more readable. A comment is not executable. The last line lets us know that we have reached the end of the procedure.

Lines 3 and 5 highlight the selection construct. In general the selection construct looks as follows:

```
1)  if p then
2)    action
```

If condition p in line one is true then the action in line two is executed. If false then the action is skipped. Whether true or false the next statement following action is executed. More on the selection construct later.

You will notice that we have used indentation to identify the statement in the pseudocode that are executed if the expression being tested evaluates to true. However should there be more than one statement to execute, delimit the multiple statements with the words begin and end as follows:

```
1)  if p then
2)    begin
3)      action one
4)      action two
5)      action three
6)    end
7)  continue with pseudocode
8)
```

action one, action two and action three will only be executed if p evaluates to true.

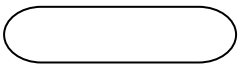
The return x statement on line seven terminates the procedure and returns the value of x. The return without a value or statement simply terminates the procedure. If there is no return statement the procedure terminates just before the last line. A procedure that returns a value is known as a function.

Flowcharts

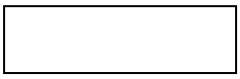
Flowcharts are used to represents any well formed sequence of activities. They are used to convey in diagrammatic form, the logic, processing operations and flow of control required by a computer program. Flow charts are used by programmers in two main ways:

a) To plan the structure of a program before it is written, by modeling an algorithm and/or pseudocode.
b) To describe the structure of a program after it has been written.

Flowcharts are made up of boxes of standard shapes linked to show the order of processing. Each box contains a brief note stating what the operation is.

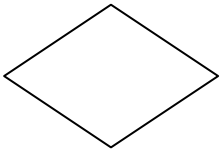


Start or End – Used to indicate the start or end of a flowchart. Note the curved edges

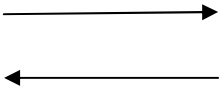


Process – Represents operations on data. Details are written in the rectangular box

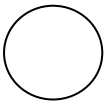
Input/Output – Input and output operations are shown in the parallelogram



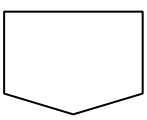
Decision – The diamond shape shows where a decision or test is to be made between two alternatives. The criteria is own inside the symbol.



Flow lines – Flow lines show the direction of processing in the flowchart



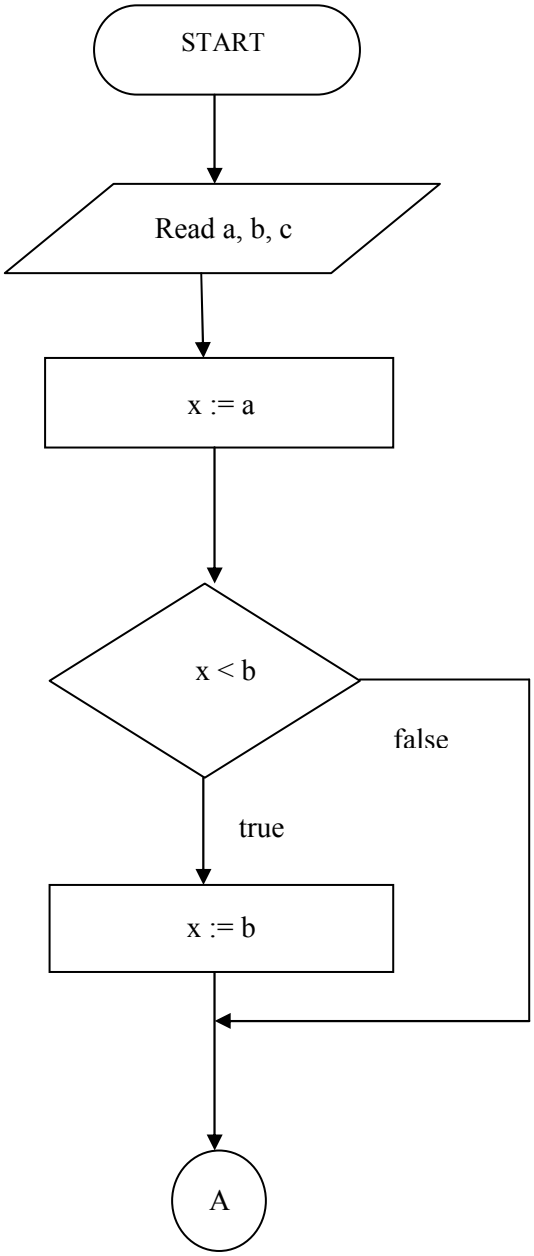
On page Connector – A small circle containing a number or a letter that is used to split a large flowchart into smaller parts on the same page

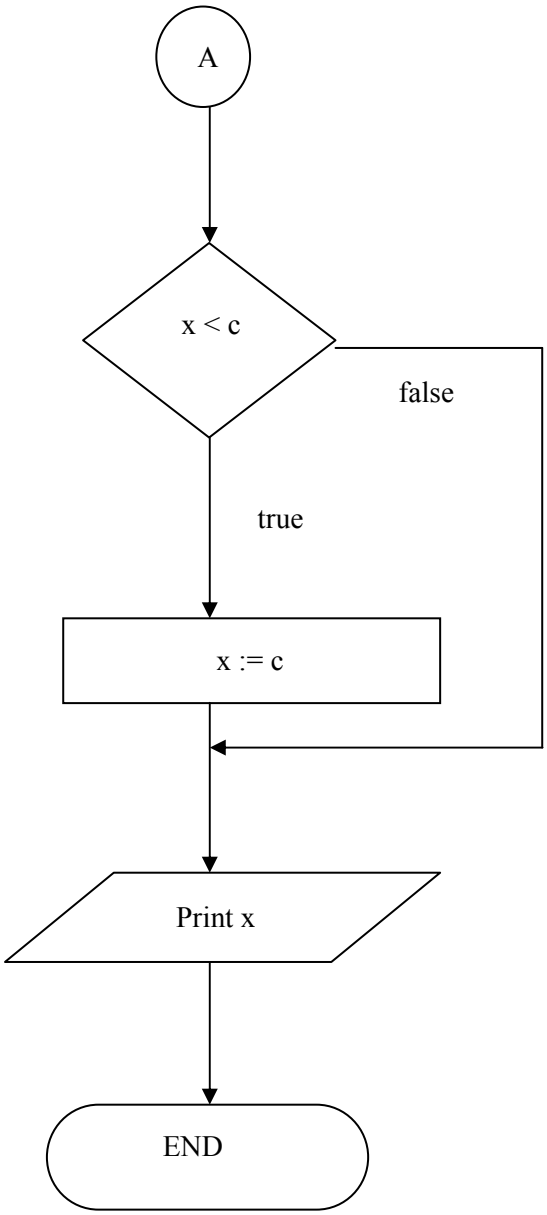


Off page Connector – A small shape containing a number or a letter that is used to split a large flowchart into smaller parts on different pages.

example: Flowchart for algorithm used to find the largest of three numbers a, b, c.

Assign largest number to variable x.





PROGRAM CODING

Coding Standards

There are two main functions performed by programming: the creation of programs and the maintenance of existing programs. The first one involves the initial writing of code and subsequent checking and amendment so that the program performs the required functions; the second one involves changing and adding of features as required.

Without control of any kind a programmer produces code which reflects his/her interest in the job, expertise and to some extent personality. The inexperienced programmer will usually use the simplest techniques to produce a solution, which may not always be the most efficient. As experience grows the programmer develops a style and uses some kind of shorthand that speeds up the production of code. This may be justified if the programmer is always going to be there to make changes to his/her program. Unfortunately, this is seldom the case, since programmers often move to other new programs and are not available to answer any queries. Furthermore, even if he/she remains working on the same program, when he/she looks at the code months, or even weeks later, the programmer is unlikely to recall what he/she did, how he/she did it and why. It is therefore relatively safe to assume that maintenance will be done by someone who has either never seen the code before or has little recall of it.

By laying down the rules to be followed, standards provide a way of ensuring that consistent quality is maintained throughout all areas of programming from design methods to coding style. Standards help inexperienced programmers to become effective more quickly and by making programs easier to comprehend, amend and check, they help the experienced programmers to become more productive. Standards aid communication between people working on the same project and increase the interchangeability of people of different aptitude, abilities and experience, within and between different projects.

A Readable Program

People often say that the readability of a program is more important than the intricacies of the code. Some of the rules that should be followed to make a program more readable are:

Data Names

Meaningful names make program code easier to read. It is much easier to understand and "student_name" as compared to "sn"

Comments

Comments help make the program code more understandable. All programming languages allow the use of comments in a program. Comments should be added at the module level to explain the purpose of the module and within the code to explain complicated algorithms or highlighting error-prone sections of the program. However it is not necessary to add comments to every line of code.

Indentation

Code should be layered out neatly, with the proper use of indentation to reflect the logic structure of the code. It is a good idea to place only one instruction on each line and make sensible use of blank lines to keep code readable, and limit components to a manageable size

Modularization

The aim of splitting up a program is to make it easier to understand. In addition to this, it also reduces the effort required to later change the program, since changes need only be made to the module whose functionality changes, or if a new feature is to be added then a new model can be added. Testing and debugging also becomes easier with small independent modules. Modules should observe the following rules:

a) Each module should contain only one program function.
b) There should only be one entry point and one exit point in each module.

STRUCTURED PROGRAMMING CONSTRUCTS

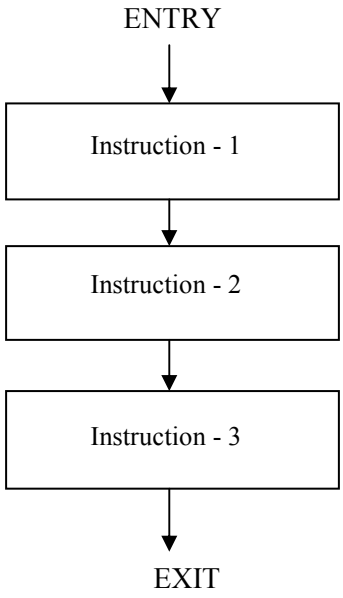
Algorithms and their equivalent programs are usually composed of one or more of the following programming constructs:

1. Sequential programming construct.

2. Selection programming construct.
3. Iterative programming construct.

Sequential Programming Construct

In the sequential programming construct, the steps are executed one after the other. Most processing, even complex problems will generally follow this elementary sequential flow of control. Sequential Programming construct has the following form:

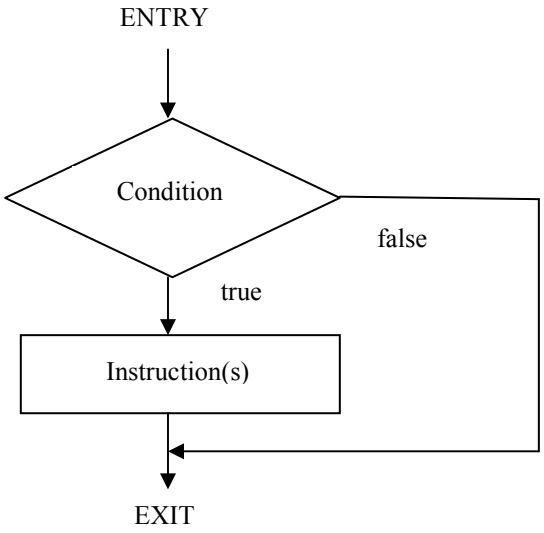


Selection Programming Construct

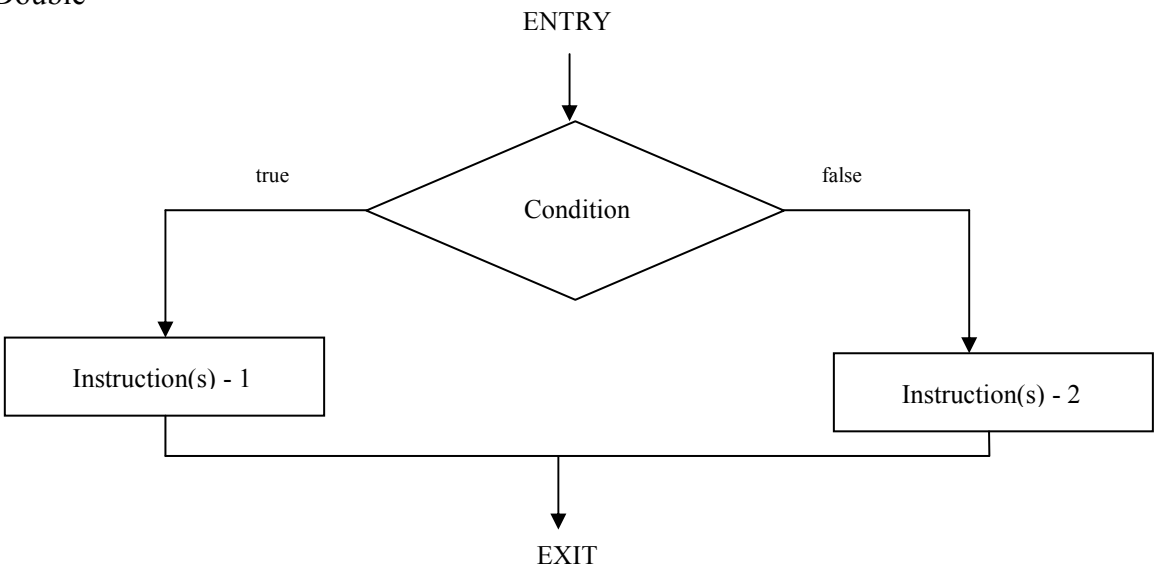
The selection programming construct employs conditions and depending on the outcome of the conditions, one of the alternatives is selected. This construct can fall into the following categories:

a) single alternative.
b) Double alternative.
c) Multiple alternative.

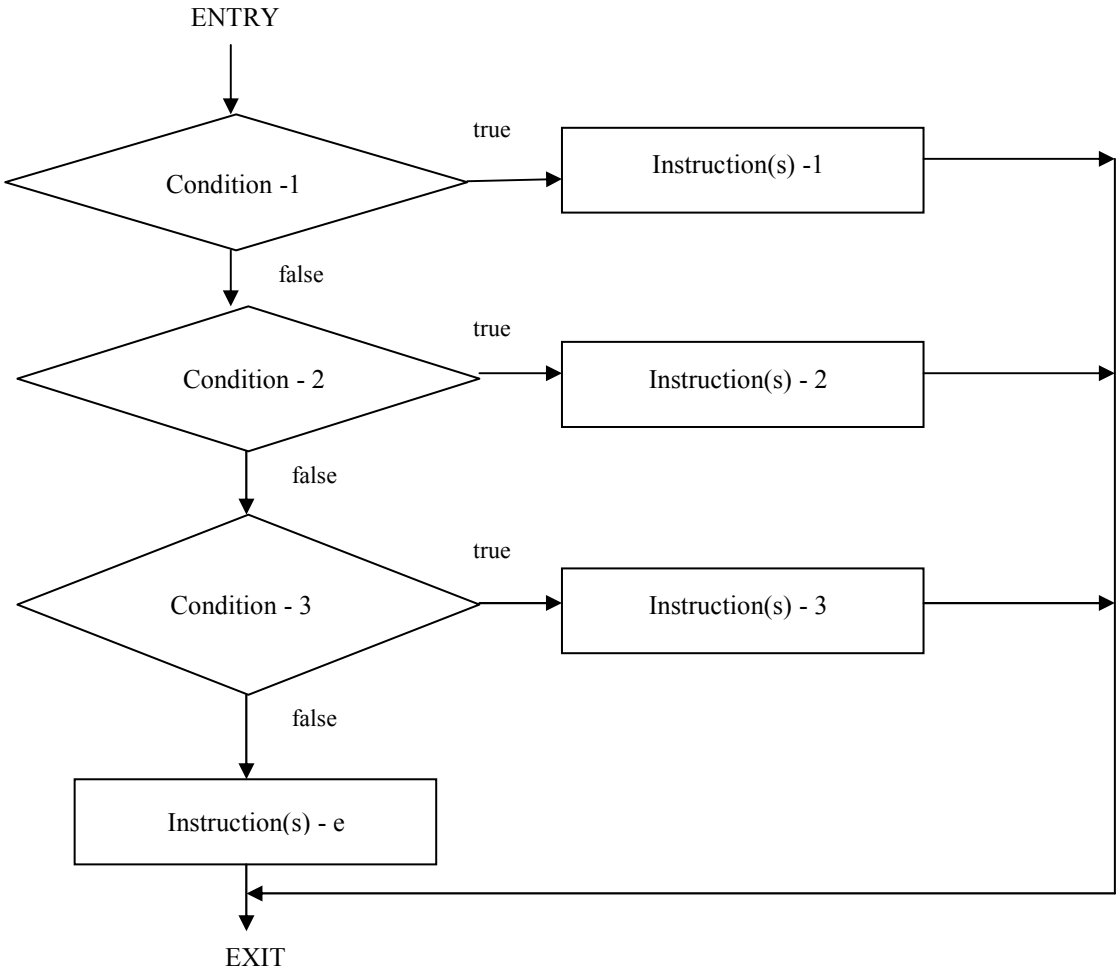
a) Single



b) Double



c) Multiple

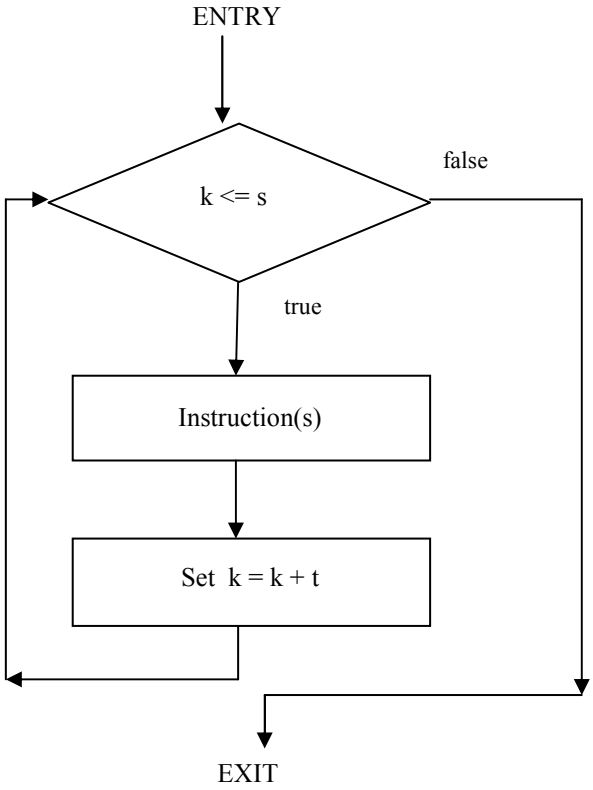


Iterative Programming Construct

The iterative programming construct refers to repeatedly executing a statement or a group of statements until some condition is met. Iterative programming constructs fall into the following categories:

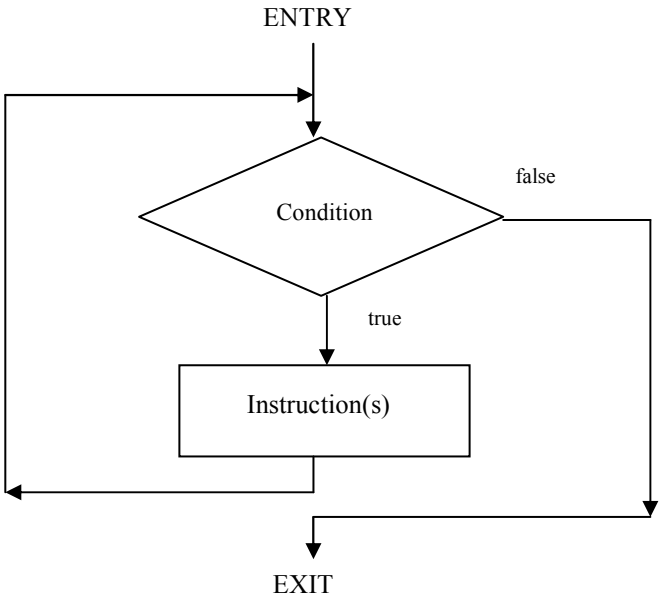
a) For construct.
b) While construct.
c) Do While construct.

a) For Construct

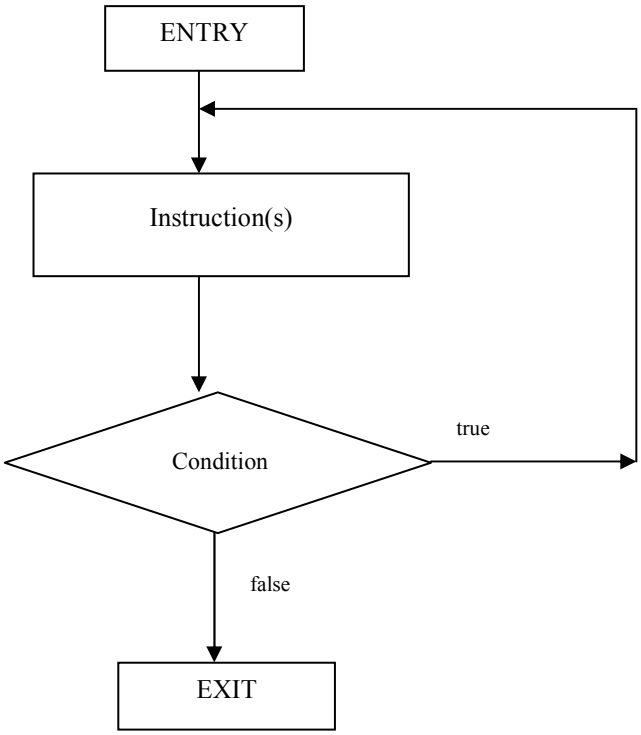


Note: k is called the index variable, it is set to an initial value. s is the final value and t is the step size.

b) While Construct



c) Do – While Construct



INTRODUCTION TO THE C PROGRAMMING LANGUAGE

History of C

The *milestones* in C's development as a language are listed below:

- UNIX developed 1969 – a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- BCPL -- DEC PDP-7 Assembly Language
- A new language ``B'' a second attempt. 1970.
- A totally new language ``C'' a successor to ``B''. 1971
- By 1973 UNIX OS almost totally written in ``C''.

Characteristics of C

Following is a list of some of C's characteristics that define the language and also have lead to its popularity as a programming language.

- Small size
- Extensive use of function calls
- Loose typing -- unlike PASCAL
- Structured language
- Low level (Bitwise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

What is an identifier?

Before you can do anything in any language, you must at least know how to name an identifier. An

identifier is used for any variable, function, data definition, etc. In the programming language C, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline. In the case of some compilers, a dollar sign is permitted but not as the first character of an identifier. It should be pointed out that even though a dollar sign may be permitted by your C compiler, it is not in general use by C programmers, and is not even allowed by most compilers. If you do not plan to write any portable code, you can use it at will if you feel it makes your code more readable.

Two rules must be kept in mind when naming identifiers.

1. The case of alphabetic characters is significant. Using **INDEX** for a variable name is not the same as using **index** and neither of them is the same as using **InDeX** for a variable name. All three refer to different variables.
2. According to the ANSI-C standard, at least 31 significant characters can be used and will be considered significant by a conforming ANSI-C compiler. If more than 31 are used, they may be ignored by any given compiler.

What About the Underline (Underscore)?

Even though the underline can be used as part of a variable name, and adds greatly to the readability of the resulting code, it seems to be used very little by experienced C programmers. Since most compiler writers use the underline as the first character for variable names internal to the system, you should refrain from using the underline to begin a variable to avoid the possibility of a name clash. To get specific, identifiers with two leading underscores are reserved for the compiler as well as identifiers beginning with a single underscore and using an upper case alphabetic character for the second. If you make it a point of style to never use an identifier with a leading underline, you will never have a naming clash with the system.

It adds greatly to the readability of a program to use descriptive names for variables and it would be to your advantage to do so. Pascal programmers tend to use long descriptive names, but most C programmers tend to use short cryptic names.

Keywords

There are 32 words defined as keywords in C. These have predefined uses and cannot be used for any other purpose in a C program. They are used by the compiler as an aid to compiling the program. They are always written in lower case. A complete list follows;

auto double int struct break else long switch case enum register typedef char extern return union const float short unsigned continue for signed void default goto sizeof volatile do if static while

In addition to this list of keywords, your compiler may define a few more. If it does, they will be listed in the documentation that came with your compiler. Each of the above keywords will be defined, illustrated, and used in this tutorial.

C Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Type definitions
- Function prototypes -- declare function types and variables passed to function.
- Variables
- Functions

A Very Simple Program

This program which will print out the message This is a C program

```
#include <stdio.h>

main ()
{
    printf ("This is a C program\n");
}
```

Though the program is very simple, a few points are worthy of note.

Every C program contains a function called main. This is the start point of the program.

#include <stdio.h> allows the program to interact with the screen, keyboard and file system of your computer. You will find it at the beginning of almost every C program.

main () declares the start of the function, while the two curly brackets show the start and finish of the function. Curly brackets in C are used to group statements together as in a function, or in the body of a loop. Such a grouping is known as a compound statement or a block.

printf ("This is a C program\n");

prints the words on the screen. The text to be printed is enclosed in double quotes. The \n at the end of the text tells the program to print a newline as part of the output.

Most C programs are in lower case letters. You will usually find upper case letters used in preprocessor definitions (which will be discussed later) or inside quotes as parts of character strings. C is case sensitive, that is, it recognizes a lower case letter and it's upper case equivalent as being different. While useful for teaching, such a simple program has few practical uses. Let us consider something rather more practical. The following program will print a conversion table for weight in pounds (U.S.A. Measurement) to pounds and stones (Imperial Measurement) or Kilograms (International).

A Weight Conversion Program

```
#include <stdio.h>
#define KILOS_PER_POUND .45359
main ()
{
        int pounds;

    printf (" US lbs     UK st. lbs      INT Kg\n");

    for (pounds=10; pounds < 250; pounds+=10)
    {
                    int stones = pounds / 14;
        int uklbs = pounds % 14;
        float kilos = pounds * KILOS_PER_POUND;
        printf ("   %d          %d   %d       %f\n",
                pounds, stones, uklbs, kilos);
    }
}
```

Again notice that the only function is called main.

int pounds; Creates a variable of integer type called pounds.

float kilos; Creates a floating point variable (real number) called kilos.

#define KILOS_PER_POUND .45359

defines a constant called **KILOS_PER_POUND**. This definition will be true throughout the program. It is customary to use capital letters for the names of constants, since they are implemented by the C preprocessor.

for (pounds=10; pounds < 250; pounds+=10)

This is the start of the loop. All statements enclosed in the following curly brackets will be repeated.

The loop definition contains three parts separated by semi-colons.

- The first is used to initialize variables when the loop is entered.
- The second is a check, when it proves false, the loop is exited.
- The third is a statement used to modify loop counters on each loop iteration after the first.

The effect of pounds += 10 is to add 10 to the value of the variable pounds. This is a shorthand way of writing pounds = pounds + 10.

The printf statement now contains the symbols %d and %f. These are instructions to print out a decimal (integer) or floating (real) value. The values to be printed are listed after the closing quote of the printf statement. Note also that the printf statement has been split over 2 lines so it can fit onto our page. The computer can recognize this because all C statements end with a semicolon.

Weight Conversion Table Using a Function

The previous program could be better structured by defining a function to convert the weights and print out their values. It will then look like this.

```
#include <stdio.h>

/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
void print_converted (int pounds)
{
        int stones = pounds / 14;
    int uklbs = pounds % 14;
    float kilos_per_pound = 0.45359;
    float kilos = pounds * kilos_per_pound;

    printf ("   %3d        %2d  %2d       %6.2f\n",
        pounds, stones, uklbs, kilos);
}

main()
{
        int us_pounds;
    printf(" US lbs     UK st. lbs      INT Kg\n");

    for(us_pounds=10; us_pounds < 250; us_pounds+=10)
        print_converted(us_pounds);
}
```

void print_converted(int pounds) is the beginning of the function definition. The line within the loop reading **print_converted(us_pounds)** is a call to that function. When execution of the main function

reaches that call, **print_converted** is executed, after which control returns to main.

The text enclosed by symbols /* and */ is a comment. These are C's way of separating plain text comments from the body of the program. It is usually good practice to have a short comment to explain the purpose of each function. The symbols /* and */ are used for multi line comments. For one line comments we can use the symbols //

Defining a function has made this program larger, but what have we gained? The structure has been improved. This may make little difference to the readability of such a small program. In a larger program, such structuring makes the program shorter, easier to read, and simplifies future maintenance of the program. Another benefit of defining a function is that the function can easily be re-used as part of another program.

Weight Conversion with a Prompt

To illustrate this, our next program will re-use the previous function. The program is similar to the last one, but instead of printing a table of weights, the user enters a weight, and this value is converted. Such a program requires user input. This can be done in two ways, both of which will be shown.

The simpler implementation is to prompt for a weight, and then read the user's keyboard input. This would be done as follows

```
#include <stdio.h>


/* Convert U.S. Weight to Imperial and International
   Units. Print the results */
void print_converted(int pounds)
{
            int stones = pounds / 14;
      int uklbs = pounds % 14;
      float kilos_per_pound = 0.45359;
      float kilos = pounds * kilos_per_pound;

      printf("   %3d        %2d  %2d       %6.2f\n",
            pounds, stones, uklbs, kilos);
}

main()
{
            int us_pounds;

      printf("Give an integer weight in Pounds : ");
      scanf("%d", &us_pounds);

      printf(" US lbs      UK st. lbs      INT Kg\n");
      print_converted(us_pounds);
}
```

A **printf** statement is used to prompt for input. **scanf** is an equivalent input statement, note that the variable to be read **us_pounds** is written as **&us_pounds** here. This is very important and it will be explained later.

Variables

As in most languages, a variable declaration reserves and names an area in memory at run time to hold a value of particular type. Syntactically, C puts the type first followed by the name of the variable. The following declares an int variable named "num" and the 2nd line stores the value 42 into num.

```
int num;
num = 42;
```

A variable corresponds to an area of memory which can store a value of the given type.
Variables, such as num, do not have their memory cleared or set in any way when they are allocated at run time. Variables start with random values, and it is up to the program to set them to something sensible before depending on their values.

Names in C are case sensitive so "x" and "X" refer to different variables. Names can contain digits and underscores (_), but may not begin with a digit. Multiple variables can be declared after the type by separating them with commas. C is a classical "compile time" language -- the names of the variables, their types, and their implementations are all flushed out by the compiler at compile time (as opposed to figuring such details out at run time like an interpreter).

```
float x, y, z, X;
```

In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function. Most local variables are created when the function is called, and are destroyed on return from that function.
A declaration begins with the type, followed by the name of one or more variables. For example,

```
int high, low, results[20];
```

Declarations can be spread out, allowing space for an explanatory comment. Variables can also be initialized when they are declared, this is done by adding an equals sign and the required value after the declaration.

```
int high = 250;     /* Maximum Temperature */
```

```
int low = -40;      /* Minimum Temperature */
int results[20];    /* Series of temperature readings */
```

C provides a wide range of types. The most common are

`int`	An Integer
`float`	A floating point (real) number
`char`	A single byte of memory, enough to hold a character

There are also several variants on these types.

`short`	An integer, possibly of reduced range
`long`	An integer, possibly of increased range
`unsigned`	An integer with no negative range, the spare capacity being used to increase the positive range
`unsigned long`	Like unsigned, possibly of increased range
`double`	A double precision floating point number.

All of the integer types plus the char are called the integral types. float and double are called the real types.

Variable Names

Every variable has a name and a value. The name identifies the variable, the value stores data. There is a limitation on what these names can be. Every variable name in C must start with a letter, the rest of the name can consist of letters, numbers and underscore characters. C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names.

Examples of legal variable names include

x result outfile bestyet x1 x2 out_file best_yet power impetus gamma hi_score

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants.

The rules governing variable names also apply to the names of functions. We shall meet functions later on in the course.

Global Variables

Local variables are declared within the body of a function, and can only be used within that function. This is usually no problem, since when another function is called, all required data is passed to it as arguments. Alternatively, a variable can be declared globally so it is available to all functions. Modern programming practice recommends against the excessive use of global variables. They can lead to poor

program structure, and tend to clog up the available name space.

A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessor directives. The variable is not declared again in the body of the functions which access it.

Arrays

An array is a collection of variables of the same type. Individual array elements are identified by an integer index. In C the index begins at zero and is always written inside square brackets. We have already met single dimensioned arrays which are declared like this

```
int results[20];
```

Arrays can have more dimensions, in which case they might be declared as

```
int results_2d[20][5];
int results_3d[20][5][3];
```

Each index has its own set of square brackets.

Where an array is declared in the main function it will usually have details of dimensions included. It is possible to use another type called a pointer in place of an array. This means that dimensions are not fixed immediately, but space can be allocated as required. This is an advanced technique which is only required in certain specialized programs.

When passed as an argument to a function, the receiving function need not know the size of the array. So for example if we have a function which sorts a list (represented by an array) then the function will be able to sort lists of different sizes. The drawback is that the function is unable to determine what size the list is, so this information will have to be passed as an additional argument.

As an example, here is a simple function to add up all of the integers in a single dimensioned array.

```
int add_array(int array[], int size)
{
        int i;
    int total = 0;

    for(i = 0; i < size; i++)
        total += array[i];

    return(total);
}
```

Expressions and Operators

One reason for the power of C is its wide range of useful operators. An operator is a function which is applied to values to give a result. You should be familiar with operators such as + (plus), - (minus), /

(divide), * (multiply).

Arithmetic operators are the most common. Other operators are used for comparison of values, combination of logical states, and manipulation of individual binary digits. The binary operators are rather low level for so are not covered here.

Operators and values are combined to form expressions. The values produced by these expressions can be stored in variables, or used as a part of even larger expressions.

Assignment Operator =

The assignment operator is the single equals sign (=).

i = 6;

i = i + 1;

The assignment operator copies the value from its right hand side to the variable on its left hand side. The assignment also acts as an expression which returns the newly assigned value. Some programmers will use that feature to write things like the following.

y = (x = 2 * x); //double x, and also put x's new value in y

Arithmetic operators

Here are the most common arithmetic operators

+ Addition
- Subtraction
* Multiplication
/ Division
% Modulo Reduction (Remainder from integer division)

*, / and % will be performed before + or - in any expression. Brackets can be used to force a different order of evaluation to this. Where division is performed between two integers, the result will be an integer, with remainder discarded. Modulo reduction is only meaningful between integers. If a program is ever required to divide a number by zero, this will cause an error, usually causing the program to crash. Here are some arithmetic expressions used within assignment statements.

velocity = distance / time;

force = mass * acceleration;

count = count + 1;

C has some operators which allow abbreviation of certain types of arithmetic assignment statements.

Shorthand	Equivalent
`i++;` or `++i;`	`i = i + 1;`
`i--;` or `--i;`	`i = i - 1;`

These operations are usually very efficient. They can be combined with another expression.

`x = a * b++;` is equivalent to
```
x = a * b;
b = b + 1;
```

Versions where the operator occurs before the variable name change the value of the variable before evaluating the expression, so

`x = --i * (a + b);` is equivalent to
```
i = i - 1;
x = i * (a + b);
```

These can cause confusion if you try to do too many things on one command line. You are recommended to restrict your use of ++ and - to ensure that your programs stay readable. Another shorthand notation is listed below

Shorthand	Equivalent
`i += 10;`	`i = i + 10;`
`i -= 10;`	`i = i - 10;`
`i *= 10;`	`i = i * 10;`
`i /= 10;`	`i = i / 10;`

These are simple to read and use.

Comparison

C has no special type to represent logical or boolean values. It improvises by using any of the integral types char, int, short, long, unsigned, with a value of 0 representing false and any other value representing true. It is rare for logical values to be stored in variables. They are usually generated as required by comparing two numeric values. This is where the comparison operators are used, they compare two numeric values and produce a logical result.

C notation	Meaning
`==`	Equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!=`	Not equal to

Note that == (two equal signs) is used in comparisons and = is used in assignments. Comparison operators are used in expressions like the ones below.

x == y

i > 10

a + b != c

In the last example, all arithmetic is done before any comparison is made.

These comparisons are most frequently used to control an if statement, a for loop or a while loop.

Logical Connectors

These are the usual And, Or and Not operators.

Symbol	Meaning
&&	And
||	Or
!	Not

They are frequently used to combine relational operators, for example

x < 20 && x >= 10

In C these logical connectives employ a technique known as lazy evaluation. They evaluate their left hand operand, and then only evaluate the right hand one if this is required. Clearly false && anything is always false, true || anything is always true. In such cases the second test is not evaluated.

Not operates on a single logical value, its effect is to reverse its state. Here is an example of its use.

```
if ( ! acceptable )
    printf("Not Acceptable !!\n");
```

Control Statements

A program consists of a number of statements which are usually executed in sequence. Programs can be much more powerful if we can control the order in which statements are run.

Statements fall into three general types;

- Assignment, where values, usually the results of calculations, are stored in variables.
- Input / Output, data is read in or printed out.
- Control, the program makes a decision about what to do next.

Control statements in C can be used to write powerful programs by;

- Repeating important sections of the program.
- Selecting between optional sections of a program.

The if else Statement

This is used to decide whether to do something at a special point, or to decide between two courses of action.

The following test decides whether a student has passed an exam with a pass mark of 45

```
if (result >= 45)
      printf("Pass\n");
else
      printf("Fail\n");
```

It is possible to use the if part without the else.

```
if (temperature < 0)
      print("Frozen\n");
```

Each version consists of a test, (this is the bracketed statement following the if). If the test is true then the next statement is obeyed. If it is false then the statement following the else is obeyed if present. After this, the rest of the program continues as normal.

If we wish to have more than one statement following the if or the else, they should be grouped together between curly brackets. Such a grouping is called a compound statement or a block.

```
if (result >= 45)
{
            printf(“Passed\n”);
            printf(“Congratulations\n”);
}
else
{     printf("Failed\n");
      printf("Good luck in the resits\n");
}
```

Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. In the following example we are awarding grades depending on the exam result.

```
if (result >= 75)
      printf("Passed: Grade A\n");
else if (result >= 60)
      printf("Passed: Grade B\n");
else if (result >= 45)
      printf("Passed: Grade C\n");
else
      printf("Failed\n");
```

In this example, all comparisons test a single variable called result. In other cases, each test may involve a different variable or some combination of tests. The same pattern can be used with more or fewer else if's, and the final lone else may be left out. It is up to the programmer to devise the correct structure for each programming problem.

The switch Statement

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
/* Estimate a number as none, one, two, several, many */
estimate(int number)
{
            switch(number) {
      case 0 :
            printf("None\n");
            break;
      case 1 :
            printf("One\n");
            break;
      case 2 :
            printf("Two\n");
            break;
      case 3 :
            printf("Several\n");
            break;
      default :
            printf("Many\n");
            break;
      }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.
Both if and switch constructs allow the programmer to make a selection from a number of possible actions.
The other main type of control statement is the loop. Loops allow a statement, or block of statements, to be repeated. Computers are very good at repeating simple tasks many times; the loop is C's way of achieving this.

The while Loop

The while loop repeats a statement until the test at the top proves false.
As an example, here is a function to return the length of a string. Remember that the string is represented as an array of characters terminated by a null character '\0'.

```
int string_length(char string[])
{
        int i = 0;
    while (string[i] != '\0')
        i++;

    return(i);
}
```

The string is passed to the function as an argument. The size of the array is not specified, the function will work for a string of any size.
The while loop is used to look at the characters in the string one at a time until the null character is found. Then the loop is exited and the index of the null is returned. While the character isn't null, the index is incremented and the test is repeated.

The do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{
        printf("Enter 1 for yes, 0 for no :");
    scanf("%d", &input_value);
}
while (input_value != 1 && input_value != 0)
```

The for Loop

The for loop works well where the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts separated by semicolons.

- The first is run before the loop is entered. This is usually the initialization of the loop variable.
- The second is a test, the loop is exited when this returns false.
- The third is a statement to be run every time the loop body is completed. This is usually an increment of the loop counter.

The example is a function which calculates the average of the numbers stored in an array. The function takes the array and the number of elements as arguments.

```
float average(float array[], int count)
{
        float total = 0.0;
    int i;

    for(i = 0; i < count; i++)
        total += array[i];

    return(total / count);
}
```

The for loop ensures that the correct number of array elements are added up before calculating the average.

The three statements at the head of a for loop usually do just one thing each, however any of them can be left blank. A blank first or last statement will mean no initialization or running increment. A blank comparison statement will always be treated as true. This will cause the loop to run indefinitely unless interrupted by some other means. This might be a return or a break statement.

It is also possible to squeeze several statements into the first or third position, separating them with commas. This allows a loop with more than one controlling variable. The example below illustrates the definition of such a loop, with variables hi and lo starting at 100 and 0 respectively and converging.

```
for (hi = 100, lo = 0; hi >= lo; hi--, lo++)
```

The for loop is extremely flexible and allows many types of program behavior to be specified simply and quickly.

The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

Prototypes

A "prototype" for a function gives its name and arguments but not its body. In order for a caller, in any file, to use a function, the caller must have seen the prototype for that function. For example, here's what the prototypes would look like for Twice() and Swap(). The function body is absent and there's a semicolon (;) to terminate the prototype

```
int Twice(int num);
void Swap(int* a, int* b);
```

In C,

1) a function may be declared static in which case it can only be used in the same file where it is used below the point of its declaration. Static functions do not require a separate prototype so long as they are defined before or above where they are called, this saves some work.

2) A non-static function needs a prototype. When the compiler compiles a function definition, it must have previously seen a prototype so that it can verify that the two are in agreement ("prototype before definition" rule). The prototype must also be seen by any client code which wants to call the function ("clients must see prototypes" rule).(The require-prototypes behavior is actually somewhat of a compiler option, but it's smart to leave it on.)

Preprocessor

The preprocessing step happens to the C source before it is fed to the compiler. The two most common preprocessor directives are #define and #include

#define

The #define directive can be used to set up symbolic replacements in the source. As with all preprocessor operations, #define is extremely unintelligent -- it just does textual replacement without understanding. #define statements are used as a crude way of establishing symbolic constants.

```
#define MAX 100
#define SEVEN_WORDS that_symbol_expands_to_all_these_words
```

Later code can use the symbols MAX or SEVEN_WORDS which will be replaced by the text to the right of each symbol in its #define.

#include

The "#include" directive brings in text from different files during compilation. #include is a very unintelligent and unstructured -- it just pastes in the text from the given file and continues compiling. The #include directive is used in the .h/.c file convention below which is used to satisfy the various constraints necessary to get prototypes correct.

```
#include "file.h" // refers to a "user" file.h file --
// in the originating directory for the compile
```

#include <file.h> // refers to a "system" file.h file --
// in the compiler's directory somewhere

file.h vs. file.c

The universally followed convention for C is that for a file named "file.c" containing a bunch of functions

• A separate file named file.h will contain the prototypes for the functions in file.c which clients may want to call. Functions in file.c which are for "internal use only" and should never be called by clients should be declared static.

• Near the top of file.c will be the following line which ensures that the function definitions in file.c see the prototypes in file.h which ensures the "prototype before definition" rule above.

#include "file.h" // show the contents of "file.h" to the compiler at //this point

• Any xxx.c file which wishes to call a function defined in file.c must include the following line to see the prototypes, ensuring the "clients must see prototypes" rule above.

#include "file.h"