Introduction to Computers and Programming

1.1 Computer Hardware Organisation

The traditional components of a digital computer are shown in figure 1. The organisation of functional parts is generally referred to as the ***architecture*** of the computer. The basic computer system consists of five units: the ***input unit***, the ***control and arithmetic unit*** (contained in the CPU or the central processing unit), the ***memory unit*** and the ***output unit***. Most of the computer systems available these days have ***secondary storage devices***.

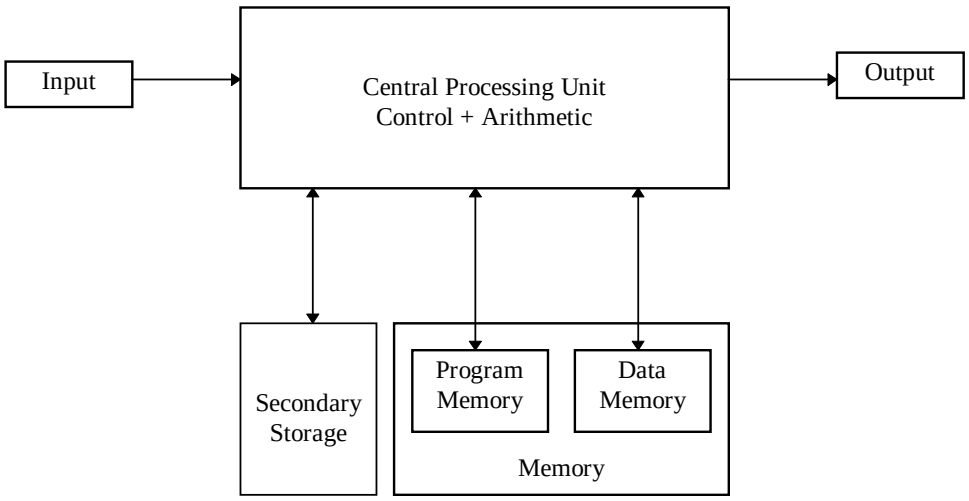


Figure - 1 : General Organisation of a Computer

These physical units are referred to as computer ***hardware***. The ***program memory*** of the computer contains the ***program*** that is to be executed. A ***program*** is a list of instructions for a computer. The process of preparing a program (or a list of instructions for a computer) is known as ***programming***. Programs manipulate information or ***data*** stored in the ***data memory***. Data can be differentiated as ***numerical data*** or ***character oriented data*** (e.g., names, addresses etc.). Scientific and technical applications primarily require the processing of numerical data whereas business applications usually involve the processing of both numerical and character data. Some computer programs are used only to process textual type character data (e.g., letters, books, manuscripts etc.).These programs are known as ***word processors***. ***Software*** is a general term given to cover all programs. If software is stored permanently in a computer's program memory, it is known as ***firmware***.

Briefly, a computer works in the following manner. Both programs and data are fed into the CPU and transferred to their respective memory locations. The CPU reads the first instruction from the program memory and executes it. Instructions may be as simple as to **ADD** two numbers, **MOVE** data from one memory location to another, **READ** data from an input device, **WRITE** data to an output device or **JUMP** to a different place in the program. After the data manipulations are complete, results are transferred to the output of the computer. Again most actions of the CPU are caused by the instructions stored in program memory.

Semiconductor memories can be divided into two main groups : ***read write memories*** and ***read only memories*** (***ROM***). The names imply the difference between the two types of memories. A ROM is a nonvolatile memory which has its contents permanently programmed by the manufacturer of the computer or its components. Read/write memory can be easily programmed, erased, and reprogrammed by the user. Programming a memory is known as ***writing*** into memory. Copying data from a memory device without destroying the contents in the memory is known as ***reading*** from memory. Read/write memory is most often called the ***RAM*** (***Random Access Memory***). Generally, information stored in RAM is volatile, i.e., it will be lost if the power to the memory device is turned off even for an instant. Thus RAM is used for temporary storage of user programs and data. ROM, on the contrary, is generally used to store programs that perform initialisation functions at start up, control input and output devices and perform tasks related to the management of other system hardware devices.

A computer's RAM is volatile. If the power to the computer is switched off, the RAM loses the data stored in it earlier. In order to store data and information for periods longer than a single operational session of a computer, secondary storage devices are used. Most common technique used in the secondary storage devices is magnetic recording. Information stored on these devices can, generally, be erased, rewritten and altered but is available even after the power to the computer is switched off. If these devices are removable, they can be detached and moved to another computer. There the information can be read and manipulated by the second computer. Most common magnetic storage devices used in computers are the magnetic tapes and magnetic disks.

The most fundamental unit of data manipulated by a computer is known as a ***bit***. The word ***bit*** is a contraction for ***binary digit***. A bit (or binary digit) can only have one of the two values 0

or 1 at any given time. The bits 0 and 1 may be said to represent ***off*** and ***on*** or ***true*** and ***false***. When voltage is present at a given location, that location is interpreted as holding the value 1; when no voltage is present, that location is interpreted as holding the value 0. Because it is practical to make electronic devices that work with on/off signals at great speeds, it is possible to make machines that work with information based on the binary number system.

Thus bits serve as the building blocks that enable the construction of larger and more meaningful information. By themselves, bits usually are not of much interest or use. It is generally a string of bits and the patterns generated by them that make more useful information. The most important and interesting collection of bits is the ***byte*****.** A byte is eight bits taken together in a single unit. This means that there are eight individual 0 or 1 settings in each byte. This allows a byte to have a total of 256 distinct values ($2^8 = 256$).

A byte inside a computer is raw data that can be used for anything. Two of the most important tasks that are generally accomplished with the help of computers are numerical arithmetic and text processing. Bytes are the building blocks of both numbers and text (character) data. Bytes basically represent numerical information. If a text processing program is being executed on the computer, it will interpret the bytes as text characters. For example, the integer value 65 is represented as ***A*** in a text processing application. Thus computers always work with numbers, the interpretations of these numbers are task-specific.

Computers are usually referred to as 8-bit, 16-bit or 32-bit computers. These represent the amount of data that the processor of the computer can manipulate in a single operation. Computers belonging to the PC family are 16-bit or 32-bit computers. Earlier home computers were 8-bit computers as their processors could only manipulate a byte at one time. Several systems used for industrial control applications are 8-bit computers as their processors can only manipulate one byte (or 8 bits) in one machine level operation.

Kilobyte (***KB***), ***megabyte*** (***MB***) and ***gigabyte*** (***GB***) are the terms most commonly used to refer to the capacity of a computer's memory or the secondary storage devices. The term kilobyte refers to 2^{10} or 1024 bytes of data. The ***kilo*** in kilobyte comes from the metric term for thousand. In a computer context, however, kilobyte represents 1024 (not 1000) bytes. Therefore, 64KB means 64×1024 or 65,536 bytes. Similarly, a megabyte is roughly a million bytes. To be exact,

a megabyte represents 2^{20}, or exactly 1,048,576 bytes. Finally, gigabyte refers to 2^{30} or 1,073,741,824 bytes.

1.2 Operating Systems

An operating system is a program that controls the execution of application programs and acts as an interface between the user of the computer and the computer hardware. An operating system can be thought of as having three objectives.

1. ***Convenience***. An operating system makes a computer more convenient to use.
2. ***Efficiency***. An operating system allows the computer system resources to be used in an efficient manner. A computer can be considered as a set of resources used for the movement, storage and processing of data and for the control of these functions. These resources include the computer's memory, its input and output devices, secondary storage media etc. The operating system is also responsible for managing these resources.
3. ***Ability to Evolve***. An operating system should be constructed in such a way as to permit the effective development, testing and introduction of new systems functions without interfering with service.

The hardware and software that are used in providing applications to a user can be viewed in a layered or hierarchical fashion as depicted in figure-2.

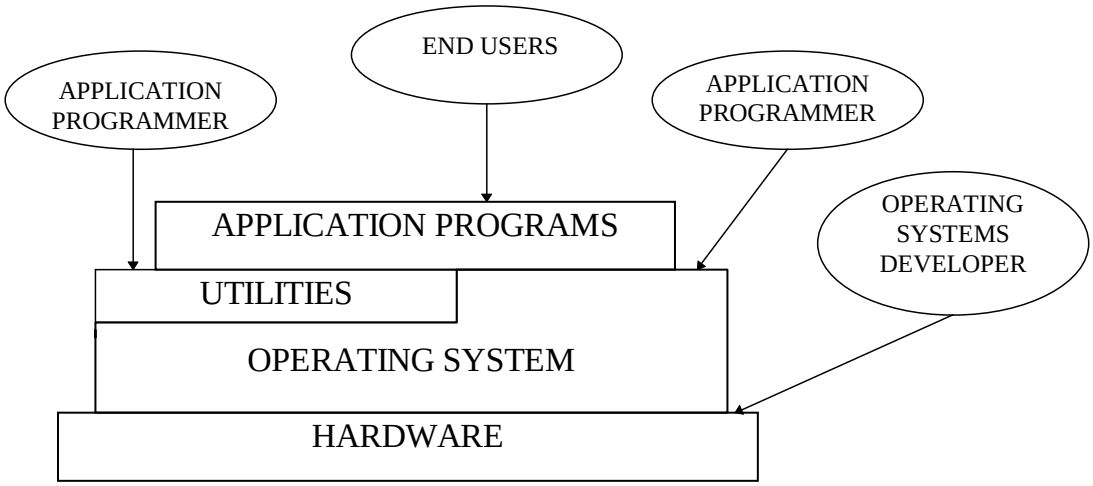


Figure-2 : Layers and Views of a Computer System

A user of these applications is called the end user and generally is not concerned with the computer's architecture. Thus, the end user views a computer system in terms of an application. That application can be expressed in a programming language and is developed by an application programmer. If one were to develop an application program as a set of machine level instructions that is completely responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex task. To ease this task, a set of system programs is provided. Some of these programs are referred to as utilities and they implement the frequently used functions that assist in program creation, the management of files and the control of input/output (IO) devices. A programmer makes use of these facilities in developing an application. An end-user application, while it is in execution invokes these utilities to perform certain functions. The most important system program is the operating system. The operating system masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as a mediator, making it easier for the programmer and the application programs to access and use these facilities and services.

For a computer to be useful, it needs to communicate with its users. This is a two way transfer of information. The users must inform the system which specific services are required and the system provides the user with the results of operations performed to provide these services. The part of the operating system for managing this interaction is the ***user interface***.

The user interface of an operating system can be divided into two parts;

1. ***Command Interface***. This consists of an ***operating system command language*** - a language by which the user conveys instructions to the operating system, and an ***operating system response language*** - a language by which the operating system returns information to the user. The operating system response language presents at least four different kinds of messages to the users,
    a. ***prompt messages*** - let the user know that the operating system is waiting for instructions or information.
    b. ***help messages*** - provide the user with information to help decide what to do next.
    c. ***progress messages*** - tell the user that a long activity is still in progress.
    d. ***termination messages*** - tell the user that the current activity has finished, this also includes error messages.

The command interface, thus, interprets commands and conveys instructions to the rest of the operating system to carry out those commands. Moreover, it accepts messages for the users from the operating system and presents them to the user.

Command interfaces are usually of the following three types,

a. ***Command Language***. Commands provide a way of expressing instructions to the computer directly. The users type the command for the operation they want to execute. The operating system performs the requested operation (either itself or by loading and executing a separate program) and provides the results of the operation to the user. Consider figure - 3. This shows the screen of a personal computer using the MS-DOS (Microsoft - Disk Operating System). The message is particularly cryptic. The '>' symbol indicates that the system is waiting to receive a command to tell it what to do next, for example, DIR command issued by the user to list the contents of a directory. The alphabets C: inform the user that currently drive C is being used. It is important for a command language designer to give appropriate names to the commands. This helps the users to remember what the commands refer to.

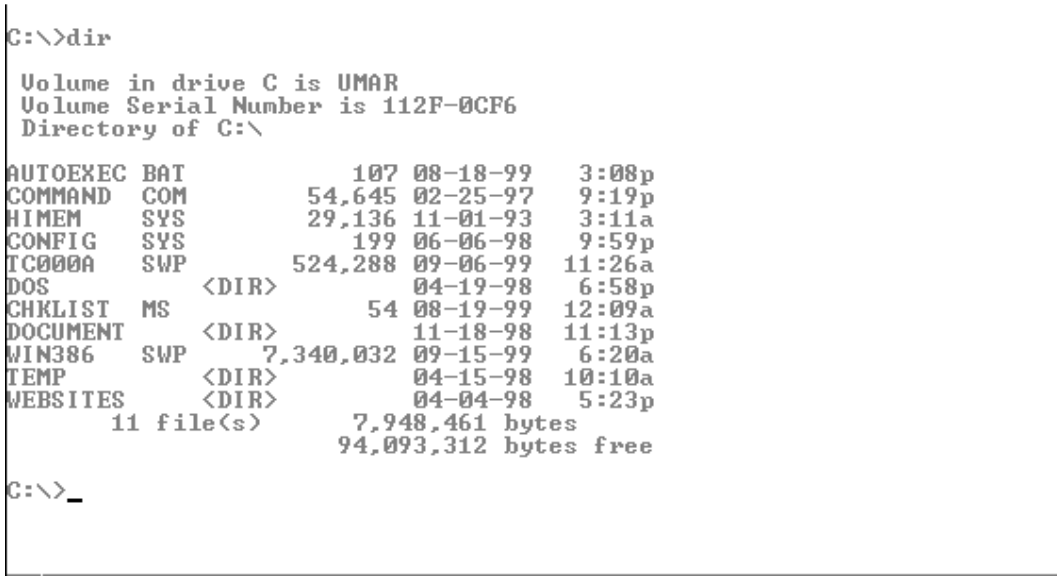


Figure-3 : DOS Command Interface

b. ***Menus***. A menu is a set of options displayed on the screen where the selection and execution of one (or more) of the options results in specific command being issued to the system. Unlike the command language interfaces, menus have the advantage that the

users do not have to remember the exact syntax of the command they want to issue to the system. This means that for menus to be effective, names selected for menu items have to be self explanatory. A menu based user interface is shown in figure-4.

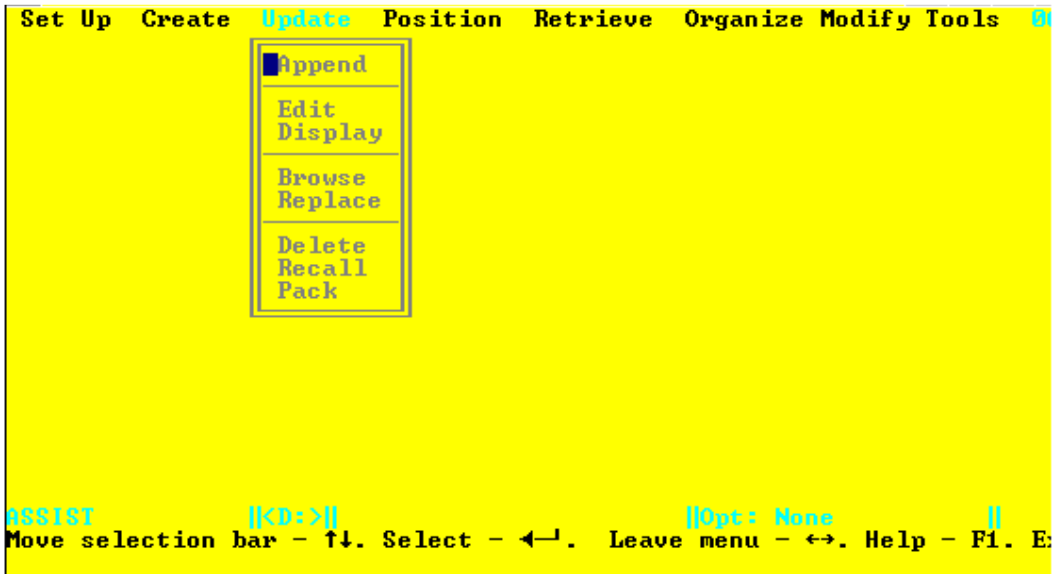


Figure - 4 : A Menu Based User Interface

b. ***Graphical User Interface***. Current Graphical User Interfaces (GUIs) employ a style of interaction known as ***direct manipulation***. The main feature of such systems is that the complex command language syntax is replaced by direct manipulation of objects of interest. These objects are made visible to the users on the computer display. For instance, a document file may be displayed as a book. To open this file, the user may just need to point at it with the mouse pointer and click. The system executes the word processor which load that specific document file and displays it to the user for editing. These systems have gained immense popularity amongst a wide range of users, i.e., from novices to experts. Novices do not have to remember complex command syntax and sequences where as the experienced users can perform the desired operation quickly and efficiently. In most of the cases, the experienced users can even modify the user interface to meet their own specific requirements. The graphical user interface of Windows 3.11 is shown in figure-5.

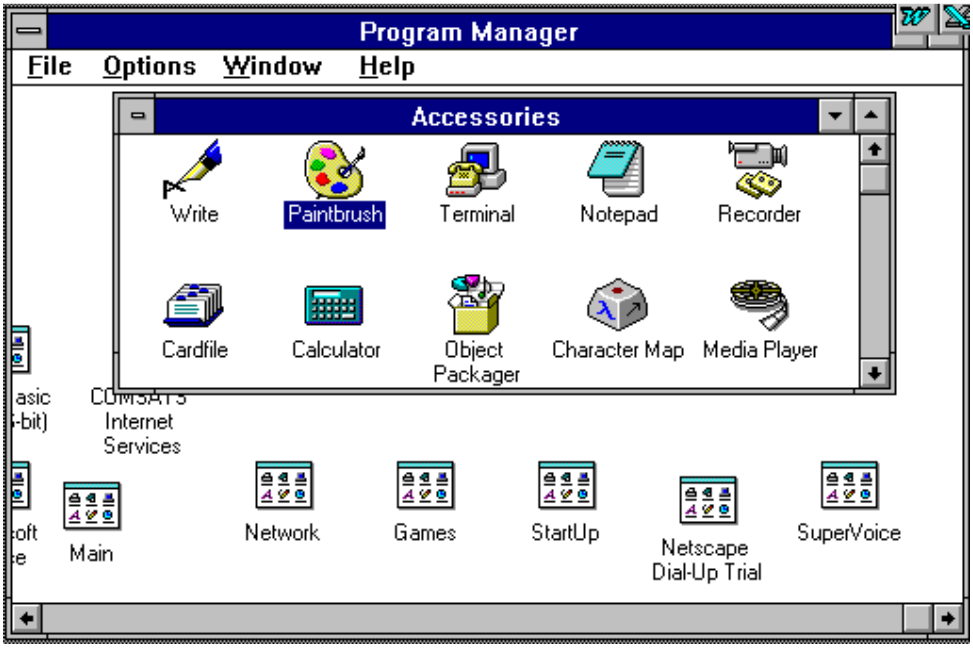


Figure - 5 : Graphical User Interface

2. ***Program Interface***. The program interface manages the programs in execution and provides interaction between these programs and system controlled resources and services. The program interface loads and sets up the programs that are to be executed. Additionally, the program interface must offer proper response to normal and abnormal termination of programs.

Some of the most famous operating systems include UNIX, Microsoft DOS (Disk Operating System), Microsoft Windows 95/98, Microsoft Windows NT, IBM OS/2.

1.3 Basic Programming Constructs

Variables. Variables may be the most fundamental aspect of any programming language. A variable is a space in the computer's memory which is set aside (or ***reserved***) for a certain kind (or ***type***) of data and is given a name for easy reference. Specifying the type of the data (i.e., integer numbers, floating point numbers, characters, strings etc.) that a variable is used for is important as it specifies to the computer how values are to be stored, interpreted and manipulated inside the computer.

Variables allow the same space in memory to hold different values at different times during the execution of the program. For instance, consider a program written to calculate pay cheque for

an employee of a company. It will have variables to store the number of hours the employee has worked in the month and the hourly rate that applies. The pay cheque for the employee is calculated by multiplying the hourly rate with the total hours worked and the result is stored in a third variable. If the same program is used to calculate the pay cheques of all the employees in the firm, the values for each of these variables will vary for each employee. Thus a variable is a space in memory that plays the same role many times, but may contains different values each time.

Operators, Expressions and Statements. Each programming language uses special symbols called operators to indicate at least addition, subtraction, division, multiplication and assignment operations. These operator symbols are shown Table - 1.

Operators	Meaning
+	Addition Operation
-	Subtraction Operation
*	Multiplication Operation
/	Division Operation
=	Assignment Operation

Table - 1 : Elementary Operator Symbols

These operators are used to connect numbers and numerical variables, thus forming ***formulas*** (or ***expression***). The indicated operations are carried out on the numeric terms in a formula, resulting in a single numerical value. Hence a *formula represents a single numerical quantity.* For example, consider three variables to hold integer numbers named, `NumberA`, `NumberB` and `NumberC`. These variables hold the values 2, 5 and 3 respectively. The expression `NumberA + NumberB - NumberC` will represent the quantity 4.

Strictly speaking, a formula or an expression can be composed of a single number or a single numeric variable as well as some combination of numbers, numeric variables and operators. It is important to understand, however, that a *numeric variable used in an expression must be assigned some numerical quantity before it can appear in a formula*.

Statements are the primary building blocks of a program. A program is just a series of statements. A statement is a complete instruction to the computer. For example,

```
NumberA + NumberB - NumberC
```

is not a complete instruction. It tells the computer to add the values of the three variables but it fails to tell the computer what to do with the answer. Alternatively, following is a complete statement

```
Answer = NumberA + NumberB - NumberC
```

Here, `Answer` is another integer variable. This is a complete instruction as it tells the computer to store the result of the numerical operation in the memory location labeled `Answer`.

Following is a program in a simple demonstration language. The text in italics are remarks explaining the statements

```
INTEGER NumberA          A variable of the type INTEGER declared
INTEGER NumberB          A variable of the type INTEGER declared
INTEGER NumberC          A variable of the type INTEGER declared
INTEGER Answer           A variable of the type INTEGER declared
NumberA = 3              3 is stored in location labeled NumberA
NumberB = 5              5 is stored in location labeled NumberB
NumberC = 1              1 is stored in location labeled NumberC
Answer = NumberA + NumberB - NumberC
            The numerical expression NumberA + NumberB - NumberC
            is evaluated and the result is stored in the location
            labeled Answer
PRINT Answer             The value contained in the location
                         Answer is printed
END                      Signals the end of the program to the
                         computer
```

Statements in this program are executed in a sequence, from the first statement to the last. All the statements in the program collectively provide the functionality of the program.

Jumps. As mentioned earlier, a complete program in any language is composed of a sequence of statements. These statements are executed in the order in which they appear unless a deliberate ***jump*** (or a transfer of control) is indicated.

The simplest of jump statements are the ***unconditional jumps***. These occur in a program, usually as statements with the indication of the statement in the program to which the control must be transferred. Consider the following program,

```
      REAL HourlyRate          A variable for floating point numbers
      REAL HoursWorked         A variable for floating point numbers
      REAL Pay                 A variable for floating point numbers
A:    INPUT HourlyRate         The program accepts data from the key-
                               board and stores it in the location
                               labeled HourlyRate. A: is the label of
                               the statement
      INPUT HoursWorked        The program accepts data from the key-
                               board and stores it in the location
                               labeled HoursWorked
      Pay = HoursWorked * HourlyRate
                               The expression HoursWorked * HourlyRate is
                               evaluated and the result is stored in the
                               location labeled Pay.
      PRINT Pay                The contents of the location labeled Pay
                               are printed on the display
      JUMP A                   Transfer the control to statement labeled
                               A:
      END
```

Here, once the contents of the variable have been printed by the statement `PRINT Pay`, the statement `JUMP A` is executed. This statement transfers the flow of control to the statement labeled `A:` which is

```
A:    INPUT HourlyRate
```

Due to the unconditional jump from near the end of the program to a statement near the beginning, the program loops continuously between these two points.

A more useful kind of transfer of control is the one that is associated with a certain condition, i.e. a ***conditional jump***. In such statements, the flow of control is transferred only when a specified condition holds true. For instance, consider the following program,

```
      REAL HourlyRate          A variable for floating point numbers
      REAL HoursWorked         A variable for floating point numbers
      REAL Pay                 A variable for floating point numbers
A:    INPUT HourlyRate         The program accepts data from the key-
                               board and stores it in the location
                               labeled HourlyRate. A: is the label of
                               the statement
      JUMP B IF HourlyRate < 0.0
                               Transfer control to the statement labeled
                               B if the value of the variable HourlyRate
```

```
                              is negative
        INPUT HoursWorked     The program accepts data from the key-
                              board and stores it in the location
                              labeled HoursWorked
        Pay = HoursWorked * HourlyRate
                              The expression HoursWorked * HourlyRate is
                              evaluated and the result is stored in the
                              location labeled Pay.
        PRINT Pay             The contents of the location labeled Pay
                              are printed on the display
        JUMP A                Transfer the control to statement labeled
                              A:
B:      END
```

In this program, after the user has entered the value for the variable `HourlyRate`, i.e., the statement

```
A:      INPUT HourlyRate
```

has executed, the statement

```
        JUMP B IF HourlyRate < 0.0
```

is executed. This statement checks the value held by the variable `HourlyRate`. If this value is a negative number, the computer is instructed to jump to the statement labeled `B`. The statement labeled `B` is the end of the program.

Alternatively, if the value of the variable `HourlyRate` is greater than 0.0, the condition `HourlyRate < 0.0` is false and the jump is not executed. Rather the computer executes the next instruction, which is

```
        INPUT HoursWorked
```

and the subsequent instructions are executed sequentially till time the computer encounters another jump statement. This program also demonstrates another interesting concept. The transfer of control in a program does not only occur in the forward but also in the reverse direction to the flow of the program.

Loops and Iterations. The programming elements that provide the facility to iterate, or perform the same operation several times are known as loops. These can be divided into the following three types.

a. ***Pre-entry Condition Loops***. In a pre-entry condition loop, a condition in the form of a logical expression is placed at the beginning of the loop body. This condition is tested each time the body of the loop has to be executed and if found to be true, the body of the loop is executed. If the condition is false, the body of the loop is not executed and the control of the program is passed to the statement immediately after the body of the loop.

Consider the following program

```
CHARACTER Choice          A variable to store a character value
                          is declared and is assigned a label
                          Choice
INPUT Choice              User is prompted to enter a character
                          which is stored in Choice
WHILE Choice NOT = 'E'    Start of the pre-entry condition loop
      PRINT "IN LOOP"     Display message IN LOOP
      PRINT Choice        Display value of Choice
      INPUT Choice        Enter a character value to be stored
                          in Choice
END LOOP                  End of loop body
PRINT "THE END"           Display message THE END
END
```

The statement `INPUT Choice` prompts the user to enter a character. The `WHILE Choice NOT = 'E'` is the start of the pre-entry condition loop. The expression `Choice NOT = 'E'` defines the condition that should be fulfilled for the body of the loop to be executed. If the value of this expression is true, the program enters the body of the loop, otherwise the control is passed to the statement immediately after the body of the loop, i.e., `PRINT "THE END"` statement. The body of the loop is bound between the `WHILE Choice NOT = 'E'` and `END LOOP` statements.

When the program starts execution the user is prompted to enter a character. If the user enters 'A', the result of the expression `Choice NOT = 'E'` is true and the body of the loop is executed. At the end of the loop the user is prompted again to enter a character. Lets assume that the user now enters 'B'. At the statement `END LOOP` the control of the program is transferred to the beginning of the loop. The result of the expression `Choice NOT = 'E'`

is again true, and the body of the loop is executed. Now, if at the end of the loop, the user enters 'E', the result of the expression

`Choice NOT = 'E'` is false and the body of the loop is not executed. The control is transferred to the `PRINT "THE END"` statement. Thus the variable `Choice` can be referred to as the ***loop control variable*** as the decision to execute the body of the loop depends upon the value of this variable.

An execution cycle of the program may appear as follows

```
? A
IN LOOP
A
? B
IN LOOP
B
? E
THE END
```

However, if the user enters 'E' immediately at the first `INPUT Choice` statement (i.e., before the beginning of the loop), the output of the program is as follows

```
? E
THE END
```

This demonstrates an important characteristic of the pre-entry condition loops, i.e., depending upon the value of the loop control variable, the body of the loop may never be executed in a particular program execution cycle.

b. ***Post-entry Condition Loops***. In post entry condition loops, the condition that determines whether the body of a loop should be executed or not is at the end of the loop. There is no condition at the start of the loop to determines whether a loop should be executed or not. So the post-entry condition loops are always executed once at least.

Consider the following program

```
CHARACTER Choice          A variable to store a character
                          value is declared and is assigned a
                          label Choice
BEGIN LOOP                Start of the body of the loop
      PRINT "IN LOOP"     Display the message IN LOOP
      INPUT Choice        Enter a character from the keyboard
                          assign its value to Choice
      PRINT Choice        Display the value of Choice
WHILE Choice NOT = 'E'    If the value of Choice is not
```

```
                              equal to E, the loop is executed
                              again
PRINT "THE END"               Display the message THE END
END                           End of the program
```

In this program, the statement `BEGIN LOOP` identifies the start of the body of the loop. The statement `WHILE Choice NOT = 'E'` is the end of the body of the loop. Note that the logical expression `Choice NOT = 'E'` is at the end of the body of the loop. When the program executes, a variable of the type `CHARACTER` is declared and is assigned the label `Choice`. The body of the loop starts from the second statement. As there is no condition at the beginning of the loop, the body of the loop is executed. The statement `PRINT "IN LOOP"` displays the message `IN LOOP` to the user. The `INPUT Choice` statement prompts the user to enter a character from the keyboard. Lets assume that the value of the character entered by the user is 'A'. When the statement `WHILE Choice NOT = 'E'` is executed, the control of the program is passed to the start of the loop as the expression `Choice NOT = 'E'` is true. The body of the loop will then be executed again.

Alternatively, if the user had entered 'E', the value of the expression
`Choice NOT = 'E'` would have been false and the body of the loop would not be executed the second time. The control of the program is transferred to the statement `PRINT "THE END"` which displays the `THE END` message to the user. The execution of the program is then terminated.

Consider the following execution cycle of the program

```
IN LOOP
? A
IN LOOP
? B
IN LOOP
? E
THE END
```

Compare this with the following execution cycle

```
IN LOOP
? E
THE END
```

This example demonstrates the concept that a post-entry condition loop is executed at least once.

c. ***For Loops***. For loops are a special kind of pre-entry condition loops. These loops are used when the body of a loop needs to be executed for a pre-determined number of times. A generic for loop has a single statement at the beginning of the body of the loop. This statement has three parts. The first part is referred to as the ***initialisation part***. This part is used to initialise the loop control variable. This part of loop is executed only once, i.e., before the first iteration of the loop. The second part is known as the ***condition part***. This contains the logical expression which determines whether the next iteration of the loop should be executed or not. This expression is evaluated before the body of the loop has to be executed. If the result of this expression is true, the body of the loop is executed. If this expression is false, body of the loop is not executed and the control is transferred to the first statement after the loop. The third part of the for loop statement is known as the ***update part***. Here the loop control variable is updated using a mathematical expression.. The update part of the for loop statement is always executed at the end of body of the loop, i.e., after the last statement in the body of the loop has been executed.

Consider the following program

```
INTEGER Anumber            A variable of type INTEGER is declared
                           and is assigned the label Anumber
FOR Anumber = 0; Anumber < 5 ; Anumber = Anumber + 1
   PRINT “IN LOOP”         Displays the message IN LOOP
   PRINT Anumber           Displays the value of Anumber
END LOOP                   End of the loop body
PRINT “THE END”            Displays the message THE END
END                        End of the program
```

In the statement `FOR Anumber = 0; Anumber < 5 ; Anumber = Anumber + 1`, the expression `Anumber = 0` is the initialisation part. The loop control variable is initialised in this expression. This part is executed only once before the beginning of the loop. The expression `Anumber < 5` is the condition part. It is executed in the beginning of each iteration. If the value of the loop control variable is less than 5, the value of this expression is true and the body of the loop is executed. The expression `Anumber = Anumber + 1` is

the update part and is executed after the last statement in the body of the loop has been executed.

When the program is executed, `INTEGER Anumber` declaration declares a variable of the type `INTEGER` and labels it as `Anumber`. Next, the statement

```
FOR Anumber = 0; Anumber < 5 ; Anumber = Anumber + 1
```

is executed. In the beginning, the expression `Anumber = 0` is executed. This expression is executed only once for the loop and initialises the value of `Anumber` to 0. The body of the loop has to be executed after the expression `Anumber < 5` has been evaluated. As the value of the loop control variable is 0, this expression is true. The body of the loop is, therefore, executed. The statements `PRINT "IN LOOP"` and

`PRINT Anumber` are executed respectively. At the end of the body of the loop the update part, i.e., `Anumber = Anumber + 1` is executed, which increments the value of the loop control variable to 1. The condition is then checked again. As the value of the loop control variable is still less than 5, the body of the loop is executed again. This process is repeated until the value of the loop control variable is either equal to or greater than 5. At that point, the control of the program is transferred to the statement immediately after the loop, i.e., `PRINT "THE END"`. Output from an execution cycle of such a program is given below.

```
IN LOOP
0
IN LOOP
1
IN LOOP
2
IN LOOP
3
IN LOOP
4
THE END
```

As the condition part of the loop is executed at the beginning of each iteration, for loops are a kind of pre-entry condition loops.

Subroutines and Program Modules. A ***subroutine*** may be defined as a small independent program that has been designed and implemented to perform a specific task. A program that uses various subroutines to achieve its desired objectives is termed as a ***modular program***. A modular program consists of a ***main module***. This main module calls other subroutines appropriately to achieve the overall objective of the system (Figure - 6). When a subroutine is

called, the control is transferred from the main module to the invoked sub-routine. This subroutine performs its task and returns control to the main module upon completion of the task. If the task assigned to a subroutine is too complex, it must be broken down into other sub-subroutines (Figure - 7). The subroutine then acts as a main module for the sub-subroutines and invokes them appropriately to perform the desired tasks.

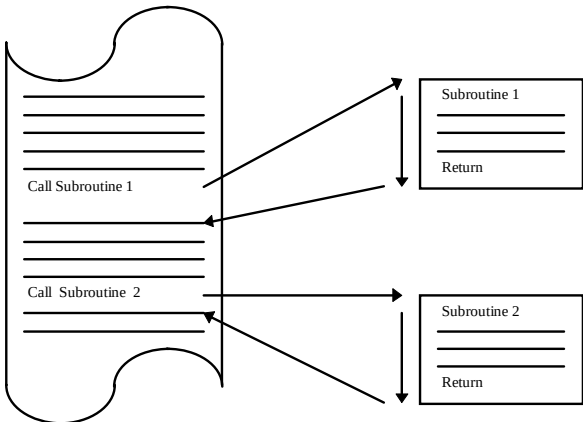


Figure - 6 : Main Module and Subroutine Interaction in a Modular Program

Significant advantages can be achieved by developing modular software. Several subroutines, for instance mathematical functions, used in a program may be of use in other programs. In such cases, these functions are implemented as libraries of code. A program that requires functions from a specific library need only call the required subroutine from the appropriate library. Thus modular programs enhance ***software reusability*** (Figure - 8). Moreover a function required at different places within the program need only be called at those points.

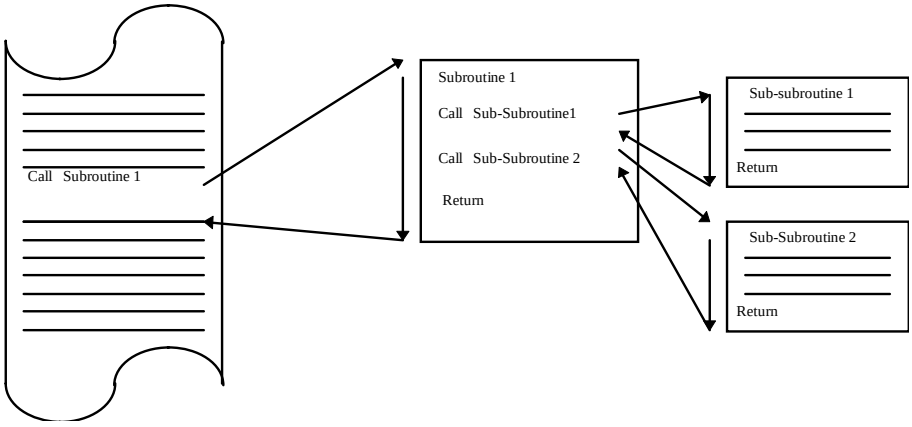


Figure - 7 : Decomposition of a Subroutine

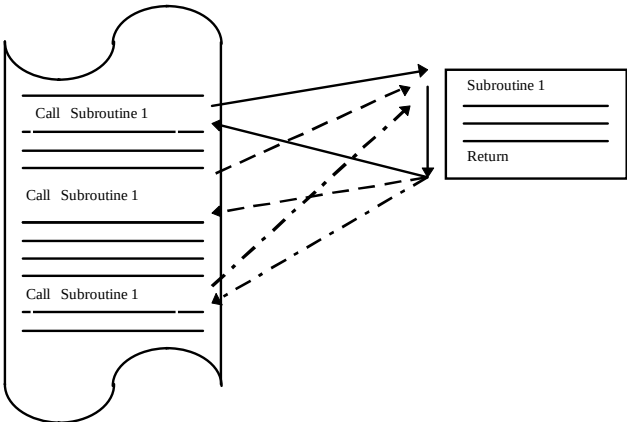


Figure -8 : Software Reusability

1.4 Flow Charts and Program Organisation

A flowchart is a pictorial representation of a specific activity or procedure. Symbols in a flowchart identify inputs to the activity, its outputs, decision points and individual processes within the activity (Table - 2) . These symbols are connected via directed lines which indicate the sequence in which each of the operations in the activity is performed.

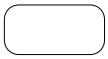
Symbol	Meaning
	Terminal
	Input / Output
	Connector
	Process
	Decision
	Function

Table - 2 : Flowchart Symbol

Flowcharts have traditionally been used to represent data processing activities, computer programs and algorithms. Flowcharts are easier to comprehend, develop and translate into a computer program. Moreover flowcharts are independent of any particular programming

language. Thus a given flowchart can be used to translate an algorithm into more than one programming language.

Lets use flow charts to provide a graphical representation to some simple problems.

Example 1. Input two numbers and print their sum and difference (Figure - 9)

Example 2. Input two numbers and print the greater number. (Figure - 10)

It may be noticed that table 2 does not contain any symbol for any of the loops mentioned in the previous section. As all loops consists of some primitive operations (i.e., initialisation, comparison and update), appropriate symbols for loops can be constructed from symbols of these basic operations.

Exercise 3. Using a for loop, implement a counter from 0 to 14 (Figure-11).

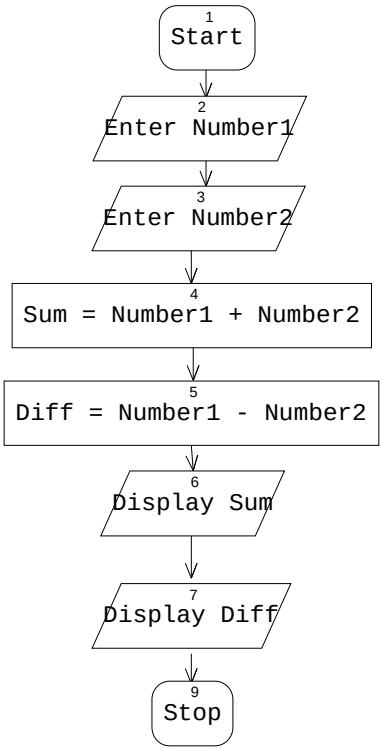


Figure - 9 : Exercise - 1

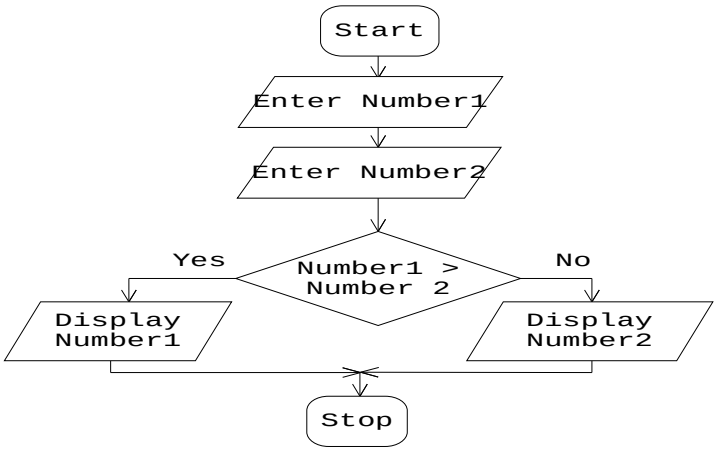


Figure - 10 : Exercise - 2

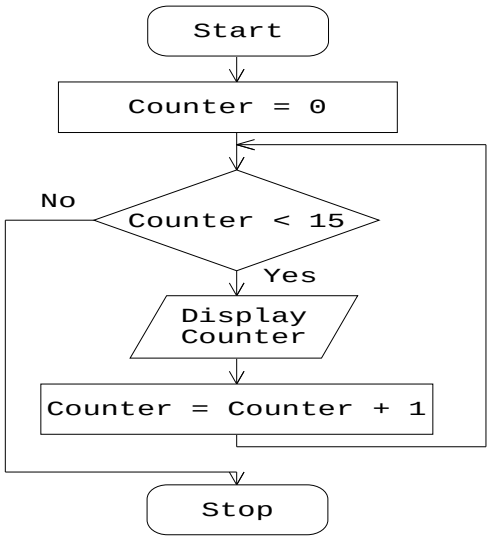


Figure - 11 : Exercise - 3

Exercise 4. Show the flow chart for a program that allows you to enter characters till you enter ‘e’. Use a pre-entry condition loop (Figure - 12).

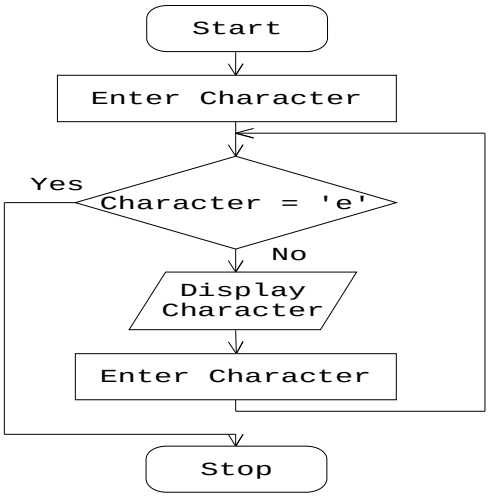


Figure - 12 : Exercise - 4

Exercise 5. Show the flow chart for a program that allows you to enter characters till you enter 'e'. Use a post-entry condition loop (Figure - 13).

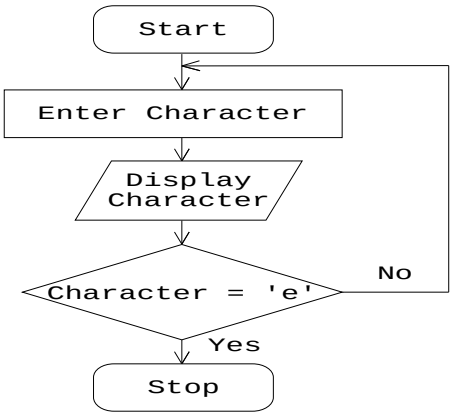


Figure - 13 : Exercise - 5

Flowcharts can be used to design modular software. In this case the design of a complete system may consist of multiple flowcharts, each displaying a specific activity within the system at a particular level of detail. The flowchart of the main module shows how different sub-modules are invoked. A flowchart for each sub-module shows the activities within that sub-module. A detailed description of this concept will be presented later with functions and structured programming.

1.5 Programming Languages

As mentioned earlier, a program is a logically self-contained collection of computer instructions. Programmers may write these instructions in many different languages, but the

computer hardware understands only one form, i.e., ***machine language***. A machine language program consists of a series of binary numbers that can be directly interpreted by the computer's processor. To program directly in machine language, programmers must know the binary codes for each of valid instructions that can be given to a particular computer. Furthermore, a machine language also requires the programmers to have a detailed working knowledge of the computer's processor. Machine language forces the programmers to determine the addresses of memory locations to be used as variables, jump addresses, various status codes of the processor, etc.

Thus a machine language program written for one specific type of computers cannot be executed over another type of computer without significant alterations. Moreover, because of the complexity involved, machine language programming tends to be extremely time consuming, tedious and error prone.

Assembly languages ease some of the problems associated with machine language programming. These languages allow the programmer to use symbolic instruction codes (called ***mnemonics***) instead of binary instructions. For instance, to copy data from one memory location to another, the binary instruction of a particular processor may be $E8_{hex}$ whereas the assembly language instruction can be MOV.

The ***assembler*** is a program that translates the symbolic code used in an assembly language program into machine language and also relieves the programmer of many housekeeping chores. For example, since symbolic language is a set of instructions with a one to one correspondence with the machine language instructions, it is a simple task for the assembler to assign address for certain known memory locations, etc.

To be able to program in assembly language, the programmer must also have a sound working knowledge of a particular computer's hardware. Because assembly language is very similar to machine language, it is frequently referred to as a ***low level language***.

Programming languages called ***high level languages*** reduce the programmer's need to understand the intimate details of the computer's architecture. The instruction set of most of the high level languages is more compatible with the human languages and the human thought processes. Most of these high level languages are general purpose languages (e.g., BASIC,

Pascal, C, C++, etc.). There are also various high level special purpose languages whose instruction sets are specifically designed for some particular type of application.

As a rule, a single instruction in a high level language will be equivalent to several instructions in machine language. Moreover, a program that is written in a high level language can generally be run on many different computers with little or no modification. Therefore, the use of a high level language offers some significant advantages over the use of machine language, namely, ***simplicity***, ***uniformity***, and ***portability*** (i.e., machine independence).

A program that is written in a high level language must, however, be translated into machine language before it can be executed. This can be achieved by either ***compilation*** or ***interpretation***. A compiler is a computer program that accepts a high level program (e.g., a program written in C or in BASIC) as input data and produces a corresponding machine language program as output. Accordingly, the original high level program is called the ***source code*** or the ***source program*** and the resulting machine language program is called the ***object code*** or the ***object program***.

Although the object code version of the program is in machine language, it cannot be executed directly for several reasons. For instance, the program may call subroutines from libraries that have previously been programmed and compiled. Moreover, the program is only a part of a complete system and requires to be called from another module to execute. A program called the ***linker*** links all the necessary object files together to produce a final executable program. The relationship of the compilation process to the linking process is shown in figure - 14.

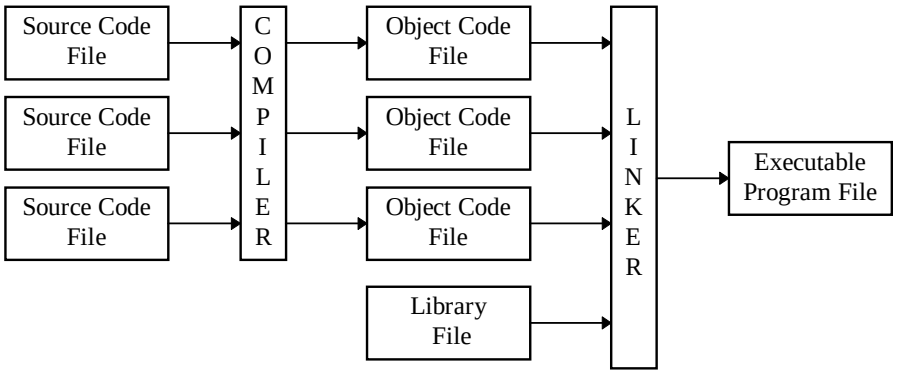


Figure - 14 : Relationship Between Linker and Compiler

On the contrary, interpreters do not produce object code. Rather, they look at and execute programs on a line by line basis. Each line of the source code is scanned, parsed and executed before moving on to the next line. Thus, an interpreter does not have distinct translation and execution phases as in compiled languages. The two phases are actually interleaved continually. That is, as soon as one phrase of the source language has been translated to the corresponding executable code, it is executed without waiting for the translation of the complete source code file.

Introduction to C

2.1 Advantages of Programming in C

C is a small language with fewer keywords (or reserved words) than most other programming languages, yet is arguably a more powerful language. C gets its power by carefully including the right control structures and data types and allowing their uses to be nearly unrestricted where meaningfully used. This language can be readily learned as a consequence of its functional minimality.

C is the native language of UNIX which is a major interactive operating system on workstations, servers and mainframes. Also C is the standard development language for PCs. Much of MS-DOS and OS2 is written in C. Many windowing packages, database programs, graphics libraries and other large applications are written in C.

C is portable. Code written on one machine can be moved to another machine. C provides the programmers with a standard library of functions that work the same on all machines. Also C has a built in preprocessor that helps the programmers isolate any system-dependent code.

C is terse. It has a powerful set of operators and some of these operators allow the programmers to access hardware at bit level. For many programmers, this is both elegant and efficient. Software productivity studies show that programmers produce, on average, only a small amount of working code each day. A language that is terse explicitly magnifies the underlying productivity of the programmer.

C is modular. C allows developing programs as sets of interacting modules or ***functions***. This not only increases the reusability of the code but also reduces the overall complexity significantly.

C is the basis for C++. Many of the constructs and methodologies that are routinely used by C programmers are also used by C++ programmers. Thus learning C can be considered a first step in learning C++.

C is efficient on most machines. Because certain constructs in the language are machine dependent, C can be implemented in a manner that is natural with respect to a specific machine's architecture. This makes compiled C code extremely efficient. Of course the programmers must be aware of any code that is machine dependent.

2.2 ANSI C

The acronym ANSI stands for ***American National Standards Institute***. ANSI Committee X3J11 is responsible for setting standards for the C programming language. In the late 1980's the committee created draft standards for ANSI C or standard C. By 1990, the committee had finished its work and the ISO (International Standards Organisation) approved the standard for ANSI C as well. Thus ANSI C or ANSI/ISO C is an internationally recognised standard.

The standard specifies the form of programs written in C and establishes how these programs are to be interpreted. The purpose of the standard is to provide portability, reliability, maintainability and efficient execution of C language programs on a variety of computers. Almost all C compilers now follow the ANSI C standard.

2.3 A Simple Program in C

Consider the following program.

```
#include <stdio.h>

int main(void)
{
      printf("Welcome to C Programming Language\n");
      return(0);
}
```

The only function of this program is to print the message

```
Welcome to C Programming Language
```

on the display. Following is a line by line analysis of the program.

`#include <stdio.h>`

A pre-processor is built into the C compiler. When the command is given to compile a program, the code is first pre-processed and then compiled. Lines that begin with a `#` communicate with

the pre-processor. The **`#include`** line causes the pre-processor to include a copy of the header file **`stdio.h`** at this point in the code. This header file is provided by the C system. This file is included here as it contains information about the **`printf()`** function.

`int main(void)`

This is the first line of the function definition for **`main().int`** and **`void`** are keywords. They have special meaning to the compiler. Each program has a function **`main()`**. Program execution always starts with this function. The words inside the parenthesis specify the data that has to be sent into the function and the keyword preceding **`main()`** specifies the type of data returned once the function has finished execution. **`void`** specifies that the function should not be sent any data and **`int`** specifies that it returns an integer value. The word **`int`** stands for integer. The word integer itself cannot be used.

`{}` (braces) surround the body of a function definition. They are also used to group statements together. Each **`{`** has to be matched with a **`}`**.

`printf("Welcome to C Programming Language\n");`

This is a call to the **`printf()`** function. In a program, the name of the function followed by parenthesis causes the function to be called or invoked. If required, the parenthesis must contain arguments. The semi-colon in the end completes the statement. Many statements in C end with a semi-colon. The semi-colon tells the compiler where one statement ends and the next begins.

`printf()` function is from the library used to provide input to the program from the keyboard and output from the program to the display. **`stdio.h`** provides essential information to the compiler for these functions and, therefore, is included in the program.

`"Welcome to C Programming Language\n"` is a string constant. In C, a string constant is defined by a series of characters surrounded by double quotes. This string is an argument to the **`printf()`** function and controls what gets printed on the display. The two characters **`\n`** at the end of the string represent a single character called ***newline***. It is a non-printing character. It advances the cursor on the screen to the beginning of the next line.

Here, when the **`printf()`** function is called, it prints its arguments, i.e., a string constant on the screen.

`return(0);`

This is a return statement. It causes the value **`0`** to be returned to the operating system. The operating system may, in turn, use the value in some way.

2.4 A Sample Program with Variables

Consider the following program

```
#include <stdio.h>

int main(void)
{
      int ANumber;
      int Result;
      ANumber = 25;
      Result = ANumber * 4;
      printf("ANumber = %d\n",ANumber);
      printf("Result  = %d\n",Result);
      return(0);
}
```

The statements `int ANumber; int Result;` are declarations. A declaration specifies two aspects of a variable, its name and its type. In the case of these statements, `ANumber` and `Result` are the names of the variables and `int` specifies their type, i.e., integer. Declarations and statements end with a semi-colon. In C all variables must be declared. This means that the programmer has to provide a list of the all the variables that are to be used in the program and must also specify their types. Programmers must choose meaningful names for their variables. The number of characters that a programmer can use will vary among implementations and in most modern compilers is a maximum of 32 characters.

The statement `ANumber = 25;` is an assignment statement. It is one of the most basic operations. This particular statement means *give the variable ANumber the value of 25*.

The statement `Result = ANumber * 4;` is also an assignment statement. The value of the expression on the right side of the assignment operator is assigned to the variable `Result`. The expression `ANumber * 4` is evaluated by multiplying the value of variable `ANumber` by `4`.

The statement `printf("ANumber = %d\n",ANumber);` calls or invokes the `printf()` function. This function can be passed a variable number of arguments. The first argument is always a string and is called the ***control string***. The control string in this statement is

"ANumber = %d\n". %d is known as the format specifier. A format specifier tells the printf() function where to put a value in the string and what format to use in printing the value. In this particular case, the %d tells the printf() function to print the value of ANumber as a decimal integer.

The effect of the following two statements,

```
printf("ANumber = %d\n",ANumber);
printf("Result  = %d\n",Result);
```

can be achieved from a single statement as shown below

```
printf("ANumber = %d\nResult  = %d\n",ANumber,Result);
```

In this statement, the first %d corresponds to the variable ANumber and the second %d corresponds to the variable Result. Thus there is a one to one correspondence between the format specifiers in the control string and the variables in the order these variables are listed after the control string.

2.5 Entering Data in Programs

In the previous program, the variable ANumber was assigned a value in the program. Once this program has been compiled and linked, the value assigned to the variable cannot be changed and the executable version of the program uses that value whenever it is executed. This makes the program extremely inflexible. Moreover, the program's source code requires an update whenever the user wishes to use a new value for ANumber. This situation can be remedied by making provisions in the program that would allow the user to enter values that can be assigned to the variables at run time.

Consider the following program. This is an interactive version of the previous program

```
#include <stdio.h>

int main(void)
{
      int ANumber;
      int Result;
      printf("Enter an integer value : ");
      scanf("%d",&ANumber);
```

```
      Result = ANumber * 4;
      printf("ANumber = %d\nResult  = %d\n",ANumber,Result);
      return(0);
}
```

The `printf("Enter an integer value : ");` prints the following on the display

```
      Enter an integer value :
```

prompting the user to type in an integer number.

The statement `scanf("%d",&ANumber);` invokes the `scanf()` function. This function is analogous to the `printf()` function but is used for input rather than output. Its first argument is a control string having formats that correspond to the various ways the data entered from the keyboard are to be interpreted. The other arguments to the `scanf()` function are ***addresses***.

In the statement `scanf("%d",&ANumber);` the format specifier `%d` is matched with the expression `&ANumber` causing `scanf()` to interpret data arriving from the keyboard as a decimal integer and to store the result at the address of `ANumber`. The symbol `&` is known as the ***address operator*** and is used with the name of the variable to determine its address. Thus the expression `&ANumber` is evaluated to the address of the memory location storing the value of the variable `ANumber`.

Additionally, `scanf()` function can accept input to several variables at once. For instance, consider the following program.

```
#include <stdio.h>

int main(void)
{
   int ANumber;
   int BNumber;
   int Result;
   printf("Enter two integer number to be multiplied seperated by a space :
");
   scanf("%d %d",&ANumber, &BNumber);
   Result = ANumber * BNumber;
   printf("%d * %d = %d\n",ANumber,BNumber,Result);
   return(0);
}
```

This program prompts the user to enter two decimal integer numbers seperated by a space, and prints out the product of these two numbers. The output of this program is as follows,

```
Enter two integer number to be multiplied seperated by a space : 15 5
15 * 5 = 75
```

As the user types the two input values, 15 and 5, the space that separates these values at the input is matched by the `scanf()` function with the corresponding space between the format specifiers in the control string `"%d %d"`. This space serves as a delimiter. Actually, with a space in the control string separating the format specifier, any whitespace character can be used to delimit the input (i.e., new line, tab, etc.). For example, consider the following execution cycle of the same program,

```
Enter two integer number to be multiplied seperated by a space : 12
6
12 * 6 = 72
```

Any character can be used as a delimiter if it has been specified in the control string of the `scanf()` function. For instance, following is a version of the program that uses a colon as a delimiter.

```
#include <stdio.h>

int main(void)
{
      int ANumber;
      int BNumber;
      int Result;
      printf("Enter two integer number to be multiplied seperated by a space
: ");
      scanf("%d:%d",&ANumber, &BNumber);
      Result = ANumber * BNumber;
      printf("%d * %d = %d\n",ANumber,BNumber,Result);
      return(0);
}
```

Notice the colon in the control string for the function call

```
scanf("%d:%d",&ANumber, &BNumber);
```

Consider the following execution cycle of this program

```
Enter two integer number to be multiplied seperated by a space : 12:3
12 * 3 = 36
```

Please note that in the program shown above, no other character except colon can act as a delimiter for the input.

2.6 Tokens

Tokens represent the basic vocabulary of the C language. In ANSI C, there are 6 kinds of tokens, namely, ***keywords***, ***identifiers***, ***constants***, ***string constants***, ***operators*** and ***punctuation.*** The compiler checks that the tokens can be formed into legal strings according to the syntax of the language. ***Syntax*** is defined as the rules for combining the alphanumeric characters and other symbols to make legal or correct programs. If there is an error, the compiler will print an error message and stop. If there are no errors, then the source code is legal and the compiler translates it into object code, which in turn gets used by the linker to produce an executable file.

2.6.1 Keywords

Keywords are explicitly reserved words that have a strict meaning as individual tokens in C. These cannot be redefined or used in other contexts. `int`, `void` and `return` are some of the keywords in C.

C has less than 40 keywords. This is a relatively small number compared to other major programming languages. Ada, for example, has 62 keywords. It is a characteristic of C that it does a lot with relatively few special symbols and keywords.

2.6.2 Identifiers

An identifier is a token that is composed of a sequence of letters, digits and the special character _ known as the underscore. A letter or underscore must be the first character of an identifier. In most implementations of C, uppercase and lowercase letters are treated as distinct. It is a good programming practice to choose identifiers that have mnemonic significance so that they contribute to the readability and documentation of the program.

Some examples of identifiers are

```
b
_oneidentifier
anotheridentifier
yet_another_identifier
```

Following are not identifiers

```
not#me      /* special character # is not allowed */
2nd_number  /* must not start with a numeric digit */
-my_name    /* do not mistake - for _ */
```

Identifiers are created to give unique names to various objects in a program. Keywords can be thought of as identifiers that are reserved to have special meaning in the C language. Identifiers such as `printf` and `scanf` are already known to the C system as input/output functions in the standard library. These names would not normally be redefined. The identifier `main` is special, in that C programs always begin execution at the function called `main`.

In ANSI C, at least the first 31 characters of an identifier are discriminated, i.e., the first 31 characters of an identifier must be a unique combination. Good programming style requires the programmer to choose names that are meaningful. If one were to write a program to figure out price of fuel at a petrol station, identifiers such as `fuel_dispensed`, `price_per_litre` and `total_price` may be used so that the statement

```
total_price = fuel_dispensed * price_per_litre;
```

would have an obvious meaning. The underscore is used to create a single identifier from what would normally be a string of words seperated by spaces. Meaningfulness and avoiding confusion go hand in hand with readability to constitute the main guidelines for a good programming style.

Programmers should exercise caution while using identifier that begin with an underscore. Such identifiers can conflict with system names. Only systems programmers should use such identifiers. For example the identifier `_iob` is often defined as the name of an array of structures in `stdio.h`. If a programmer tries to use `_iob` for some other purpose, the compiler may complain or the program may misbehave. Application programmers are best advised to use identifiers that do not begin with an underscore.

2.6.3 Constants

C manipulates various kinds of values. Numbers such as `0` and `17` are examples of integer constants and numbers such as `1.0` and `3.14159` are examples of floating constants. Like most languages, C treats integers and floating constants differently. There are also character constants in C, such as `'a'`, `'b'` and `'*'`. Character constants are written between single quotes and are closely related to integers. Some character constants are special, such as the newline character, written as `'\n'`. The backslash is the escape character. `'\n'` can be, therefore, be referred to as *escaping the usual meaning of n*. Even though `\n` is written as two characters `\` and `n`, it represents a single character.

All constants are collected by the compiler as tokens. Because of implementation limits, constants that are syntactically expressible may not be available on a particular machine. For instance, an integer may be too large to be stored in a machine word.

Decimal integers are finite strings of decimal digits. Because C provides octal and hexadecimal integers as well as decimal integers, one has to be careful to distinguish between the different kinds of integers. For example, `17` is a decimal integer constant, `017` is an octal integer constant and `0x17` is a hexadecimal integer constant. Moreover, negative integers such as `-33` are considered as constant expressions.

Following are some example of decimal integers,

```
0
27
564332233   /*    too large for the machine     */
```

Following are not decimal integers

```
0432        /*    an octal integer  */
-87         /*    a constant expression   */
325.0       /*    a floating constant     */
```

2.6.4 String Constants

A sequence of characters enclosed in a pair of double quote marks, such as `"abc"`, is a string constant or a string literal. It is collected by the compiler as a single token. String constants are

stored by the compiler as an array of characters and hence are treated differently from character constants. For example, `"a"` and `'a'` are not the same.

Note that the double quote mark `"` is just one character, not two. If the character `"` itself is to occur in a string constant, it must be preceded by a backslash character `\`. If the character `\` is to occur in a string constant, it too must be preceded by a backslash

Some examples of string constants are given below,

```
"a string of text"
""                      /* the null string */
"     "                 /* a string of blanks */
" a = b + c; "          /* not a statement but a string constant */
" /* This is not a comment */ "
" a string with double quotes \" within "
" a single backslash \\ is in this string "
```

but following are not string constants

```
/* "this is not a string " */
" and
 neither is this "
```

Character sequences that would have meaning if outside a string constant are just a sequence of characters when surrounded by double quotes. In the previous examples, one string contains what appears to be the statement `a = b + c;` but since it occurs surrounded by double quotes, it is explicitly this sequence of characters.

2.6.5 Operators and Punctuation

In C, there are special characters with particular meanings. Examples include the arithmetic operators,

```
+     -     *     /     %
```

which stands for the usual arithmetic operations of addition, subtraction, multiplication, division and modulus respectively. Recall that in mathematics, the value of *a modulus b* is obtained by taking the remainder after dividing *a* by *b*. Thus, for example, `5%3` has the value `2`, and `7%2` has the value `1`.

Some symbols have meanings that depend on the context . As an example of this, consider the `%` symbol in the two statements `printf("%d", a);` and `a = b%7;`

The first `%` symbol is the start of a conversion specifier or format, whereas the second `%` symbol represents the modulus operator.

Examples of punctuators include parenthesis, braces, commas and semicolons. Consider the following code,

```
int main(void)
{
      int a,b,c;
      a = 17 * (b+c);
      ...
}
```

The parentheses immediately following `main` are treated as an operator. They tell the compiler that `main` is the name of a function. After this, the symbols `{ , ; ( )` are punctuators. Both operators and punctuators are collected by the compiler as tokens and, along with white spaces, they serve to separate language elements.

Some special characters are used in many different contexts, and the context itself can determine which use is intended. For example, parenthesis are sometimes used to indicate a function name; at other times they are used as punctuators.

2.6.5.1 Operator Precedence and Associativity

Operators have rules of precedence and associativity that determine precisely how expressions are evaluated. `*`, `/` and `%` have a higher precedence than `+` and `-`. This means that the former are evaluated before the latter. Expressions within parenthesis are evaluated first. Parenthesis can, therefore, be used to clarify or change the order in which operators are performed.

Consider the expression

```
1 + 2 * 3
```

As the operator `*` has a higher precedence than `+`, multiplication is performed before addition. Hence the value of the expression is `7`. An equivalent expression is

```
1 + (2 * 3)
```

On the other hand, because expressions inside parenthesis are evaluated first, the expression

```
(1 + 2) * 3
```

is different; its value is `9`.

Table 2-1 summerises the rules for these operators. Notice that the two uses of the minus sign have different priorities. The associativity column indicates how an operator associates with its operands. For example, the unary minus sign associates with the quantity to its right, and in division, the left operand is divided by the right.

Operators	Associativity
`( )`	left to right
`-` (unary)	right to left
`* / %`	left to right
`+ -` (subtraction)	left to right
`=`	right to left

Table 2-1 : Operators in Decreasing Order of Precedence

Consider the expression `1 + 2 - 3 + 4`. Because the binary operators `+` and `-` have the same precedence, the associativity rule *left to right* is used to determine how it is evaluated. The *left to right* rule means that the operations are performed from left to right. Thus `(((1 + 2) -3) + 4) -5` is an equivalent expression.

Similarly, consider the expression, `-a * b - c`. The first minus sign is unary and the second is binary. Using the rules of precedence, this expression can be transformed into `((-a) * b) - c`.

2.6.5.2 Increment and Decrement Operators

The increment `++` and decrement `--` operators are unary operators with the same precedence as the unary minus. Both `++` and `--` can be applied to variables, but not to constants or ordinary expressions. Moreover, the operators can occur in either prefix and postfix position and different effects may occur. Some examples are

```
++i
cnt--
```

but not

```
753++       /* constants cannot be incremented  */
++(a - b)   /* ordinary expressions cannot be incremented   */
```

Each of the expressions `++i` and `i++` has a value. Moreover, each causes the stored value of `i` in memory to be incremented by `1`. The expression `++i` causes the stored value of `i` to be incremented first, with the expression then taking as its value the new stored value of `i`. In contrast, the expression `i++` has as its value the current value of `i`, then the expression causes the stored value of `i` to be incremented. The following program illustrates this phenomenon.

```
#include <stdio.h>

int main(void)
{
      int i = 0;
      int j,k;
      printf("Original value of i = %d\n\n",i);
      j = i++;
      printf ("j = i++ performed\nj = %d :: i = %d\n",j,i);
      k = ++i;
      printf("k = ++i performed\nk = %d :: i = %d\n",k,i);
      return(0);
}
```

The output of this program is given below

```
Original value of i = 0

j = i++ performed
j = 0 :: i = 1
k = ++i performed
k = 2 :: i = 2
```

In the similar fashion, `--i` causes the stored value of `i` in memory to be decremented by `1` first, with the expression then taking this new stored value as its value. With `i--` the value of the expression is the current value of `i`; then the expression causes the stored value of `i` in memory to be decremented by `1`.

Note carefully that `++` and `--` cause the value of a variable in memory to be changed. Other operators do not do this. For example, an expression such as `a + b` leaves the values of the variables `a` and `b` unchanged. These ideas are expressed by saying that the operators `++` and `--` have a *side effect*, not only do these operators yield a value, they also change the stored value of a variable in memory.

In some cases `++` or `--` can be used in either prefix or postfix position, with both uses producing equivalent results. For example, each of the two statements `i++;` and `++i;` is equivalent to `i = i + 1;`.

In simple situations, these operators can be considered to provide concise notation for the incrementing and decrementing of a variable. In other situations, careful attention must paid as to whether prefix or postfix position is desired.

2.6.5.3 Assignment Operators

Assignment operator (=) are used to change the value of a variable. Its precedence is lower than all other operators and its associativity is right to left. To understand = as an operator, consider + for the sake of comparison. The binary operator + takes two operands, as in the expression `a + b`. The value of the expression is the sum of the values of `a` and `b`. By comparison, a simple assignment expression is of the form

```
variable = right_side
```

where `right_side` is itself an expression. Notice that a semicolon placed at the end would have made this an assignment statement. The assignment operator = has the two operands `variable` and `right_side`. The value of `right_side` is assigned to `variable`, and that value becomes the value of the assignment expression as a whole.

In addition to =, there are other assignment operators, such as += and -=. An expression such as `k = k + 2` will add `2` to the old value of `k` and assign the result to `k`, and the expression as a whole will have that value. The expression `k += 2` accomplishes the same task. The following list contains the most commonly used assignment operators.

```
=  +=  -=  *=  /=  %=
```

Consider the following demonstration program

```
#include <stdio.h>

int main(void)
{
      int Original, Surrogate;
      Original = 15;
      Surrogate = Original;
      Original += 7;
      printf("Original = %d :: Original += 7 = %d\n",Surrogate,
              Original);
      Original = Surrogate;
      Original -= 7;
      printf("Original = %d :: Original -= 7 = %d\n",Surrogate,
              Original);
      Original = Surrogate;
      Original *= 7;
      printf("Original = %d :: Original *= 7 = %d\n",Surrogate,
              Original);
      Original = Surrogate;
      Original /= 7;
      printf("Original = %d :: Original /= 7 = %d\n",Surrogate,
              Original);
      Original = Surrogate;
      Original %= 7;
      printf("Original = %d :: Original MOD= 7 = %d\n",Surrogate,
              Original);
      return(0);
}
```

The output of this program is as follows

```
Original = 15 :: Original += 7 = 22
Original = 15 :: Original -= 7 = 8
Original = 15 :: Original *= 7 = 105
Original = 15 :: Original /= 7 = 2
Original = 15 :: Original MOD= 7 = 1
```

2.7 Comments

Comments are arbitrary strings of symbols placed between delimiters `/*` and `*/`. Comments are not tokens. The compiler changes each comment into a single blank character. Thus comments are not part of the executable program. Following are some examples of comments

```
/* This is a comment */       /*** another comment ***/      /*******/
```

```
/*
      A comment can be written in such a fashion
      to set it off from the surrounding code
*/
```

Comments are used by programmers as a documentation aid. The aim of documentation is to explain clearly how the program works and how it is to be used. Sometimes a comment contains an informal argument demonstrating the correctness of the program.

Comments should be written simultaneously with program text. Although some programmers insert comments as a last step, there are two problems with this approach. The first is that once the final implementation version of the program has been developed, the tendency is to either abbreviate or omit the comments. The second is that ideally the comments should serve as running commentary, indicating program structure and contributing to program clarity and correctness. They cannot serve this purpose if they are inserted after coding has finished.

Fundamental Data Types

3.1 Variable Declarations

Programs work with data. Users or computer operators feed numbers, letters and words to the computers and computers use programs to manipulate these data to provide the required results. Data in a computer program may be specified as constants or may be assigned to some variables. These constants and variables are used in formulae and expressions that form the basis of data manipulation operations in the computer programs.

In C, all variables must be declared before they can be used. Consider the following program,

```
#include <stdio.h>

int main(void)
{
      int   ANumber,BNumber,CNumber;              /* declaration */
      float AReal,BReal,CReal;                    /* declaration */
      printf("Enter two integer numbers : ");
      scanf("%d %d",&ANumber,&BNumber);
      printf("Enter two real numbers : ");
      scanf("%f %f",&AReal,&BReal);
      CNumber = ANumber + BNumber;
      CReal = AReal + BReal;
      printf("CNumber = %d\n",CNumber);
      printf("CReal   = %f\n",CReal);
      return(0);
}
```

Declarations serve two purposes. First, they tell the compiler to set aside an appropriate amount of space in the memory to hold values associated with variables, and second, they enable the compiler to instruct the machine to perform specified operations correctly. In the expression `ANumber + Bnumber`, the operator `+` is being applied to two variables of type `int`, which at the machine level is a different operation than `+` applied to the variables of type `float`, as occurs in the expression `AReal + BReal`. Of course, the programmer need not be concerned that the two `+` operations are mechanically different, but the C compiler has to recognise the difference and give the appropriate machine instructions.

3.2 Overview of the Fundamental Data Types

C provides several fundamental types. These are listed below,

`char`	`signed char`	`unsigned char`
`signed short int`	`signed int`	`signed long int`
`unsigned short int`	`unsigned int`	`unsigned long int`
`float`	`double`	`long double`

Table 3-1 : Fundamental Data Types - Long Form

These are all keywords. These should not be used as names of variables. Of course, `char` stands for ***character*** and `int` stands for ***integer***, but only `char` and `int` can be used as keywords. Other data types are derived from the fundamental types.

Usually, the keyword `signed` is not used. For example, `signed int` is equivalent to `int`, and because shorter names are easier to type, `int` is typically used. Also, the keywords `short int`, `long int` and `unsigned int` may be, and usually are shortened to just `short`, `long` and `unsigned` respectively. The keyword `signed` by itself is equivalent to `int`, but is seldom used in this context. With all these convention, the list shown above may be modified as shown below,

`char`	`signed char`	`unsigned char`
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

Table 3-2 : Fundamental Data Types - Short Form

The syntax of a declaration is shown below

```
type identifier;
```

Here type represents any one of the fundamental data types given in the preceding table. Syntax for declaring several variables of the same data type is given below,

```
type identifier1,identifier2,...identifiern;
```

The fundamental types can be grouped according to functionality. The integral types are those types that can be used to hold integer values whereas the floating point types are those that can be used to hold real values. They are all arithmetic types.

Integral types:	char	signed char	unsigned char
	short	int	long
	unsigned short	unsigned	unsigned long
Floating types:	float	double	long double
Arithmetic types:	Integral types + Floating types		

Table 3-3 : Functional Grouping of Fundamental Data Types

3.3 Characters and the Data Type `char`

In C, variables of any integral type can be used to represent characters. In particular, both `char` and `int` variables are used for this purpose. This is because a computer can only store a pattern of 1s and 0s. To handle characters, the computer uses a numerical code in which certain integers represent certain characters. The most common code is the ASCII (American Standard Code for Information Interchange). The standard ASCII code runs numerically from 0 to 127. This is small enough that 7 bits can hold the largest code value. The `char` type typically is defined as a 1 byte (or 8 bits) unit of memory, so it is more than large enough to encompass the standard ASCII code. Many systems, such as the IBM PC and the Apple Macintosh, offer extended ASCII codes (not the same for both systems) that still stay within an 8-bit limit. The following table illustrates the correspondence between some character and integer values in the ASCII table.

Observe that there is no particular relationship between the value of the character constant representing a digit and the digit's intrinsic integer value. That is, the value of `'2'` is not `2`. The property that the value for `'a'`, `'b'`, `'c'`, and so on occur in order is important. It makes convenient for sorting of characters, words and lines into lexicographical order. In addition to representing characters, a variable of type `char` can be used to hold small integer values.

Character Constants	:	'a'	'b'	'c'	...	'z'
Corresponding Values	:	97	98	99	...	122
Character Constants	:	'A'	'B'	'C'	...	'Z'
Corresponding Values	:	65	66	67	...	90
Character Constants	:	'0'	'1'	'2'	...	'9'
Corresponding Values	:	48	49	50	...	57
Character Constants	:	'&'	'*'	'+'		
Corresponding Values	:	38	42	43		

Table 3-4 : Some Character Constants and Their Integer Codes

Variables of type `char` are declared in the same manner as other variables. Following are some examples,

```
char Acharacter;
char Operator, UserChoice;
```

This program would create three variables of the type `char`, `Acharacter`, `Operator` and `UserChoice`. Some C implementations make `char` a signed type; this means it can hold values in the range -128 through +127. Other implementations make `char` an unsigned type. This provides a range from 0 through 255. Many newer implementations allow the programmers to use the keywords `signed` and `unsigned` with `char`. Then regardless of what char is, `signed char` would be signed and `unsigned char` would be unsigned.

There are two ways to initialise character variables. In the first case, the numerical ASCII code can be used as shown below,

```
char grade = 65;
```

In this example, `65` is type `int`, but, since the value is smaller than the maximum size of variables of type `char`, it can be assigned to the variable `grade` without any problems. ASCII code `65` represents alphabet `A`.

Alternatively, the programmer can assign the character value `A` to the variable `grade` with the following initialisation statement

```
char grade = 'A';
```

As mentioned earlier, a single letter contained between single quotes is a C character constant. When the compiler sees `'A'`, it converts it to the proper code value. This initialisation can be divided into the following two steps,

```
char grade;
grade = 'A';
```

If the programmer omits the single quotes, the compiler thinks that the letter after the assignment operator is the name of a variable.

```
char grade;
grade = A;  /* A is taken as variable */
```

`printf` and `scanf` functions use `%c` format specifier to indicate that a character should be printed or read from the keyboard respectively. Recall that a character is stored as an integer, therefore, `%c` informs `printf` function to convert the integer value to its corresponding character. Consider the following program,

```
#include <stdio.h>

int main(void)
{
      char ACharacter = 76;
      char BCharacter = 't';
      printf("ACharacter %c :: %d\n",ACharacter,ACharacter);
      printf("BCharacter %c :: %d\n",BCharacter,BCharacter);
      return(0);
}
```

The statement `char ACharacter = 76;` declares a variable `ACharacter` of the type `char` and initialises it with an integer value `76`. The statement `char BCharacter = 't';` declares a variable `BCharacter` of the type `char` and initialises it with a character constant `'t'`.

The statement `printf("ACharacter %c :: %d\n",ACharacter,ACharacter);` prints the value of the variable `ACharacter`, first as a character and then as its integer ASCII code. Similarly the statement

`printf("BCharacter %c :: %d\n",BCharacter,BCharacter);` prints the value of `BCharacter` as a character and then as an ASCII integer value. The output of this program is as follows,

```
ACharacter L :: 76
BCharacter t :: 116
```

Consider another program shown below,

```
#include <stdio.h>

int main(void)
```

```
{
      char Character1;
      printf("Enter a character : ");
      scanf("%c",&Character1);
      printf("%c has an integer value %d\n",Character1,Character1);
      return(0);
}
```

The statement `char Character1;` declares a variable `Character1` of the type `char`. The statement `printf("Enter a character : ");` prompts the user to enter a character by printing the following message on the display,

```
Enter a character :
```

The statement `scanf("%c",&Character1);` waits for the user to enter a character from the keyboard. Once the user has typed the character and pressed the ENTER key, the character entered is stored in the variable `Character1`. The statement `printf("%c has an integer value %d\n",Character1,Character1);` prints the value of `Character1` twice, first as a character (as specified by the `%c` format specifier) and then as a decimal integer (as specified by the `%d` format specifier).

Some non-printing and hard to print characters require an escape sequence. The horizontal tab character, for example, is written as `\t` in character constants and in strings. Even though it is being described by the two characters `\` and `t`, it represents a single character. The backslash character is called the ***escape character*** and is used to escape the usual meaning of the character that follows it. The following table contains some non-printing and hard to print characters.

Name of Character	**Written in C**	**Integer Value**
Alert	`\a`	7
Backslash	`\\`	92
Backspace	`\b`	8
Carriage Return	`\r`	13
Double quote	`\”`	34
Formfeed	`\f`	12
Horizontal tab	`\t`	9
Newline	`\n`	10

Null character	\0	0
Single quote	\'	39
Vertical tab	\v	11

Table 3-5 : Non Printing and Hard to Print Characters

In addition to `scanf` and `printf` functions, the standard C library provide few other functions that can be used to input characters from the keyboard and print them on the display. Programmers can use the `getchar` function to read one character from the keyboard into the executing program. The `putchar` function gets one character from the executing program and prints it on the display. Definitions for `getchar` and `putchar` are in `stdio.h` file.

Consider the following example,

```
#include <stdio.h>
int main(void)
{
      char ch;
      printf("Enter a character and then press <ENTER>\n");
      ch = getchar();
      putchar(ch);
      return(0);
}
```

The output of this program is as follows,

```
Enter a character and then press <ENTER>
r[ENTER]
r
```

`[ENTER]`represents the user's action of pressing the enter key. Initially the system prints the message `Enter a character and then press <ENTER>`. The user then presses the character key `r` and then presses the enter key. The program prints the character entered by the user on the display.

The `getchar` function takes no arguments. It simply fetches the next character from the keyboard and makes it available to the program. It can be said that the function returns a value. The statement `ch = getchar();` assigns the return value to `ch`. Thus, if the programmer types

a letter `r`, the `getchar` function reads the character and makes it into a return value. And the assignment statement then assigns the return value to `ch`.

The `putchar` function, on the other hand, takes an argument. The programmer must place between the parenthesis whatever single character that needs to be printed. The argument can be a single character or a variable or a function whose return value is a single character. The following example demonstrates the use of `putchar`,

```
#include <stdio.h>

int main(void)
{
      char ch;
      ch = 'u';
      putchar(ch);      /* Prints the value of ch, i.e., 'u' */
      putchar('\n');    /* Prints a newline character */
      putchar('W');     /* Prints 'W' */
      putchar(10);      /* Prints a newline character, 10 is the ASCII
                           value of '\n' */
      putchar(99);      /* Prints 'c', 99 is the ASCII value for 'c' */
      return(0);
}
```

The output of this program is as follows,

```
u
W
c
```

`getchar` provides ***buffered input***. In buffered input, the characters a user types are collected and stored in an area of temporary storage called a ***buffer***. Pressing the enter key then causes the block of characters to be made available to the program. Alternatively, in ***unbuffered input*** (or ***direct input***) the character a user types is made available to the waiting program immediately without waiting for the enter key. Consider another session of the previous program using `getchar`,

```
Enter a character and then press <ENTER>
Hello World[ENTER]
H
```

Here, the `getchar` function is called after the `printf` function has displayed the line

`Enter a character and then press <ENTER>`. The user can enter more than one character before pressing the enter key. All these characters are stored in the buffer. Once the user presses enter, these characters are made available to the program. As this program only has one `getchar` function call, the first character is read and its value assigned to the variable `ch`.

Consider a modified version of the previous program,

```
#include <stdio.h>

int main(void)
{
      char ch1;
      char ch2;
      char ch3;
      char ch4;
      char ch5;
      printf("Enter characters now ");
      ch1 = getchar();
      putchar(ch1);
      ch2 = getchar();
      putchar(ch2);
      ch3 = getchar();
      putchar(ch3);
      ch4 = getchar();
      putchar(ch4);
      ch5 = getchar();
      putchar(ch5);

      return(0);
}
```

Now consider the following execution session,

```
Enter characters now Hello [ENTER]
Hello
```

In this case the user types in the whole word `Hello` and then presses the enter key. The state of the buffer is shown in figure 3-1.

Figure 3-1 : Buffer

The first `getchar` function call returns the character `H`, the second returns the character `e`, the third returns the character `l`, and so on. As there are only five `getchar` function calls, the buffer is only read until (and including) the character `o`. Consider another program execution session,

```
Enter characters now H[ENTER]
H
e[ENTER]
e
l[ENTER]
l
```

Here, the user attempts to type in the data, one character at a time and presses the enter key after each character. The state of the buffer is shown in figure 3-2

Figure 3-2 : Buffer

The first `getchar` function call returns the character `H`, the second `getchar` call returns the newline character, the third `getchar` function call returns the character `e`, the fourth `getchar` function call returns the newline character and the fifth `getchar` function call returns the character `l`.

C also provides functions for direct input. One such function is the `getche` function. `get` indicates that a value needs to be entered, `ch` indicates that the value is of the type character and `e` specifies that the value should be echoed on the output device. This means that whenever a program is awaiting input as a result of the `getche` function invocation, and the user presses a key, that `getche` function displays the character on the display before returning its value to the calling program. The prototype for this function is declared in the `conio.h` file. This file, therefore has to be included in a program using the `getche` function. Consider the following program,

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
      char ch;
      printf("Type a character : ");
      ch = getche();
      printf("\n");
      printf("You typed %c\n",ch);
      return(0);
}
```

An execution session of this program is shown below,

```
Type a character : w
You typed w
```

Please note that after typing `w`, the user does not need to press the enter key, the `getche` function returns the character which is assigned to variable `ch`. The value of this variable is printed on the display by the `printf` function call.

Consider another program shown below,

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
	char ch1;
	char ch2;
	char ch3;
	char ch4;
	char ch5;
	printf("This program uses getche() function\n");
	printf("Enter characters now ");
	ch1 = getche();
	ch2 = getche();
	ch3 = getche();
	ch4 = getche();
	ch5 = getche();
	putchar('\n');
	putchar(ch1);
	putchar(ch2);
	putchar(ch3);
	putchar(ch4);
	putchar(ch5);
	return(0);
}
```

An execution session of this program is shown below,

```
This program uses getche() function
Enter characters now Hello
Hello
```

In this program, as the user types characters on the keyboard, the `getche` function calls return the values of these character which get assigned to the corresponding variables. The user does not need to press the enter key for the function to return the value of the character. Once five

characters have been typed by the user, i.e., the last `getche` function call has been executed, a series of `putchar` function calls print these character values on the display.

The `getch` function has the same functionality as the `getche` function, except that the `getch` function does not echo the typed character on the display. The prototype for this function is also declared in `conio.h` file. Consider the following program,

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
      char ch;
      printf("Type a character : ");
      ch = getch();
      printf("\n");
      printf("You typed %c\n",ch);
      return(0);
}
```

An execution cycle of this program is given below,

```
Type a character :
You typed w
```

Note that not only does `getch` provide direct input, but it also does not echo the character entered on the display.

3.4 Integer Data Type, `int`, `long` and `short`

The data type `int` is the principal working type of the C language. This type, along with the other integral types such as `char`, `short` and `long`, is designed for working with the integer values that are representable on the machine.

In mathematics, the natural numbers are 0, 1, 2,..., and these numbers, along with their negatives comprise the integers. On a machine, only a finite portion of these integers are representable for a given integral type. Typically, an `int` is stored in a machine word. Some computers use a machine word of 2 bytes (16 bits), others use a machine word of 4 bytes (32 bits). There are other possibilities, but many machines fall into these two categories. Examples of machines with two byte words are the personal computers. Examples of machines with four byte words are the high end personal computers, workstations made by Apollo, Hewlet-

Packard, Next, Silicon Graphics, Sun etc. Because the word size varies from one machine to another, the number of distinct values that an `int` can hold is machine dependent.

So for a computer with a word size of 4 bytes, a variable of type `int` can have 2^{32} distinct states. Half of these states are used to represent negative integers and half are used to represent positive integers. Similarly, if the word size of a computer is 2 bytes, then a variable of type `int` can take on only 2^{16} distinct states. Again, half of these states are used to represent negative integers and half are used to represent positive integers.

The keyword `int` is used to declare variables of that type. A typical declaration is shown below,

```
int AnIntegerNumber;
```

To declare more than one variable, each variable can be declared separately or these can be declared in the same statement as shown below,

```
int IntegerNumber, IntegerNumber2, IntegerNumber3;
```

In the above statement, storage space for three integer variables is arranged and a name is associated with each one of them.

Integer variables can be initialised when they are declared. This is accomplished by following the variable name with an equals sign and the value that variable should have. For example,

```
int Age = 32;
int Packets = 12, DataBytesPerPacket = 1500;
int InterfaceCards, PortsPerCard = 2;
```

In the last line, only `PortsPerCard` is initialised. A quick reading might lead one to think that `InterfaceCards` is also initialised to `2`. It is, therefore, best to avoid putting initialised and non-initialised variables in the same declaration statement.

The various integer values in the last example (i.e., `32`, `12`, `1500` and `2`) are integer constants. A number written without a decimal point or an exponent is recognised by C as an integer. C

treats most integer constants as numbers of type `int`. Very large values may be treated differently, as will be shown later.

Normally C assumes that the integer constants written by the programmer are decimal numbers, i.e., in the base 10 number system. However, octal (base 8) and hexadecimal (base 16) numbers are popular with many programmers. Because, 8 and 16 are powers of 2 and 10 is not, these number systems are more natural for computers. To write numbers in octal and hexadecimal notations, programmers have to use correct prefixes so that C will know which number system the programmer is using. A `0` (zero) prefix means that the number is to be treated as an octal number. For example, the decimal `16` is written as `020` in C octal. Similarly, a prefix of `0x` or `0X` (zero-exe) means that the programmer is writing a hexadecimal number. Therefore, `16` is written as `0x10` or `0X10` in hexadecimal.

One important point to realise, however, is that this option of using different number systems is provided as a service for programmers' convenience. It does not affect how the number is stored. This means that `16`, `020` or `0x10` are all stored exactly the same way, i.e., in the binary code used internally by the computers.

C offers three ***adjective*** keywords to modify the basic data type `int`. These are `unsigned`, `long` and `short`. The type `short int`, or more briefly, `short` ***may*** use less storage than `int`, thus saving space when only small numbers are needed. The type `long int`, or `long` ***may*** use more storage than `int`, thus allowing larger integers to be used. The keyword `unsigned` shifts the range of numbers that can be stored. For example, a 2-byte `unsigned int` allows a range from 0 to 65535 in value instead of from -32768 to +32767. C also recognises `unsigned long int`, or `unsigned long`, and `unsigned short int` or `unsigned short` as valid types.

It should be noted that C only guarantees that `short` is no longer than `int` and that `long` is no shorter than `int`. The idea is to fit the types to the machine. On IBM PC AT and below, for example, an `int` and a `short` are both 16 bits, while a `long` is 32 bits. On a VAX computer, however, a `short` is 16 bits, while both `int` and `long` are 32 bits. The natural word size on a VAX is 32 bits. Since this allows integer values in excess of 2 billion, the implementors of C on the VAX did not see a necessity for anything larger. Thus `long` is kept of the same size as the `int`. But for many applications, integers of that size are not needed, so a space saving `short`

was created. The IBM PC AT, on the other hand, has only a 16 bit word, which means that a larger `long` was needed. *The most common practice today is to set up* `long` *as 32 bits,* `short` *as 16 bits and* `int` *to either 16 bits or 32 bits, depending upon the machine's natural word size.*

`unsigned` types are generally used in counting applications as negative numbers are not needed and the `unsigned` types allow higher positive numbers than the `signed` types. The `long` type is used if values used in the application cannot be handled by `int`. However, on systems for which `long` is bigger than `int`, using `long` may slow down calculations. `long` may, therefore, not be used if it is not essential. If the code is being written on a machine for which `int` and `long` are synonymous, and if the application does require 32 bit integers, `long` may be used instead of `int` so that the program may function as per specifications if ported to a 16 bit machine. `short` is used to save storage space, i.e., when 16 bit values are required by the application on a 32 bit machine. Saving storage space is usually important only if the program uses large arrays of integers.

Variables of other integer types are declared in the same manner as those of the `int` type. The following list shows several examples,

```
long int ALongInteger;
long AnotherLongInteger;
short int AShortInteger;
short AnotherShortInteger;
unsigned int AnUnsignedInt;
unsigned long AnUnsignedLong;
unsigned short AnUnsignedShort;
```

Normally, a number like `2345` in a program is stored as an `int` type. However large numbers like `100000` are stored as type `long` on machines on which `int` cannot hold such a large number. Sometimes a programmer may need to store a small number as a long integer. To cause a constant value to be stored as type `long`, the programmer can add an `L` as a suffix. Thus on a system with a 16-bit `int` and a 32-bit `long`, the integer `7` is stored in 2 bytes, and the integer `7L` in 4 bytes. The suffix `L` can also be used with octal and hexadecimal numbers.

During runtime, an integer variable may be assigned an incorrect value, i.e., a value greater than it can hold. For instance, assigning a value `76000` to a variable of type `int`. ***Integer overflow*** is said to occur here. Consider the following program,

```
#include<stdio.h>
int main(void)
{
      int i = 32760;
      printf("%d :: %d :: %d\n",i,i+10,i+20);
      return(0);
}
```

The output of this program is given below,

```
32760 :: -32766 :: -32756
```

The integer `i` is acting like a car's odometer. When it reaches its maximum value, it starts over at the beginning. The main difference is that an odometer begins at `0`, while integer variables of the type `int` begin at `-32768`. It is important to note that when an integer overflow occurs, the program continues to execute, but provides logically incorrect results. For this reason, the programmer must strive at all times to keep the values of integer expressions within the proper range.

Various format specifiers used with integral types to display data as decimal numbers are given in the following table,

Format Specifier	Type
%d	int
%ld	long
%u	unsigned int
%lu	unsigned long

Table 3-6 : Format Specifier for Integral Types

Additionally, a digit can be used between the `%` and the first letter (e.g., `d`) to specify the minimum field width, e.g., `%4d`. A wider field will be used if the printed number or string does not fit in the field. A minus (-) specifies that the item will be printed beginning at the left of its field width. Normally, the item is printed so that it ends at the right of its field, e.g., `%-10d`. Consider the following example,

```
#include <stdio.h>
int main(void)
{
      int i;
      i = 45;
      printf("%4d:\n",i);
      printf("%-4d:\n",i);
      i = 12345;
      printf("%4d:\n",i);
      printf("%-4d:\n",i);
      return(0);
}
```

The output of this program is given below,

```
  45:
45  :
12345:
12345:
```

The first line shows that the 2-digit value is printed in a 4-digit field aligned to the right. The second line shows that the 2-digit value is printed in a 4-digit field aligned to the left. The last two lines show that a 5-digit value is being printed in a 4-digit field. It can be seen that the system automatically selects a field width suitable for the value as the 4 digits originally specified are insufficient for displaying a 5-digit value.

3.5 Floating Point Numbers, `float` and `double`

The various integer types serve well for most software development projects. However, mathematically oriented programs often make use of ***floating point*** numbers. In C, such numbers are stored in variables of type `float`. This type allows the representation of a much greater range of numbers, including decimal fractions. Floating point numbers are analogous to scientific notation, a system used by scientists to express very large and small numbers.

In scientific notation, numbers are represented as decimal numbers times powers of ten. Some examples are given in the table 3-7. The first column shows the usual notation, the second column scientific notation and the third column exponential notation. Exponential notation is the way scientific notation is usually written for and by computers, with the `e` followed by the power of ten.

Number	Scientific Notation	Exponential Notation

1,000,000,000	= 1.0×10^{9}	= 1.0e9
123,000	= 1.23×10^{5}	= 1.23e5
322.56	= 3.2256×10^{2}	= 3.2256e2
0.000056	= 5.6×10^{-5}	= 5.6e-5

Table 3-7 : Scientific and Exponential Notations

The possible values that a floating type can be assigned are described in terms of attributes called ***precision*** and ***range***. Precision describes the number of significant decimal places that a floating value carries. Range describes the limits of the largest and smallest positive floating values that can be represented in a variable of that type.

Usually 32 bits are used to store a floating point number. Eight bits are used to give the exponent its value and sign, and 24 bits are used to represent the non-exponent part. This produces a precision of six decimal places and a range of $\pm(10^{-38}$ to $10^{+38})$.

Many systems also support the type `double` for double precision floating point numbers. Typically, a C compiler will provide more storage for a variable of type `double` than for one of type `float`, although it is not required to do so. On most machines, a variable of type `double` is stored in twice as many bits as a variable of type `float`, i.e., 64 bits. Thus a variable of type `double`, on most machines, has an approximate precision of 15 significant figures and an approximate range of $\pm(10^{-308}$ to $10^{+308})$.

To declare variables of types `float` and `double`, name of the variable should follow the keyword `float` or `double` respectively. To declare several variables in one declaration statement, the list of variable names should follow the keyword `float` or `double`. Variable names in the list should be separated by commas. Some examples are shown below,

```
float AFloatingPointNumber;
double MassOfEarth, MassOfMoon, Distance;
```

Variables of type `float` and `double` can be initialised as shown below,

```
float PlancksConstant = 6.63e-34;
```

The basic form of a floating point constant is a signed series of digits including the decimal point, then an `e` or `E`, then a signed exponent indicating the power of 10 used. However, decimal

notation can also be used to represent floating point constants. Some examples of valid floating point constants are given below,

```
-.157e+12   2.55E-4     3.141       .2    100   4e16
```

It should be noted that these is no space within the floating point constant number. *Floating point constants are taken to be double precision floating point numbers*. Consider the following program segment,

```
float some;
some = 4.0 * 2.0;
```

Here, the variable `some` is of the type `float`. However, `2.0` and `4.0` are stored as type `double`, using (typically) 64 bits for each. The product (`8.0`) is calculated using double precision arithmetic, and only then is the answer trimmed down to regular size for the type `float`. This ensures maximum precision for calculations.

Format specifiers `%f` and `%e` are used to print floating point numbers. `%f` is used to print the floating point numbers in the decimal notation whereas `%e` is used to print the floating point numbers in the exponential notation. Additionally, field width specifiers are also used with the format specifiers to print the floating point numbers. Field width specifiers for the floating point numbers are written as real numbers between the characters `%` and `f`. The number following the decimal point in the field width specifier controls how many characters will be printed following the decimal point. The digits preceding the decimal point in the field width specifier controls the width of the space to be used to contain the number when it is printed. Moreover, a minus sign preceding the field width specifier will put the output on the left side of the field instead of the right. Consider the following program,

```
#include <stdio.h>

int main(void)
{
       float ANumber;
       ANumber = 314.5615;
       printf("Printing the number in decimal notation          :: %f\n",
                ANumber);
       printf("Printing the number in exponential notation      :: %e\n",
                ANumber);
       printf("Printing the number in decimal notation (9.2)    :: %9.2f\n",
                ANumber);
       printf("Printing the number in exponential notation (9.2) :: %9.2e\n",
```

```
                ANumber);
        printf("Printing the number in decimal notation    (-9.2) :: %-9.2f\n",
                ANumber);
        printf("Printing the number in exponential notation (-9.2):: %-9.2e\n",
                ANumber);
        printf("Printing the number in decimal notation (9.3)     :: %9.3f\n",
                ANumber);
        printf("Printing the number in exponential notation (9.3) :: %9.3e\n",
                ANumber);
        return(0);
}
```

The output of this program is as follows,

```
Printing the number in decimal notation           :: 314.561493
Printing the number in exponential notation       :: 3.145615e+02
Printing the number in decimal notation (9.2)     ::    314.56
Printing the number in exponential notation (9.2) ::  3.15e+02
Printing the number in decimal notation    (-9.2) :: 314.56
Printing the number in exponential notation (-9.2):: 3.15e+02
Printing the number in decimal notation (9.3)     ::   314.561
Printing the number in exponential notation (9.3) :: 3.146e+02
```

Thus the digit `9` in `%9.2f` or `%9.2e` specifies the total number of characters of the number to be printed. The `.2` specifies the number of digits after the decimal place that should be printed.

The main points that one must be aware of while using floating point numbers are

1. not all real numbers are representable,
2. floating point arithmetic operations, unlike the integer arithmetic operations, need not be exact.

The latter is usually of not much concern for small computations. For very large computations, such as numerically solving a large system of ordinary differential equations, a good understanding of rounding effects, scaling, etc., may be necessary.

3.6 The `#define` Preprocessor Directive

The `#define` preprocessor directive, like all preprocessor directives, begins with a `#` symbol in the far left column. Note that the ANSI Standard permits the `#` symbol to be proceeded by a space or a tab. It can appear anywhere in the source code, and the definitions hold from its place of appearance to the end of the file. It is mainly used for defining ***symbolic constants***. Here it is used to assign names (`DAYS_IN_YEAR` or `PI`, for instance) to constant values (such as `365` or `3.14159`).

Consider the following program, it calculates the area and circumference of a circle.

```
#include <stdio.h>
#define PI 3.14159

int main(void)
{
      float Radius;
      float Area;
      float Circumference;
      printf("Enter the radius of the circle ");
      scanf("%f", &Radius);
      Circumference = 2 * PI * Radius;
      Area = PI * Radius * Radius;
      printf("Circumference = %7.2f\n",Circumference);
      printf("Area          = %7.2f\n",Area);
      return(0);
}
```

A typical execution session of this program produces the following output on the display,

```
Enter the radius of the circle 56
Circumference =  351.86
Area          = 9852.03
```

In this program, the preprocessor first looks for all program lines beginning with the hash sign (`#`). When it sees the `#define` directive, it goes through the entire program, and at every place it finds `PI` it substitutes the phrase `3.14159`. This is a mechanical process, i.e., simply substituting one group of characters `3.14159` for another one `PI`. The structure of a `#define` directive is shown below.

The phrase on the left (`PI`), which will be searched for, is called the ***identifier***. The phrase on the right (`3.14159`), which will be substituted for is called the ***text***. A space separates the identifier from the text. By convention, the identifier (in this case, `PI`) is written in uppercase

letters. This makes it easy when looking at the listing to tell which parts of the program will be altered by the `#define` directive.

Using the `#define` directive has two main advantages over simply writing the constant values at the required locations within the source code. Firstly, it improves the readability of the code. If a constant is given a suitable symbolic identifier within a program, its easier for the readers to read and understand the meaning and the use of the symbolic identifiers at various locations within the program. Secondly, if a constant has to be used at several places within a program, and at some later stage the value of the constant needs to be modified, the programmer will have to go through the entire source code and manually modify each occurrence of the constant. However, if the constant has been defined as a symbolic identifier in a `#define` directive, the programmer is only required to make one change, i.e., in the `#define` directive itself. The change will be made automatically to all occurrences of the constant before compilation begins.

It is also possible for the programmers to declare variables of appropriate types, provide them suitable names and permit one change to effect many occurrences of the constant. However, `#define` directives allow the programmers to produce more efficient and compact code for constants than it can for variables. Moreover, it is possible that the value of a variable being used as a constant gets changed inadvertently during program execution. This may cause the program to give incorrect results.

3.7 The Use of `typedef`

The `typedef` declaration causes the compiler to recognise a different name for a variable type. Thus, `typedef` feature lets the programmers create their own names for a type. In that respect, it is a lot like `#define` but with the following three differences,

1. Unlike `#define`, `typedef` is limited to giving symbolic names to data types only.
2. The `typedef` function is performed by the compiler, not the preprocessor.
3. Within its limits, `typedef` is more flexible than `#define`.

Consider the following program,

```
#include <stdio.h>
typedef unsigned char BYTE;
```

```
int main(void)
{
      BYTE x;
      x = 250;
      printf("%d, ",x);
      x += 5;
      printf("%d, ",x);
      x += 5;
      printf("%d, ",x);
      x += 5;
      printf("%d\n",x);
      return(0);
}
```

It is assumed here that variable `x` will be used in a context in which declaring them to be of type `BYTE` is more meaningful than declaring them to be of the type `unsigned char`. For instance, it might be used to represent values in a microprocessor's registers. Uppercase letters are often used to make it clear that a renamed data type is being used.

An execution cycle of this program gives the following output on the display,

```
250, 255, 4, 9
```

The scope of this definition depends on the location of the `typedef` statement. If the definition is inside a function, the scope is local to that function. If the definition is outside a function, then the scope is global.

In addition to choosing suitable names for data types, `typedef` can also be used to abbreviate the names for data types. Thus, when used properly, `typedef` can help to clarify the source code.

3.8 The `sizeof` Operator

C provides the unary operator `sizeof` to find the number of bytes needed to store an object. It has the same precedence and associativity as all the other unary operators. An expression of the form `sizeof(object)` returns an integer that represents the number of bytes needed to store the object in memory. An object can be a type such as `int` or `float`, or it can be an expression such as `a + b`, or it can be an array or structure type.

The following program uses the `sizeof` operator to print the information about the storage requirements for the fundamental data types on a given machine.

```
#include <stdio.h>

int main(void)
{
      printf("Size of char                    :: %d\n",sizeof(char));
      printf("Size of int                     :: %d\n",sizeof(int));
      printf("Size of short                   :: %d\n",sizeof(short));
      printf("Size of long                    :: %d\n",sizeof(long));
      printf("Size of float                   :: %d\n",sizeof(float));
      printf("Size of double                  :: %d\n",
              sizeof(double));

      return(0);
}
```

Output of an execution cycle of this program is as follows,

```
Size of char                    :: 1
Size of int                     :: 2
Size of short                   :: 2
Size of long                    :: 4
Size of float                   :: 4
Size of double                  :: 8
```

3.9 Mathematical Functions

There are no built in mathematical functions in C. Functions such as

```
sqrt()      pow()       exp()       log()       sin()       cos()
```

are available in the mathematics library, which is conceptually part of the standard library. All of these functions, except the power function `pow()`, take a single argument of type `double` and return a value of the type `double`. The power function takes two arguments of type `double` and returns a value of type `double`. `math.h` file contains the prototypes for the functions in the mathematics library. It therefore needs to be included before function `main` with the help of the `#include` preprocessor directive.

Consider the following program. It accepts a floating point number. First it prints out the square root of the number. Then it asks the user to enter another number and raises the former to the power of the latter.

```
#include <stdio.h>
#include <math.h>
```

```
int main(void)
{
      float InNumber,Power;
      float SquareRoot, RaisedNumber;
      printf("Enter number :: ");
      scanf("%f",&InNumber);
      SquareRoot = sqrt(InNumber);
      printf("Square Root of %7.2f = %7.2f\n",InNumber,SquareRoot);
      printf("Enter power :: ");
      scanf("%f",&Power);
      RaisedNumber = pow(InNumber,Power);
      printf("%7.2f raised to %7.2f = %7.2f\n",InNumber,Power,
              RaisedNumber);
      return(0);
}
```

Following lines show an execution session of this program,

```
 Enter number :: 9.00
 Square Root of    9.00 =    3.00
 Enter power :: 2.00
    9.00 raised to    2.00 =    81.00
```

In many languages, the function `abs()` returns the absolute value of its real argument. In this respect C is different. In C, the function `abs()` takes an argument of type `int` and returns its absolute value as an `int`. Its function prototype is in `stdlib.h`. For real numbers (i.e., of the types `float` and `double`), the C programmers should use `fabs()`, which takes an argument of type `double` and returns its absolute value as `double`. Its function prototype is in `math.h`.The name `fabs` stands for floating absolute value.

3.10 Type Conversions and Type Casting

Statements and expressions should normally use variables and constants of just one type. If, however, data of different types needs to be mixed in expressions and statements, C uses a set of rules to make type conversion automatically. This can be a convenience, but it can also be a danger, especially if types are mixed inadvertently.

Type conversion rules used in C are given below,

1. In any operation involving two types, both values are converted to the *higher* ranking of the two types. This process is called ***promotion***.
2. The ranking of types, from highest to lowest, is `double`, `float`, `long`, `int` , `short` and `char`.

3. In an assignment statement, the final result of the calculations is converted to the type of the variable that is being assigned a value. This process can result in promotion, as described in rule 1, or demotion, in which a value is converted to a lower ranking type.

Promotion is usually a smooth and an uneventful process, but demotion may lead to problems. This is because the lower-ranking type may not be big enough to hold the complete number. For example, a variable of type `char` can hold an integer value `101`, but not the integer value `22334`.

This process of conversion takes place automatically and is, therefore, also known as ***automatic conversion***, ***implicit conversion***, ***coercion***, ***promotion*** and ***demotion***.

In addition to implicit conversions, which can occur across assignments and in mixed expressions, there are explicit conversions call ***casts***. Casting consists of preceding the quantity with the name of the desired type in parenthesis. The parenthesis and the type name together constitute a ***cast operator***. The cast operator is represented as `(type)` where the actual type desired is substituted for the word `type`.

Consider the following program,

```
#include <stdio.h>

int main(void)
{
      int AnInteger;
      int AnotherInteger;
      AnInteger = 1.6 + 1.7;
      AnotherInteger = (int) 1.6 + (int) 1.7;
      printf ("AnInteger = %d\n",AnInteger);
      printf("AnotherInteger = %d\n",AnotherInteger);
      return(0);
}
```

In the statement, `AnInteger = 1.6 + 1.7;` automatic conversion takes place. First `1.6` and `1.7` are added to yield `3.3`. This number is then converted through truncation to the integer `3` in order to match the variable `AnInteger` of type `int`. In the statement `AnotherInteger = (int) 1.6 + (int) 1.7;` `1.6` and `1.7` are converted to integer values `1` before addition so that the variable `AnotherInteger` of type `int` is assigned the value `2`.

The output from an execution session of this program is given below,

```
AnInteger = 3
AnotherInteger = 2
```

Flow of Control

Statements in a program are normally executed one after the other. This is known as the ***sequential flow of control***. Often it is desirable to alter the sequential flow of control to provide for a choice of action or a repetition of action. Programmers need to use suitable programming constructs provided by the language to alter program's flow of control. These constructs use relational, equality and logical operators in expressions to determine how the flow of control should be altered.

4.1 Relational Operators and Expressions

Table 4-1 shows the relational operators provided by the C language.

Operator	Operation
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Table 4-1 : Relational Operators

All relational operators are binary. They each take two expressions as operands and return an integer value representing true or false depending upon the value of the expression. For example, consider the expression `a<b`. If the value of `a` is less than the value of `b`, then the expression returns an integer value representing true. Alternatively, it returns a value representing false. The precedence of relational operators is lower than the arithmetic operators (i.e., `+`, `-`, `*`, `/`, `%`) and they have the left to right associativity. Consider the examples shown in table 4-2.

Declarations and Initialisations			
`char c = 'w';`			
`int i = 1, j = 2, k = -7;`			
`double x = 7e+33, y = 0.001;`			
Expression	Equivalent Expression	Substituting Values	Result
`'a' + 1 < c`	`('a' + 1) < c`	`'b' < 'w'`	`true`
`-i-5*j>=k+1`	`((-i)-(5*j))>=(k+1)`	`-11 >= -6`	`false`
`3 < j < 5`	`(3 < j) < 5`	`0 < 5`	`true`
`-3.3333 <= x + y`	`(-3.3333) <= (x + y)`	`-3.3333 <= 7e33`	`true`
`x < x + y`	`x < (x + y)`	`7e33 < 7e33`	`false`

Table 4-2 : Use of Relational Operators

4.2 Equality Operators and Expressions

The equality operators `==` and `!=` are binary operators acting on expressions. They yield either the `int` value `0` or the int value `1`. Intuitively, an equality expression such as `a == b` is either true or false. An equivalent expression is `a - b == 0` which is implemented at the machine level. If `a` equals `b`, then `a - b` has the value `0` and `0 == 0` is true. In this case the expression `a == b` will yield the `int` value corresponding to true. If `a` is not equal to `b` then, `a == b` will yield an int value corresponding to false. Equality operators have precedence lower than the relational operators and they associate from left to right.

The following table shows some examples of equality operators.

Declarations and Initialisations		
`int i = 1, j = 2, k = 3;`		
Expression	Equivalent Expression	Result
`i == j`	`j == i`	`false`
`i != j`	`j != i`	`true`
`i + j + k == -2 * -k`	`((i + j) + k) == ((-2) * (-k))`	`true`

Table 4-3 : Use of Equality Operator

Observe that the expression `a != b` is equivalent to the expression `!(a == b)`. Also note that the two expressions, `a == b` and `a = b` are visually similar. They are close in

form but are radically different in function. The expression `a == b` is a test for equality whereas `a = b` is an assignment expression.

4.3 Concept of Truth in C

Consider the following program,

```
#include <stdio.h>

int main(void)
{
      int TruthValue;
      TruthValue  = (10 < 5);
      printf("10 < 5 is false :: TruthValue = %d\n",TruthValue);
      TruthValue = (5 > 2);
      printf("5 > 2 is true :: TruthValue = %d\n",TruthValue);
      return(0);
}
```

In this example, the value of a relational expression is assigned to a variable `TruthValue` of type `int`. The value of `TruthValue` is then displayed to the user. An execution session of this program displays the following lines on the screen,

```
10 < 5 is false :: TruthValue = 0
5 > 2 is true :: TruthValue = 1
```

According to the results displayed by this program, truth for C is an integer value `1` and falsity is an integer value `0`. In fact, all non-zero values are treated as true and only `0` is recognised as false.

4.4 Logical Operators and Expressions

Logical operators used in C are listed in table 4-4. The logical operator `!` is unary and the logical operators `&&` and `||` are binary. All of these operators when applied to expressions yield either the `int` value `0` or the `int` value `1`.

Logical negation can be applied to an arithmetic expression. If an expression has a zero value, then its negation will yield the `int` value `1`. If the expression has a nonzero value, then its negation will yield the `int` value `0`.

Operator	Symbol
(unary) negation	!
logical AND	&&
logical OR	||

Table 4-4 : Logical Operator

The syntax for the negation operator is given below.

```
! expression
```

Table 4-5 is the truth table for the negation operator.

Value of `expression`	Value of `!expression`
zero	1
nonzero	0

Table 4-5 : Truth Table for the Negation Operator

Consider the following program,

```
#include <stdio.h>

int main(void)
{
      int Num = 10;
      printf("Value of Num = %d\n\n",Num );
      printf("Value of Num-2 = %d :: Value of !(Num-2) = %d\n",
              Num-2,!(Num-2));
      printf("Value of Num-10 = %d :: Value of !(Num-10) = %d\n",Num-10,
              !(Num-10));
      printf("Value of Num > 5 = %d :: Value of !(Num>5) = %d\n",
            (Num > 5), !(Num>5));
      return(0);
}
```

An execution session of this program prints the following on the display,

```
Value of Num = 10
```

```
Value of Num-2 = 8 :: Value of !(Num-2) = 0
Value of Num-10 = 0 :: Value of !(Num-10) = 1
Value of Num > 5 = 1 :: Value of !(Num>5) = 0
```

Although logical negation is a very simple operator, there is one subtlety. The operator `!` in C is unlike the not operator in ordinary logic. If `s` is a logical statement, then

```
not(not (s)) = s
```

whereas in C, the value of `!!5`, for example, is `1`. This is because `!!5` is equivalent to `!(!5)`. Furthermore, `!(!5)` is equivalent to `!(0)` which has a value `1`.

The binary logical operators `&&` and `||` also act on expressions and yield either the `int` value `0` or the `int` value `1`. `&&` is the logical AND operator and `||` is the logical OR operator. `exp1 && exp2` is true only if both `exp1` and `exp2` are true. `exp1 || exp2` is true if either or both of `exp1` and `exp2` are true. The truth table for these operators is shown below,

Exp 1	Exp 2	Exp 1 && Exp 2	Exp 1 || Exp 2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Table 4-6 : Truth Table for Logical AND and Logical OR Operators

The following program automatically generates the truth table shown above.

```
#include <stdio.h>

int main(void)
{
      int exp1,exp2;
      int exp1_AND_exp2;
      int exp1_OR_exp2;
      printf("exp1   exp2   exp1&&exp2   exp1||exp2\n");
      exp1 = 0;
      exp2 = 0;
      exp1_AND_exp2 = exp1 && exp2;
      exp1_OR_exp2 = exp1 || exp2;
      printf("%2d %6d %9d %11d\n",exp1,exp2,exp1_AND_exp2,exp1_OR_exp2);
```

```
        exp1 = 0;
        exp2 = 1;
        exp1_AND_exp2 = exp1 && exp2;
        exp1_OR_exp2 = exp1 || exp2;
        printf("%2d %6d %9d %11d\n",exp1,exp2,exp1_AND_exp2,exp1_OR_exp2);
        exp1 = 1;
        exp2 = 0;
        exp1_AND_exp2 = exp1 && exp2;
        exp1_OR_exp2 = exp1 || exp2;
        printf("%2d %6d %9d %11d\n",exp1,exp2,exp1_AND_exp2,exp1_OR_exp2);
        exp1 = 1;
        exp2 = 1;
        exp1_AND_exp2 = exp1 && exp2;
        exp1_OR_exp2 = exp1 || exp2;
        printf("%2d %6d %9d %11d\n",exp1,exp2,exp1_AND_exp2,exp1_OR_exp2);
        return(0);
}
```

Different values are assigned to the variables `exp1` and `exp2` in the program. Logical AND and OR operators are applied to these variables and the results of the AND and OR operations are assigned to variables `exp1_AND_exp2` and `exp1_OR_exp2` respectively. Results are then printed on the display. The output of this program is given below,

```
exp1   exp2   exp1&&exp2   exp1||exp2
 0      0         0           0
 0      1         0           1
 1      0         0           1
 1      1         1           1
```

It is important to note that in the evaluation of expressions that are the operands of `&&` and `||`, the evaluation process stops as soon as the outcome, true or false is known. This is called ***short circuit evaluation***. It is an important property of these operators. Suppose that `exp1` and `exp2` are expressions and that `exp1` has a zero value. In the evaluation of the logical expression `exp1 && exp2` the evaluation of `exp2` will not occur because the value of the logical expression as a whole is already determined to be `0`. Similarly, if `exp1` has a nonzero value, then in the evaluation of `exp1||exp2`, the evaluation of `exp2` will not occur because the value of the logical expression as a whole is already determined to be `1`.

4.5 The Compound Statement

A compound statement is a series of declarations and statements surrounded by braces. The chief use of the compound statement is to group statements into an executable unit. When declarations are given at the beginning of a compound statement, that compound statement is also called a ***block***. In C, wherever it is syntactically correct to place a statement, it is also syntactically correct to place a compound statement. *A compound statement is itself a statement*. Consider the following program,

```
#include <stdio.h>

int main(void)
{
      int ANumber, BNumber;
      {
            ANumber = 10;
            {
                  printf("Enter BNumber :: ");
                  scanf("%d",&BNumber);
                  ANumber += BNumber;
            }
            printf("ANumber = %d ::: BNumber = %d\n",ANumber, BNumber);
      }
      return(0);
}
```

An execution cycle of this program is as follows,

```
Enter BNumber :: 12
ANumber = 22 ::: BNumber = 12
```

Note that in this example, there is a compound statement within a compound statement.

4.6 The Empty Statement

The empty statement is written as a single semicolon. It is useful where a statement is needed syntactically, but no action is required semantically. Consider the following program,

```
#include <stdio.h>

int main(void)
{
      printf("This statement is printed before 3 empty statements\n");
      ;
      ;
```

```
      ;
      printf("This statement is printed after 3 empty statements\n");
      return(0);
}
```

The three semicolons following the statement

`printf("This statement is printed before 3 empty statements\n");`

are the three empty statements. The output of this program is as follows,

```
This statement is printed before 3 empty statements
This statement is printed after 3 empty statements
```

4.7 The `if` and the `if-else` Statements

The general form of an `if` statement is

```
      if (expr)
            statement;
```

If `expr` is nonzero (true) then `statement;` is executed, otherwise `statement;` is skipped and control passes to next statement. Consider the following example,

```
#include <stdio.h>

int main(void)
{
      float Pay;
      float Tax = 0.0;
      float Net;
      printf ("Enter your pay ");
      scanf("%f",&Pay);
      if (Pay > 10000.0)
            Tax = Pay * 0.1;
      Net = Pay - Tax;
      printf("Pay = %.2f\n",Pay);
      printf("Tax = %.2f\n",Tax);
      printf("Net = %.2f\n",Net);
      return(0);
}
```

This program accepts a floating point input from the user and assigns it to the variable `Pay` of type `float`. If the value of `Pay` is greater than `10000.0`, tax is calculated as 10 percent of that value. The result is assigned to variable `Tax` of type `float`. This variable is initialised at declaration to `0.0`. If the value of `Pay` is less than or equal to `10000.0`, the

statement `Tax = Pay * 0.1;` is not executed and the control is transferred to the statement `Net = Pay - Tax;`. An execution cycle of this program is shown below,

```
Enter your pay 5000.0
Pay = 5000.0
Tax = 0.00
Net = 5000.0
```

Consider another execution cycle where the user enters a value greater than 10000.0

```
Enter your pay 12000.0
Pay = 12000.0
Tax = 1200.00
Net = 10800.0
```

It is possible to execute a number of statements under the control of a single `if` expression. All such statements should be grouped into a single compound statement.

The `if-else` statement is closely related to the if statement. It has the general form,

```
if (expression)
      statement1;
else
      statement2;
```

If `expression` is nonzero, then `statement1` is executed and `statement2` is skipped. If `expression` is zero, then `statement1` is skipped and `statement2` is executed. In both cases control then passes to the next statement. The following program allows the user to enter a character. If the character entered is a lowercase alphabet, the program prints the message

```
A lowercase alphabet was entered
```

If the character entered is not a lowercase alphabet then the program prints the message

```
The character entered is not a lowercase alphabet
```

Before the program terminates, the character entered by the user is printed on the screen.

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char ch;
   printf ("Enter a character ");
   ch = getche();
   printf("\n");
   if ((ch >=97) && (ch<=122))
   {
      printf("A lowercase alphabet was entered\n");
   }
   else
   {
      printf("The character entered is not a lowercase alphabet\n");
   }
   printf("You entered '%c'\n",ch);
   return(0);
}
```

Because `if` and `if-else` statements are themselves statements, these can be used as the statement parts of other `if` and `if-else` statements. For example,

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char ch;
   printf("Enter a character ");
   ch = getche();
   printf("\n");
   if (((ch >= 'a') && (ch <= 'z')) || ((ch >= 'A') && (ch <= 'Z')))
   {
      printf("An alphabet is entered\n");
      if ((ch >= 'a') && (ch <= 'z'))
      {
         printf("It is a lowercase alphabet\n");
      }
      else
      {
         printf("It is an uppercase alphabet\n");
      }
   }
   else
   {
      if ((ch >= '0') && (ch <= '9'))
      {
         printf("The character entered is a numeric digit\n");
      }
      else
      {
```

```
            printf("The character entered is neither an alphabet ");
            printf("nor a numeric digit\n");
         }
      }
      printf("You entered %c\n",ch);
      return(0);
}
```

A few execution sessions of this program are given below,

```
Enter a character %
The character entered is neither an alphabet nor a numeric digit
You entered %

Enter a character e
An alphabet is entered
It is a lowercase alphabet
You entered e

Enter a character T
An alphabet is entered
It is an uppercase alphabet
You entered T

Enter a character 5
The character entered is a numeric digit
You entered 5
```

It is generally considered to be a safe practice to use compound statements with the conditional statements even if one statement has to be executed. This is because the braces of compound statements clearly mark the boundaries of the statements that need to be executed once a specific condition is fulfilled.

4.8 Multiple Choices with `switch` and `break`

The `if-else` construction makes it easy to write programs that choose between two alternatives. Sometimes, however, a program needs to choose one of several alternatives. This can be achieved by using several `if-else` constructs, but in many cases it is more convenient to use the `switch` statement provided by C.

`switch` is a multiway conditional statement generalising the `if-else` statement. The syntax of this statement is shown below.

```
switch (integral expression)
{
      case value1:
            statement1;
            break; /* Optional */
      case value2:
            statement2;
            break; /* Optional */
      case value3:
            statement3;
            break; /* Optional */
      case value4:
            statement4;
            break; /* Optional */
      default:     /* This entire section is optional */
            statement5;
}
```

The expression in parenthesis following the keyword `switch` is the controlling variable as the value of this expression determines the statements that must be executed and the statements that must be skipped. This expression is evaluated and the flow of control is transferred to the line with the `case` label having a constant value that matches the value of the expression. If a match is not found, the flow of control is transferred to the line with the `default` label. If there is no line with the `default` label, the switch statement is terminated and the flow of control is transferred to the statement immediately after the `switch` statement.

The `switch` statement is also terminated when a `break` statement is encountered. If there is no `break` statement, all the statements in the `switch` construct from the matched label to the end of the `switch` would be processed.

The `case` labels in a `switch` statement must be integer type (including characters) constants or constant expressions (expressions containing only constants). A variable cannot be used for a label. Similarly, the expression in the parenthesis should be one with an integer value.

Consider the following example. It prints a happy birthday message once the user presses `b` on the keyboard, a Christmas greeting if the user presses `c` on the keyboard and a hello message once the user presses `h` on the keyboard.

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char ch;
   printf("Enter a character : ");
   ch = getche();
   printf("\n");
   switch(ch)
   {
      case 'b':
         printf("Happy Birthday\n");
         break;
      case 'c':
         printf("Merry Christmas\n");
         break;
      case 'h':
         printf("Hello how do you do\n");
         break;
      default:
         printf("The character entered has not been recognised\n");
   }
   printf("End of program\n");
   return(0);
}
```

Some execution sessions of this program are shown below,

```
Enter a character : g
The character entered has not been recognised
End of program

Enter a character : c
Merry Christmas
End of program

Enter a character : b
Happy Birthday
End of program
```

Several real world applications want the same response to a number of different requests. This is accomplished with the help of the `switch` statement by using `case` labels without statements. This provides the same response for a number of labels. A modified version

of the program shown above is as follows. This program prints a hello if `h` or `H` are entered, Christmas greetings if `c` or `C` are entered and happy birthday if `b` or `B` are entered.

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char ch;
   printf("Enter a character : ");
   ch = getche();
   printf("\n");
   switch(ch)
   {
      case 'B':
      case 'b':
         printf("Happy Birthday\n");
         break;
      case 'C':
      case 'c':
         printf("Merry Christmas\n");
         break;
      case 'H':
      case 'h':
         printf("Hello how do you do\n");
         break;
      default:
         printf("The character entered has not been recognised\n");
   }
   printf("End of program\n");
   return(0);
}
```

If the user presses `H`, the program jumps to `case 'H':` .As there is no statement (not even `break`), the program proceeds until it encounters a `break` statement. Thus it has to execute the `printf("Hello how do you do\n");` statement after the `case 'h':` label. Some execution cycles of this program are given below,

```
Enter a character : C
Merry Christmas
End of program

Enter a character : b
Happy Birthday
End of program

Enter a character : Z
The character entered has not been recognised
End of program

Enter a character : B
Happy Birthday
```

```
End of program

Enter a character : c
Merry Christmas
End of program
```

Consider the following program.

```
#include <stdio.h>

int main(void)
{
      int ResultValid = 1;
      float Operand1, Operand2;
      char Operator;
      float Result;
      printf("Type OPERAND1 OPERATOR OPERAND2 and then press ENTER ");
      scanf("%f %c %f",&Operand1,&Operator,&Operand2);
      switch(Operator)
      {
            case '+':
                  Result = Operand1 + Operand2;
                  break;
            case '-':
                  Result = Operand1 - Operand2;
                  break;
            case '*':
                  Result = Operand1 * Operand2;
                  break;
            case '/':
                  if (Operand2 == 0.0)
                  {
                        ResultValid = 0;
                        printf("DENOMINATOR IS A ZERO\n");
                  }
                  else
                  {
                        Result = Operand1/Operand2;
                  }
                  break;
            default:
                  printf("UNRECOGNISED OPERATOR\n");
                  ResultValid = 0;
      }
      if (ResultValid)
      {
            printf("%f %c %f = %f\n",Operand1,Operator,Operand2,Result);
      }
      printf("End of Program\n");
      return(0);
}
```

This program performs the operation of a simple arithmetic calculator. The user has to enter two operands and an operator, and depending upon the operator, the program

performs appropriate arithmetic operations on the operands. If the operator is `/`, i.e. division, then the program checks if the operand in the denominator is zero. If the operand in the denominator is zero, an error message, `DENOMINATOR IS A ZERO`, is printed and a flag, `ResultValid`, is reset which indicates that the results should not be printed. Similarly, if the user enters an operator which is not recognised, the program prints another error message, `UNRECOGNISED OPERATOR`, and resets `ResultValid`. Some execution cycles of this program are given below,

```
Type OPERAND1 OPERATOR OPERAND2 and then press ENTER 7.2 / 3
7.200000 / 3.000000 = 2.400000
End of Program

Type OPERAND1 OPERATOR OPERAND2 and then press ENTER 3.5 * 5
3.500000 * 5.000000 = 17.500000
End of Program

Type OPERAND1 OPERATOR OPERAND2 and then press ENTER 3.5 / 0.0
DENOMINATOR IS A ZERO
End of Program

Type OPERAND1 OPERATOR OPERAND2 and then press ENTER 0.01 & 55.3
UNRECOGNISED OPERATOR
End of Program
```

`switch` cannot be used if the choice is based on evaluating a floating point variable or expression. Nor can `switch` be conveniently used if a variable must fall into a certain range. For instance, to implement `if(i < 1000 && i > 2)` with `switch` would involve setting up `case` labels for each integer from 3 to 999. However, in situations where `switch` can be employed, its use results in efficient execution of the program.

4.9 The `while` Loop

`while` loops are the basic pre-entry condition loops in C. The general form of the `while` loop is shown below

```
while (expression)
      statement1;
statement2;
```

If the `expression` is true (or, more generally, nonzero), `statement1` is executed once, and then the `expression` is tested again. This cycle of test and execution is repeated until

the `expression` becomes false (or, more generally, zero). Each cycle is called an ***iteration***. The statement (i.e., `statement1`) part can be a simple statement with a terminating semicolon or it can be a compound statement enclosed in braces.

The structure is very similar to that of an `if` statement. The chief difference is that in an `if` statement, the test and (possibly) the execution are done only once; but in the `while` loop, the test and execution may be repeated several times.

Consider the following program,

```
#include <stdio.h>

int main(void)
{
	float Number;
	float Sum = 0.0;
	float Avg;
	int Count = 0;
	printf("Enter number[%d] : ",Count);
	scanf("%f",&Number);
	while(Number >= 0.0)
	{
		Sum += Number;
		Count++;
		printf("Enter number[%d] : ",Count);
		scanf("%f",&Number);
	}
	Avg = Sum/Count;
	printf ("%d numbers entered\n",Count);
	printf ("Sum = %.2f\n",Sum);
	printf ("Average = %.2f\n",Avg);
	return(0);
}
```

This program allows the user to enter floating point numbers into variable `Number`. The `while` loop continues to execute as long as the user continues to enter positive numbers because the expression `Number >= 0.0` is true. These floating point numbers are added to the variable `Sum`, which was initialised to `0.0` at declaration. The `Count` variable counts the number of floating point numbers entered by the user. As soon as the user enters a negative number, the expression `Number >= 0.0` becomes false and the loop is not executed. The control is transferred to the statement immediately after the `while` loop, i.e., `Avg = Sum/Count;`. An execution cycle of this program is shown below.

```
Enter number[0] : 5.5
Enter number[1] : 4.5
Enter number[2] : 6.0
Enter number[3] : -2.0
3 numbers entered
Sum = 16.00
Average = 5.33
```

Please note that the first prompt to enter the data is performed by the statements before the `while` loop. This data entry operation initialises the value of the variable `Number`. The user may enter the first number as a negative number, as a result the `Number >= 0.0` is false even for the first iteration of the loop. The loop is, therefore, not executed.

Consider another example,

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
      char ch;
      printf("Enter '*' to exit\n\n");
      ch = getch();
      while(ch != '*')
      {
            if((ch >= 'a') && (ch <= 'z'))
            {
                  ch -= 32;
            }
            putchar(ch);
            ch = getch();
      }
      printf("\nEnd of program");
      return(0);
}
```

This program allows the user to enter characters as long as the user does not press the character `*`. Once the user presses the character `*`, the condition `ch != '*'` becomes false and the `while` loop terminates. Within the body of the loop, the statement `if((ch >= 'a') && (ch <= 'z'))` checks if the character entered is a lowercase alphabet. All lowercase alphabets are converted to uppercase alphabets before they are printed on the display.

Variables that appear in the test expression of the `while` loops are termed as ***loop control variables***, as these determine whether the body of the loop should be executed for the next iteration or not. It is important that the value of the loop control variable changes in the body of a loop so that the test expression eventually becomes false. Otherwise the loop will never terminate. Consider the following example,

```
#include <stdio.h>

int main(void)
{
      int i = 0;
      printf ("Program to print Good Morning 5 times\n");
      while(i < 5)
      {
            printf("Good Morning\n");
      }
      return(0);
}
```

This program continuously prints the message `Good Morning` as the value of the loop control variable, `i`, is never changed within the body of the loop. The expression `i < 5` is, therefore, always true. Consider another example,

```
#include <stdio.h>

int main(void)
{
      int i = 0;
      while (i < 5)
      {
            printf("Hello World\n");
            i--;
      }
      return(0);
}
```

Here, the value of the loop control variable changes within the body of the loop, but in the wrong direction. However, this version will eventually terminate when the value of `i` drops below the most negative number the system can handle.

As mentioned earlier, if the initial value of the loop control variable is such that the test expression is false even when the `while` statement is executed the first time, the body of the loop is never executed. Consider the following example,

```
#include <stdio.h>

int main(void)
{
      int i = 10;
      while (i < 5)
      {
            printf("Hello World\n");
      }
      printf("End of loop\n");
      return(0);
}
```

The output of an execution cycle of this program is given below,

```
End of loop
```

As the condition `i < 5` is false to start with, the body of the loop is never entered. If the initial value of the loop control variable, `i`, is set less than 5, the body of the loop is executed.

4.10 The `for` Loop

`for` loops are used to perform a set of operations a fixed number of times. The basic structure of a `for` loop is shown below,

```
for(initialisation expression ; test expression ; update expression)
{
   statement1;
}
statement2;
```

The parenthesis following the keyword contains the ***loop expression***. The loop expression is divided by semicolons into three separate expressions, i.e., the `initialise expression`, the `test expression` and the `update expression`. Consider the following example,

```
#include <stdio.h>

int main(void)
{
   int i;
   printf("Program to print the first 10 integer numbers\n\n");
   for (i=0;i<10;i++)
   {
      printf("%d\n",i);
   }
   printf("End of the program\n");
   return(0);
}
```

An execution session of this program produces the following output,

```
Program to print the first 10 integer numbers

0
1
2
3
4
5
6
7
8
9
End of the program
```

The variable `i` occupies a key role in this `for` loop. In conjunction with the three parts of the loop expression, `i` is used to control the operation of the loop. Specifically, it keeps a track of how many times the loop has been executed. Thus `i` is the loop control variable.

The initialise expression is used to initialise the loop control variable. The initialisation expression is always executed once only as soon as the loop is entered. The loop control variable may be initialised to any value. In the example shown above, the expression `i=0` is the initialisation expression in `for(i=0;i<10;i++)`

The second expression is the test expression. This expression is evaluated each time the body of the loop has to be executed. If the value of this expression if true, then the body of the loop is executed. If the expression is false, the loop is terminated and the control is

passed to the statement following the loop. For example, in the above mentioned program, the expression `i<10` is the test expression in `for(i=0;i<10;i++)`. Each time the body of the loop has to be executed, this expression is evaluated. If the value of this expression is true, the body of the loop, i.e.,

```
{
   printf("%d\n",i);
}
```

is executed. When the value of this expression becomes false, the loop is terminated and the control is transferred to the `printf("End of the program\n");` statement after the loop.

The third expression is the update expression. It updates the value of the loop control variable each time the loop is executed. In this example `i++` is the update expression. The value `i` is incremented with each iteration. The value of the loop control variable can also be decreased, or changed by some other number or expression.

The statement (single or compound) following the keyword `for` and the loop expression is the body of the loop which is executed in each iteration.

In the above mentioned program, when the `for` loop is executed, the initialisation expression is evaluated first. This initialises the value of `i` to `0`. Then the test expression is evaluated and if the value of the test expression is true, the body of the loop is executed. Once the body of the loop has been executed, the update expression is evaluated and the value of `i` is incremented. Then the next iteration has to be performed, but the test expression has to be evaluated again before entering the body of the loop. If the value of the test expression is false, the loop is terminated and the control is passed to the statement immediately after the loop.

Consider the following program,

```
#include <stdio.h>

int main(void)
{
   char Alphabet;
   printf("Program to print alphabets from 'z' to 'a'\n\n");
   for (Alphabet = 'z'; Alphabet >= 'a' ; Alphabet--)
   {
      printf("%c,",Alphabet);
   }
   printf("\nI have printed all the characters\n\n");
   return(0);
}
```

This program uses a `for` loop to print lower case alphabets from `z` to `a`. It declares a variable `Alphabet` of type `char`. Then it executes the statement

`printf("Program to print alphabets from 'z' to 'a'\n\n");` which prints the message `Program to print alphabets from 'z' to 'a'` on the screen. Then the program executes the `for` loop. The initialisation expression, `Alphabet = 'z'`, is executed first. The test expression, `Alphabet >= 'a'`, is executed then. As the value of `Alphabet` is greater than `'a'`, the body of the loop is executed, i.e., the value of the variable `Alphabet` is printed. At the end of the loop, the update expression, `Alphabet--`, is executed. The program then prepares to execute the next iteration. The test expression is evaluated, and as the value of variable `Alphabet` is still greater than `'a'`, the body of the loop is executed again. This process is repeated until the value of the variable `Alphabet` does not become less than `'a'`. Once the value of the variable Alphabet is less than 'a', the loop is terminated and the statement

`printf("\nI have printed all the characters\n\n");` is executed. This prints the message `I have printed all the characters` on the display. An execution session of this program produces the following results,

```
Program to print alphabets from 'z' to 'a'

z,y,x,w,v,u,t,s,r,q,p,o,n,m,l,k,j,i,h,g,f,e,d,c,b,a,
I have printed all the characters
```

4.11 The Comma Operator

The comma operator extends the functionality of the `for` loop by allowing the programmer to include more than one initialisation or update expressions in a `for` loop specification. For example, consider the following program,

```
#include <stdio.h>

int main(void)
{
   int x;
   int y;
   for (x = 0, y = 1 ; (x+y) < 10 ; x++, y += 2)
   {
      printf("%d :: %d \n",x,y);
   }
   return(0);
}
```

This program produces the following output,

```
0 :: 1
1 :: 3
2 :: 5
```

The initialisation expression of the `for` loop has two expressions separated by a comma, i.e., `x = 0, y = 1`. Here the expression `x = 0` is evaluated first and then the expression `y = 1` is evaluated. So the variable `x` is initialised to integer value `0` and the variable `y` is initialised to integer value `1` in the beginning of the execution of the loop. As mentioned earlier, the initialisation expression is only evaluated once in the beginning of the loop, so these values are assigned only once before the first iteration of the loop.

Similarly the update expression of the `for` loop has two expressions separated by a comma, i.e., `x++, y += 2`. As mentioned above, the expression `x++` is evaluated first and then the expression `y += 2` is evaluated. Both these expressions are evaluated at the end of each iteration.

The comma operator is not restricted to `for` loops, but that is where it is most often used. In C, comma is also used as separator. Thus the commas in

`printf("%d :: %d \n",x,y);` are separators and not comma operators.

4.12 The `do - while` Loops

The `while` and the `for` loop are both pre-entry condition loops. The test condition is checked before each iteration of the loop. C also has a post entry condition loop in which the condition is checked after each iteration of the loop. This construct is known as the `do-while` loop and has the following syntax

```
do
{
   statement1;
}
while (test_expression);
statement2;
```

First `statement1` is executed and `test_expression` is evaluated. If the value of `test_expression` is nonzero (true), then the control passes back to the beginning of the `do` statement and the process repeats itself. When the value of `test_expression` is zero (false), then the control passes to `statement2`.

Consider the following example. This program allows the user to enter characters from the keyboard. If the characters entered are uppercase alphabets, these are converted to lowercase and are printed on the display. This process is repeated until the user enters period (.) .

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
   char ch;
   printf("This program allows you to enter characters till \n");
   printf("the time you press '.'. Each uppercase character is converted \n");
   printf("to a lowercase character\n");
   do
   {
      ch = getch();
      if ((ch >= 'A') && (ch <= 'Z'))
      {
            ch += 32;
      }
      putchar(ch);
   }
```

```
   while(ch != '.');
   return(0);
}
```

A `do-while` loop is executed at least once, since the test is made after the body of the loop has been executed. Alternatively, a `for` loop or a `while` loop may be executed zero times, since the test is made before execution of the body of the loop.

Consider another example.

```
#include <stdio.h>

int main(void)
{
   int Number;
   int Sum = 0;
   do
   {
      printf("Enter a positive integer number to add ");
      scanf("%d",&Number);
      if (Number >= 0)
      {
            Sum += Number;
      }
   }
   while(Number >= 0);
   printf("Sum = %d\n",Sum);
   return(0);
}
```

This program allows the user to enter integer numbers till the time the user enters a negative number. All numbers entered up to the negative number are added and the sum is displayed to the user. Following is the output of an execution cycle of this program,

```
Enter a positive integer number to add 13
Enter a positive integer number to add 9
Enter a positive integer number to add 8
Enter a positive integer number to add -1
Sum = 30
```

4.13 Choice of Loops

Pre-entry condition loops are the most widely used loops where as the post entry condition loops are about 5% of all the loop used in C programs (source : *The C Programming Language, by Kernighan & Ritchie, Englewood Cliffs : Prentice-Hall,*

1978). There are several reasons why computer scientists consider a pre-entry condition loop superior. One is the general principle that pre-entry condition loops result in safer programs as the body of the loop is never executed once it is not supposed to be executed. Secondly, its easier to read and understand programs if the loop test is found at the beginning of the loop. Finally, in many applications, it is important that the loop is skipped entirely if the test is not initially met.

If it is decided that the programmer should use a pre-entry condition loop, which one from the `for` and `while` loops should be selected. Both `for` and `while` loops can be used to perform the same operation. To make a `while` loop look like a `for` loop, preface it with an initialisation statement and include suitable update statements at the end of the body of the loop.

```
initialise;
while (test_expression)
{
   body;
   update;
}
```

is the same as

```
for (initialise; test_expression; update)
{
   body;
}
```

4.14 Nested Loops

A ***nested loop*** is a loop which is inside another loop. Consider the following program. This program prints prime numbers from the first 50 integer numbers. The `for` loop is the outer most loop and is used to count from `3` to `50`. The value of the counter (i.e. variable `Number` of the type `int`) is checked to determine if it is a prime number or not. This check is performed by dividing the value of `Number` by integer numbers from `2` till one less than the value of `Number` (i.e., *to determine if the value of* `Number` *is divisible by* `1` *or itself*). If the number is divisible by any number other than `1` or the value of `Number`, then the value

of `Number` is not a prime number. The program raises a flag `NotPrime`. This terminates the `while` loop as a true for `NotPrime` results in a false value for the expression `(! NotPrime) && (i < Number)`. Alternatively, if `NotPrime` remains false, the `while` loop is terminated once the value of variable `i` becomes equal to the value of `Number`. If the value of `Number` is a prime number, i.e., the `while` loop has ended and the flag `NotPrime` was not set to true, the value of `Number` is displayed as a prime number.

```
#include <stdio.h>

int main(void)
{
   int Number;
   int i;
   int NotPrime;
   printf("Printing prime numbers from the first 50 integers\n\n");
   printf("%d ",2);
   for (Number = 3 ; Number <= 50 ; Number++)
   {
      NotPrime = 0;
      i = 2;
      while ((! NotPrime) && (i < Number))
      {
            if ((Number % i) == 0)
            {
                  NotPrime = 1;
            }
            else
            {
                  i++;
            }
      }
      if (! NotPrime)
      {
            printf("%d ",Number);
      }
   }
   return(0);
}
```

An execution session of this program is given below,

```
Printing prime numbers from the first 50 integers

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

4.15 `break` and `continue` Statements

The `break` statement causes an exit from the innermost enclosing loop or `switch` statement. Consider the following example. This program allows the user to enter a positive integer number. The program, with the help of an infinite `while` loop (a loop that never ends), decrements the number till it becomes less than zero. When the value of the integer number becomes less than zero, the program uses a `break` statement to terminate the loop.

```
#include <stdio.h>

int main(void)
{
   int i;
   printf("Enter a positive integer ");
   scanf("%d",&i);
   while(1)       /* While True : An Infinite Loop    */
   {
      if (i < 0)
      {
            break;
      }
      else
      {
            printf("%d ",i);
            i--;
      }
   }
   return(0);
}
```

A typical execution session of this program is as follows,

```
Enter a positive integer 7
7 6 5 4 3 2 1 0
```

The `continue` statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately. This statement can only be used with the three loops but not with a `switch`. Like `break`, it interrupts the flow of a program. However, instead of terminating the whole loop, `continue` causes the rest of an iteration to be skipped and the next iteration to be started.

Consider the following example. This program allows the user to enter an integer number and then prints all even numbers up to the number entered.

```
#include <stdio.h>

int main(void)
{
   int i,max;
   printf("Enter a positive number ");
   scanf("%d",&max);
   printf("All even numbers up to %d are \n",max);
   for(i=0;i<=max;i++)
   {
      if ((i % 2) != 0)
      {
            continue;
      }
      printf("%d ",i);
   }
   return(0);
}
```

Once the user has typed in the integer number at the prompt `Enter a positive number` the value is assigned to the variable `max` of type `int`. The `for(i=0;i<=max;i++)` is then executed. The `if ((i % 2) != 0)` statement checks if the value of variable `i` is divisible by 2. If the value of the expression `(i % 2) != 0` is true, then the value of `i` is an odd integer and should not be printed. Thus `continue;` statement is executed to skip the remaining part of the loop for that iteration and begin the next iteration. Alternatively, if the value of the expression `(i % 2) != 0` is false, the `continue;` is not executed and the program proceeds with the remaining part of the loop.

A typical execution session of this program is as follows,

```
Enter a positive number 12
All even numbers up to 12 are
0 2 4 6 8 10 12
```

The use of `break` and `continue` in loops is not a good programming practice as these statements cause alterations in the execution of loops in a manner not prescribed in the test expressions with the `while` and `for` loops. This may cause the programs to be less readable and may result in bugs in the code.

Functions

5.1 Introduction

The design philosophy of C is based on using functions. The C system provides several functions to help the programmers perform several tasks. For example, `printf`, `scanf`, `putchar` and `getchar` are a part of the C system. All programmers, however, have to write the function `main`. Programs always start executing the instructions in `main`. After that, `main` can call other functions, like `printf` and `scanf` to perform the required tasks.

A ***function*** is a self-contained unit of program code designed to accomplish a particular task. A function in C plays the same role that functions, subroutines and procedures play in other languages, although the way these are implemented and invoked may differ for different languages. Some functions only cause an action to take place. For example, `printf` causes data to be printed in the display. Some functions find a value for the program to use. For example, `atan` tells the program the angle in radians corresponding to the value specified to it. In general, a function can both produce actions and provide values.

Following may be some of the reasons why programmers would want to write and use functions,

- ***Code Reusability.*** Probably the original reason functions, subroutines or procedures were invented was to avoid having to write the same code over and over again. For instance, a programmer writes an application to perform statistical analysis of a manufacturing process. This program computes averages and variances of certain parameters of the process. If the programmer implements functions to calculate averages and variances of a set of numbers, these functions can be reused in an application to statistically analyse the performance of an automobile under different operating conditions. Moreover these functions are reused several times within the same program.

- ***Program Organisation.*** This is all the early subroutines did. However, over the years it was found that using the subroutines made it easier not only to organise programs but also the complete development activity. If the complete functionality of the program could be divided into separate activities, and each activity placed in a separate subroutine, then each subroutine could be written and checked out more or less independently. Separating code into modular functions also made programs easier to design and understand.
- ***Independence.*** As this idea took hold, it became clear that there was an advantage in making subroutines as independent from the main program and from one another as possible. For instance, subroutines were invented that had their own private variables, i.e., variables that could not be accessed from the main program or the other subroutines. This meant that the programmer did not need to worry about accidentally using the same variable name in different subroutines. The variables in each subroutine were protected from inadvertent tampering by other subroutines. Thus it was easier to write large and complex programs.

5.2 Creating and Using Simple Function

The following program uses a simple function to print a header on a page on the screen.

```
#include <stdio.h>

void printHeader(void);

int main(void)
{
      printHeader();
      printf("This is the body of the page\n");
      printf("Does the header look OK to you?\n");
      return(0);
}

void printHeader(void)
{
      int i;
      for(i=0;i<80;i++)
      {
            printf("=");
      }
      printf("\nCRACKER'S COMPUTER Co.\n");
      for(i=0;i<80;i++)
      {
            printf("=");
```

```
        }
        printf("\n");
}
```

This program produces the following output on the display,

```
================================================================================
CRACKER'S COMPUTER Co.
================================================================================
This is the body of the page
Does the header look OK to you?
```

This program looks almost like two small programs, but actually each of these programs is a function. As mentioned earlier, `main` is one function and `printHeader` is the second function. The only special thing about `main` is that it is executed first. It does not even matter if `main` is the first function in the listing.

In this example, main ***calls*** the function `printHeader`. ***Calls*** means *to cause to be executed*. Please note that there are three program elements involved in using a function, namely, function ***prototype***, function ***call*** and function ***definition***. These are explained in the following paragraphs,

- ***Function Definition***. The function itself, i.e., the code that describes what a function does is called a function definition. A function definition has the following general form,

  ```
  return_type function_name(parameter_list)
  {
      declarations
      statements
  }
  ```

  In the above mentioned example, the line,
  `void printHeader(void)`
  is the ***declarator***. The first `void` (i.e., `return_type`) means that the function does not return anything. The second `void` (i.e., the `parameter_list`) means that it does not take any arguments. Note that the declarator does not end with a semicolon. It is not a program statement whose execution causes something to happen. Rather, it tells the compiler that a function is being defined.

The function definition continues with the ***body*** of the function, i.e., the program statements that perform the task of the function. The body of the function is enclosed in braces. Following is the body of the function `printHeader`.

```
{
    int i;
    for(i=0;i<80;i++)
    {
          printf("=");
    }
    printf("\nCRACKER'S COMPUTER Co.\n");
    for(i=0;i<80;i++)
    {
          printf("=");
    }
    printf("\n");
}
```

- ***Function Call***. To make a function perform its designated task, it has to be called from another function. The general forms of a function call are

```
variable_name = function_name(parameter_list); /* if the return_type
                                                  & parameter_list are
                                                  not void */
function_name(parameter_list); /* if the return_type is void but
                                  parameter_list is not void */
function_name(); /* if the return_type and parameter_list both are
                    void */
variable_name = function_name(); /* if the return_type is not
                                    void */
```

If a function returns a value at the end of its execution, that value may be assigned to a variable. If the function requires some data to perform its tasks, these may be provided as function arguments. Because a function call a statement in the calling function, it ends with a semicolon. A function call causes the control of program to be transferred to the code in the definition of the called function. The called function performs its tasks by executing statements in the body of the function. Once the called function ends its execution, the control is returned to the calling program, to the statement immediately following the function call.

In the above mentioned example, the function `printHeader` does not return any value and does not require any data for its execution. Therefore the function call inside the function `main` is

```
printHeader();
```

The empty parenthesis after the function name also help to distinguish function calls from variable names. Once this statement is executed, the control is transferred to the definition of the function `printHeader`. The body of `printHeader` function is executed. Once the execution of function `printHeader` ends, the control is returned to function `main`, to the statement immediately following the function call `printHeader();`, i.e.,

```
printf("This is the body of the page\n");
```

- ***Function Prototypes***. Functions should be declared before they are used. ANSI C provides for a new function declaration syntax called the function prototype. A function prototype tells the compiler the number and type of arguments that are to be passed to the function and the type of the value that is to be returned by the function. The general form of a function prototype is

  ```
  return_type function_name(parameter_type_list);
  ```

  The `parameter_type_list` is typically a comma-separated list of types. Identifiers are optional and do not affect the prototype. For instance, the function prototype `int aFunction(float FirstVariable, char SecondVariable);` is equivalent `int aFunction(float, char);` The identifiers such as `FirstVariable` and `SecondVariable` that occur in the parameter type lists in function prototypes are not used by the compiler. Their purpose is to provide documentation to the programmer and other readers of the code. The keyword `int` before the function name, `aFunction`, indicates that the function returns an integer value after its execution ends. The key thing to remember about the prototype is that the data type of the return value must agree with that of the declarator in the function definition and the number of arguments and their data types must agree with those in the function definition.

In the above mentioned example, the prototype of the function printHeader is

```
void printHeader(void);
```

Here the parameter type list is `void`, indicating that the function is not to be passed any data by the calling function. Moreover, the return type is also `void`, again indicating that the function does not return any data back to the calling function. The function prototype is written before the function `main`. This causes the prototype to be visible to all the functions in the program.

5.3 Local Variables

The variables declared inside a function are visible only to the code inside that particular function and not to any other function. These variables are known as ***local variables*** or ***automatic variables*** because they are automatically created when a function is called and destroyed when the function returns.

In the above mentioned example, the variable `i` in function `printHeader` is visible to that function only and not to function `main`.

5.4 Passing Data into a Function

Several functions need to be provided information which they require for their operation. For instance, the `sqrt` function requires a double precision floating point value be passed to it. The code inside the function then calculates the square root of the value passed to it and returns the result of the square root operation to the calling program.

The example developed in the last section to draw a page header is an extremely inflexible program as it can only generate headers of one fixed design for an 80 column printer. If the size of the paper varies or the user requires a separate design, the program needs to be modified and recompiled and the modified version has to be supplied to the user. Following is a modified version of the program. This program allows the users to print headers of variable widths.

```
#include <stdio.h>

void printHeaderLine(int);
void printHeader(int);

int main(void)
{
      int Width;
      printf("Enter the width of the header ");
      scanf("%d",&Width);
      printHeader(Width);
      printf("This is the body of the page\n");
      printf("Does the header look OK to you?\n");
      return(0);
}

void printHeaderLine(int W)
{
      int i;
      for(i=0;i<W;i++)
      {
            printf("=");
      }
      printf("\n");
}

void printHeader(int Wid)
{
      printHeaderLine(Wid);
      printf("CRACKER'S COMPUTER Co.\n");
      printHeaderLine(Wid);
}
```

This program differs from the previous example in two ways. Firstly, `printHeader` function calls the `printHeaderLine` function twice to draw the header lines above and below the message `CRACKER'S COMPUTER Co.` Secondly, both these functions are passed a value corresponding to the user's choice of the width of the header.

In the definition for the function `printHeader`, the line `void printHeader(int Wid)` specifies that the function uses an argument `Wid` of type `int`. Such arguments are known as ***formal arguments***. The compiler uses this information to create a new variable named `Wid` of the type `int` which is local to the function `printHeader`. Similarly, in the definition for the function `printHeaderLine`, the line `void printHeaderLine(int W)` specifies that the function uses an argument `W` of the type `int`. As with function

`printHeader`, the compiler uses this information to create a new variable named `W` of type `int` which is local to the function `printHeaderLine`.

In function `main`, the statement `printHeader(Width);` is a call to the function `printHeader`. The value of the variable `Width`, inside the parenthesis, is assigned to the variable `Wid` declared in the function definition. The argument in the function call (`Width` in this case) is a particular value that is assigned to the variable that is the formal argument in the function definition (`Wid` in this case). Such arguments are known as ***actual arguments***. Actual arguments can be variables, constant values, function calls or expressions that provide a value of the required type.

Once this program is executed, the operating system invokes function `main`. Variable `Width` of type `int` is declared. The statement
`printf("Enter the width of the header ");` prints the prompt for the user to enter data. The statement `scanf("%d",&Width);` allows the user to enter an integer number from the keyboard. The value entered is stored in the variable `Width`. The function `printHeader` is then called and `Width` is specified in the parenthesis as an actual argument. Once the function `printHeader` is invoked, a new variable `Wid` is declared and the value of `Width` is stored in the formal argument `Wid`.

The first statement of the `printHeader` function is a function call to the function `printHeaderLine`. In this function call, the variable `Wid` is specified as the actual argument. Once the function `printHeaderLine` is invoked, a new variable `W` is declared and the value of `Wid` (which is equal to the value of `Width` declared in function main) is stored in the formal argument `W`. Inside the function `printHeaderLine`, the loop
`for(i=0;i<W;i++)` prints the symbol `=` for number of times equal to the value of `W`.

Once the function `printHeaderLine` terminates, the control is transferred back to the calling program, i.e., `printHeader` function. The next statement, i.e.,
`printf("CRACKER'S COMPUTER Co.\n");` is executed. This prints the message

`CRACKER'S COMPUTER Co.` on the display. The next statement is again a call to the `printHeaderLine` function. Finally, when the function `printHeader` terminates execution, the control is transferred to the `main` function where subsequent statements are executed.

The outputs of two execution sessions of this program are given below,

```
Enter the width of the header 30
==============================
CRACKER’S COMPUTER Co.
==============================
This is the body of the page
Does the header look OK to you?
```

```
Enter the width of the header 70
======================================================================
CRACKER’S COMPUTER Co.
======================================================================
This is the body of the page
Does the header look OK to you?
```

Therefore, by passing the information related to the width of the header into the function that print the header, the size of the header can be varied. However, the basic design remains the same and the user cannot vary either the title or the symbol used in the lines above and below the title.

The following program is another modified version of the initial example. This program not only allows printing headers of variable widths but the user can also specify the symbol to be used for the lines above and below the title.

```
#include <stdio.h>

void printHeaderLine(int,char);
void printHeader(int,char);

int main(void)
{
      int Width;
      char Symbol;
      printf("Enter the width of the header ");
      scanf("%d",&Width);
      getchar();
      printf("Enter the symbol to be used for lines ");
```

```
      scanf("%c",&Symbol);
      printHeader(Width,Symbol);
      printf("This is the body of the page\n");
      printf("Does the header look OK to you?\n");
      return(0);
}

void printHeaderLine(int W,char S)
{
      int i;
      for(i=0;i<W;i++)
      {
            putchar(S);
      }
      printf("\n");
}

void printHeader(int Wid,char Sym)
{
      printHeaderLine(Wid,Sym);
      printf("CRACKER'S COMPUTER Co.\n");
      printHeaderLine(Wid,Sym);
}
```

The function prototypes

```
void printHeaderLine(int,char);
void printHeader(int,char);
```

specify that the function `printHeader` and `printHeaderLine` require two arguments, first of which is an integer value whereas the second one is a character value. The function call `printHeader(Width,Symbol);` in function `main` also take two actual arguments, i.e., `Width` and `Symbol`. `Width` is a variable of type `int` and `Symbol` is of type `char`. The declarator `void printHeader(int Wid,char Sym)` in the function definition also specifies two formal arguments, the first one being of type `int` and the second one being of the type `char`. So when the function `printHeader` is called, the value of the first actual argument is copied into the first formal argument. Similarly, the value of the second actual argument is copied into the second formal argument.

Typical execution sessions of this program are as follows,

```
Enter the width of the header 40
Enter the symbol to be used for lines *
****************************************
CRACKER'S COMPUTER Co.
****************************************
This is the body of the page
```

```
Does the header look OK to you?


Enter the width of the header 65
Enter the symbol to be used for lines ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CRACKER'S COMPUTER Co.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This is the body of the page
Does the header look OK to you?
```

5.5 Returning a Value from a Function

Functions can also return the results of processing, they are required to perform, to the calling function. The value is returned to the calling function by placing the value or the variable name between the parenthesis following the `return` keyword. The `return` statement has two purposes. First, executing it immediately transfers control from the called function back to the calling function. Second, whatever is inside the parenthesis following `return` keyword is returned as a value to the calling program.

Consider the following program. It uses a function `getSign` which takes in a floating point number and returns an integer number. The value of the integer number is `1` if the floating point number passed to the function was a negative number. Otherwise it is `0`.

```
#include <stdio.h>
#include <math.h>

int getSign(float);

int main(void)
{
      float InNumber;
      int SignOfNumber;
      printf("Enter a floating point number ");
      scanf("%f",&InNumber);
      SignOfNumber = getSign(InNumber);
      if (SignOfNumber)
      {
            printf("%f is a negative number\n",InNumber);
      }
      else
      {
            printf("%f is a positive number\n",InNumber);
      }
      return(0);
}

int getSign(float ANumber)
{
```

```
	float AbsValue;
	int SignOfNumber;
	AbsValue = fabs(ANumber);
	if (fabs(AbsValue - ANumber) <= 1e-6)
	{
		SignOfNumber = 0;
	}
	else
	{
		SignOfNumber = 1;
	}
	return(SignOfNumber);
}
```

The function prototype `int getSign(float);` suggests that the `getSign` function takes in a floating point value and returns an integer. The `main` function declares two variables, `InNumber` and `SignOfNumber` of types `float` and `int` respectively. The statement
`printf("Enter a floating point number ");`
prompts the user to enter a floating point number by printing the statement
`Enter a floating point number` . The statement `scanf("%f",&InNumber);` accepts the data from the keyboard and assigns it to the variable `InNumber`.

The statement `SignOfNumber = getSign(InNumber);` is a function call to the function `getSign`. The actual argument to the function is the value of the variable `InNumber`. As the function is expected to return a value of type `int`, it must be assigned to an appropriate variable or otherwise it will be lost. In this statement, the value returned is assigned to the variable `SignOfNumber`.

When the function `getSign` is invoked the value of the actual argument is copied to the formal argument `ANumber`. The statement `AbsValue = fabs(ANumber);` determines the absolute value of the variable `ANumber` which is assigned to the variable `AbsValue`. Then the following `if-else` statement is executed

```
	if (fabs(AbsValue - ANumber) <= 1e-6)
	{
		SignOfNumber = 0;
	}
	else
	{
		SignOfNumber = 1;
	}
```

The expression `fabs(AbsValue - ANumber)` calculates the absolute value of the difference between `AbsValue` and `ANumber`. If the absolute value of the difference is less than or equal to `1e-6` then the statement `SignOfNumber = 0;` is executed. Alternatively, the statement `SignOfNumber = 1;` is executed. Please note that the variable `SignOfNumber` in function `getSign` is local to function `getSign`.

In the end, the statement `return(SignOfNumber);` is executed which returns the value of `SignOfNumber` to the calling function, which in this case is function `main`. Additionally the control of program is also transferred to function `main`, which executes the statement immediately after the function call. The statement after the function call is an `if-else` statement. The statement checks the value of `SignOfNumber` and if the value is non-zero, the following statement is executed,

`printf("%f is a negative number\n",InNumber);`. Alternatively, the following statement is executed,

```
printf("%f is a positive number\n",InNumber);
```

Following are a few sessions of the program

```
Enter a floating point number 53.523
53.52300 is a positive number

Enter a floating point number -10.215
-10.21500 is a negative number

Enter a floating point number 77
77.00001 is a positive number
```

5.6 Finding Addresses : The & Operator

The `&` operator provides the address at which a variable is stored. If `IntValue` is the name of the variable, then `&IntValue` is the address of the variable. This address is the location in memory where the value assigned to the variable is stored. So if `IntValue = 24` and the address where `IntValue` is stored is `12126`, then

```
printf (“%d :: %u\n”,IntValue, &IntValue);
```

will produce

```
24 :: 12126
```

5.7 Altering Variables in the Calling Program

Consider a situation where one function is to be used to make changes in the variables of a different function. For example, a function is used to swap the values of two variables `x, y`. If the following program is used

```
/*    noptr.c     Demo of pointers  */

#include <stdio.h>

void interchange(int a,int b);

int main(void)=
{
      int x = 10;
      int y = 12;
      printf("Original :: x = %d :: y = %d\n",x,y);
      interchange(x,y);
      printf("New      :: x = %d :: y = %d\n",x,y);
      return(0);
}

void interchange(int a,int b)
{
      int temp;
      printf("Original :: a = %d :: b = %d\n",a,b);
      temp = a;
      a = b;
      b = temp;
      printf("New      :: a = %d :: b = %d\n",a,b);

}
```

The output of this program will be

```
Original :: x = 10 :: y = 12
Original :: a = 10 :: b = 12
New      :: a = 12 :: b = 10
New      :: x = 10 :: y = 12
```

So these values did not get swapped in function main even when swapping did occur inside the function `interchange`. This problem is solved if two values could be passed back, instead of one, from the `interchange` function. This is not possible using the `return` statement. To return more than one value from a function, pointers are used.

5.8 Pointers

A pointer is a symbolic representation of an address. So the address operator (`&`) with the name of the variable is known as the ***pointer*** to that variable. Thus `&x` is the ***pointer to*** `x`. The actual address is a number and pointer to the variable is a ***pointer constant*** (as the variable does not change its memory location as the program executes). If `ptr` is the name of a pointer variable, the statement `ptr = &IntValue` assigns `IntValue`'s address to `ptr`. Note that `ptr` is a variable and `&IntValue` is a constant.

5.9 The Indirection Operator : *

If there is a particular pointer, `ptr`, to a variable, the value stored in the memory location being pointed at by `ptr` can be found by using the ***indirection operator*** (`*`).

```
val = *ptr;
```

Thus if `ptr` is a pointer to `IntValue` and `IntValue = 27`, then after executing this statement `val = 27`.

5.10 Declaring Pointers

Declaration of a pointer variable is associated with the type of data stored in that specific memory location. This is because different variable types have different storage requirements and some pointer operations require knowledge of the storage size of the data type. Also, the program may need to know what kind of data is stored at that address. For example, a `long` and a `float` may use the same amount of storage but they store the numbers quite differently. Pointer variables are, therefore, declared as follows;

```
int *pi;          /*pointer to an integer*/
char *pc;         /*pointer to a char variable*/
float *pf;        /*pointer to a floating point variable*/
```

The type specification identifies the type of the variable pointed to and the asterisk identifies the variable itself as a pointer. So `int *pi` states that `pi` is a pointer and that `*pi` is of type `int`.

5.11 Using Pointers to Communicate Between Functions

The following example is a modified version of the previous example and uses pointers

```
/*    ptr1.c      Demo of pointers  */

#include <stdio.h>

void interchange(int *a,int *b);

int main(void)
{
      int x = 10;
      int y = 12;
      printf("Original :: x = %d :: y = %d\n",x,y);
      interchange(&x,&y);
      printf("New      :: x = %d :: y = %d\n",x,y);
      return(0);
}

void interchange(int *a,int *b)
{
      int temp;
      printf("Original :: a = %d :: b = %d\n",*a,*b);
      temp = *a;
      *a = *b;
      *b = temp;
      printf("New      :: a = %d :: b = %d\n",*a,*b);

}
```

Now the output of this program is

```
Original :: x = 10 :: y = 12
Original :: a = 10 :: b = 12
New      :: a = 12 :: b = 10
New      :: x = 12 :: y = 10
```

This program managed to function as desired because the function call `interchange(&x,&y)` transmitted the addresses of `x` and `y` instead of their values. Thus `a` and `b` in `interchange(int *a, int *b)` are actually addresses of `x` and `y`. Therefore `a` and `b` are pointers to `x` and `y`.

Now as a==&x and b==&y, *a==x and *b==y, so temp = *a; actually assigns temp the value of x. Similarly, *a=*b; actually assigns x the value of y and *b=temp; assigns y the value of temp which was the original value of x.

5.12 Example

The following example uses a function, calculate, to perform the addition, subtraction, multiplication and division operations on the values assigned to two variables.

```
/*    calc.c      Demo of pointers  */

#include <stdio.h>

void calculate(float x,float y,float *sum,float *diff,float *product,
          float *division);

int main(void)
{
      float x, y, sum, difference, product, division;
      x = 7.5;
      y = 0.5;
      calculate(x,y,&sum,&difference,&product,&division);
      printf("%5.2f + %5.2f = %5.2f\n",x,y,sum);
      printf("%5.2f - %5.2f = %5.2f\n",x,y,difference);
      printf("%5.2f * %5.2f = %5.2f\n",x,y,product);
      printf("%5.2f / %5.2f = %5.2f\n",x,y,division);
      return(0);
}

void calculate(float x,float y,float *sum,float *diff,float *product,
               float *division)
{
      *sum = x + y;
      *diff = x - y;
      *product = x * y;
      *division = x / y;
}
```

5.13 Storage Classes

In addition to having a type, each variable has a storage class. There are four keywords used to describe storage classes; extern (for external), auto (for automatic), static and register. The storage class of a variable is determined by where it is defined and by what keyword. The storage class determines two things. First, it controls which functions have access to a variable. The extent to which a variable is available is called its ***scope***. Second, the storage class determines how long the variable persists in memory.

Following paragraphs describe the properties of each class

5.13.1 Automatic Variables

By default, variables declared in a function are automatic. One can, however, explicitly declare the variables as being automatic by using the keyword `auto`. Consider the following example,

```
void AFunction(void)
{
   auto int AVar = 0;
   AVar +=3;
   printf("In Function AVar = %d\n",AVar);
}
```

Here, the variable `AVar` has been explicitly declared as an automatic variable of type `int`. Such declarations are generally used by programmers to show that they have intentionally overridden an external definition.

An automatic variable has local scope within a block, i.e., the pair of braces in which the variable is declared. It is considered a good programming practice to declare automatic variables at the beginning of a function definition. Only the function in which the variable is defined knows the variable. Of course, arguments can be used to communicate the value and the address of the variable to another function, but that is partial and indirect knowledge. Other functions can use variables with the same name, but they will be independent variables stored in different memory locations.

An automatic variable comes into existence when the function that contains it is called. When the function finishes its task and returns control to its caller, the automatic variable disappears. The memory location can now be used for something else. Thus, the values of these variables are also lost. If the function is invoked again, the system once again allocates memory, but the previous values are unknown.

5.13.2 External Variables

A variable defined outside a function is external. An external variable can also be declared in a function that uses it with the `extern` keyword. For instance, consider the following example,

```
#include <stdio.h>

int GlobalVar;

void change(void);

int main(void)
{
   extern int GlobalVar;
   GlobalVar = 0;
   printf("Before function GlobalVar = %d\n",GlobalVar);
   change();
   printf("After function GlobalVar = %d\n",GlobalVar);
   return(0);
}

void change(void)
{
   extern int GlobalVar;
   printf("In function, before change GlobalVar = %d\n",GlobalVar);
   GlobalVar += 3;
   printf("In function, after change GlobalVar = %d\n",GlobalVar);
}
```

Here, the declarations `extern int GlobalVar;` in functions `main` and `change` indicate that the variable `GlobalVar` being used in these functions has been declared elsewhere, i.e., outside these functions. Please note that both these functions use the same variable `GlobalVar`. Thus this variable is considered to be global to all functions declared after it, and upon exit from the block or function, the external variable remains in existence. An execution session of this program produces the following output,

```
Before function GlobalVar = 0
In function, before change GlobalVar = 0
In function, after change GlobalVar = 3
After function GlobalVar = 3
```

Program code can be divided into different source code files which are compiled separately. Thus, if an external variable declared in one source code file is required in another source code file, it needs to be declared with the `extern` keyword. Thus use

of this keyword tells the compiler that the variable has been defined elsewhere, either in the same source code file or a different source code file. However, the complete declaration with the extern keyword may be omitted entirely if the original definitions occur in the same file and before the function that uses them. For instance, consider another version of the program shown above.

```
#include <stdio.h>

int GlobalVar;

void change(void);

int main(void)
{
   GlobalVar = 0;
   printf("Before function GlobalVar = %d\n",GlobalVar);
   change();
   printf("After function GlobalVar = %d\n",GlobalVar);
   return(0);
}

void change(void)
{
   printf("In function, before change GlobalVar = %d\n",GlobalVar);
   GlobalVar += 3;
   printf("In function, after change GlobalVar = %d\n",GlobalVar);
}
```

Unlike the program mentioned above, this program does not contain the declarations `extern int GlobalVar;` in the functions `change` and `main`. However, as the variable `GlobalVar` is originally declared in the same source code file outside all the functions, it is by default visible to all the functions defined after its declaration. Hence the output of an execution session of this program is the same as that for the previous program.

However, if just the `extern` keyword would have been omitted from the declaration in the function, then a separate automatic variable is setup by that name for that function. For instance consider the following program,

```
#include <stdio.h>

int GlobalVar;
```

```
void change(void);

int main(void)
{
   GlobalVar = 0;
   printf("Before function GlobalVar = %d\n",GlobalVar);
   change();
   printf("After function GlobalVar = %d\n",GlobalVar);
   return(0);
}

void change(void)
{
   int GlobalVar = 1;
   printf("In function, before change GlobalVar = %d\n",GlobalVar);
   GlobalVar += 3;
   printf("In function, after change GlobalVar = %d\n",GlobalVar);
}
```

In this program, the statement `int GlobalVar = 1;` inside function `change` declares a local variable with the same name as that of the external variable, i.e., `GlobalVar` and initialises it to the value `1`. Thus all operations on `GlobalVar` inside function `change` affect only its local variable `GlobalVar` and not the global variable `GlobalVar`. An execution session of the above mentioned program produces the following output.

```
Before function GlobalVar = 0
In function, before change GlobalVar = 1
In function, after change GlobalVar = 4
After function GlobalVar = 0
```

External variables never disappear. Because they exist throughout the execution life of the program, they can be used to transmit values across functions. However, passing information to functions by the use of the parameter mechanism is a preferred method. It tends to improve the modularity of the code and it reduces the possibility of undesirable side effects.

One form of side effect occurs when a function changes a global variable from within its body rather than through its parameter list. Such a construction is error prone. Correct practice is to effect changes to global variables through the parameter and

`return` mechanisms, or by the use of pointers. Adhering to this practice improves modularity and readability, and because changes are localised, programs are typically easier to write and maintain.

5.13.3 Static Variables

static and volatile variables

Static declarations allow a local variable to retain its previous value when the block is reentered. This is in contrast to ordinary automatic variables which lose their values upon block exit and have to be reinitialised. Consider the following program,

```
#include <stdio.h>

void check(void);

int main(void)
{
   printf("First call\n");
   check();
   printf("Second call\n");
   check();
   printf("Third call\n");
   check();
   return(0);
}

void check(void)
{
   int Volatile = 0;
   static int NonVolatile = 0;
   Volatile++;
   NonVolatile++;
   printf("Volatile    = %d\n",Volatile);
   printf("NonVolatile = %d\n",NonVolatile);
}
```

When the function `check` is invoked for the first time, the variable `NonVolatile` is initialised to `0` and the variable `Volatile` is also initialised to `0`. On function exit, the variable `Volatile` is destroyed whereas the value of variable `NonVolatile` is preserved in memory. Whenever the function is invoked again the value of `NonVolatile` is not reinitialised whereas the value of `Volatile` is reinitialised. `NonVolatile` retains its previous value from the last time the function was called. The declaration of this `NonVolatile` as `static int` inside the body of `check` keeps it

private to `check`. If it were declared outside `check`, other functions could access it too.

An execution session of this program produces the following output,

```
First call
Volatile    = 1
NonVolatile = 1
Second call
Volatile    = 1
NonVolatile = 2
Third call
Volatile    = 1
NonVolatile = 3
```

5.13.4 Register Variables

Variables are normally stored in computer memory. It may be possible to store some variables in CPU registers where they can be accessed and manipulated more rapidly than in memory. In other respects, register variables are the same as automatic variables. A register variable may be declared as shown in the following statement

```
register int RegVar;
```

Declaring a variable as register class is more a request than a direct command. The compiler has to weigh the request against the number of registers that are available. If enough registers are not available, the variable is created as an ordinary automatic variable.

5.13.5 External Static Variables

A static variable can be declared outside any function. This creates an external static variable. The difference between an ordinary external variable and external static variable lies in the scope. The ordinary external variable can be used by functions in any source code file, while the external static variable can be used only by functions in the same file and below the variable declaration.

5.14 Modular Programming

Modular programming allows the programmers to increase their productivity through improved planning at early stages of system development. It aims at decomposing the entire software into small modules, each of which performs a single limited function. Ideally modules can be designed, implemented and tested independently from other modules.

A modular program consists of a ***main module***. This main module invokes other sub-modules appropriately to achieve the overall objective of the system (Figure 5-1). As a module is invoked , the control is transferred from the main module to the invoked sub-module. This sub-module performs its task and returns control to the main module upon completion of the task. If the task assigned to a sub-module is too complex, it must be broken down into other sub-sub-modules (Figure 5-2). The sub-module then acts as a main module for the sub-sub-modules and invokes them appropriately to perform the desired tasks.

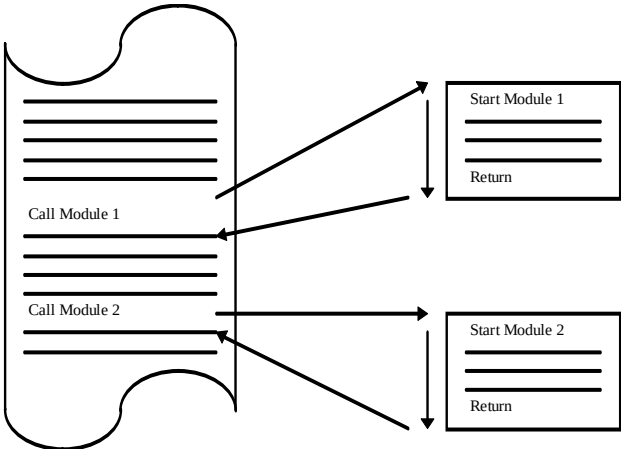


Figure 5-1 : Main Module and Sub-Module Interaction in a Modular Program

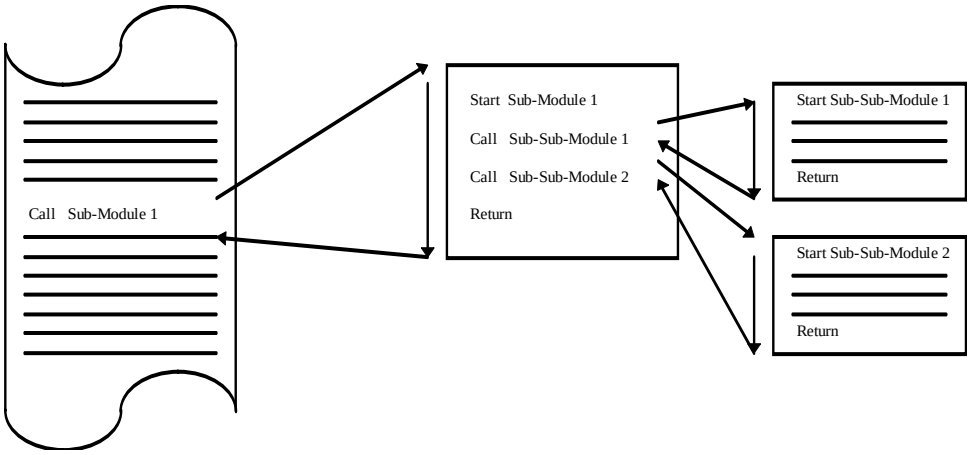


Figure 5-2 : Decomposition of a Sub-Module

Significant advantages can be achieved by developing modular software. Several modules, for instance mathematical functions, used in a program may be of use in other programs. In such cases, these functions and modules are implemented as libraries of code. A program that requires functions from a specific library need only call the required function and the appropriate library need only be linked to the program at compile time. Thus modular programs enhance ***software reusability*** (Figure 5-3). Moreover a function required at different places within the program need only be called at those points.

Modular program development facilitates employment of a team of programmers and software engineers on a big project. In such cases, once the initial design has been prepared, different programmers will be tasked to develop different parts of the project. As an ideal software module can be designed, implemented and tested independently of other parts of the main system, modular development can enhance the efficiency of the development team significantly. Additionally modular programming allows incremental development of the system. For instance, a skeleton program (or the initial mock-up) of the system can be developed quite quickly to demonstrate how the completed system would operate. This is usually accomplished by writing trace messages within the modules. Once the designers are satisfied with the functionality of the system at that level of design, trace messages within these modules are replaced with actual code.

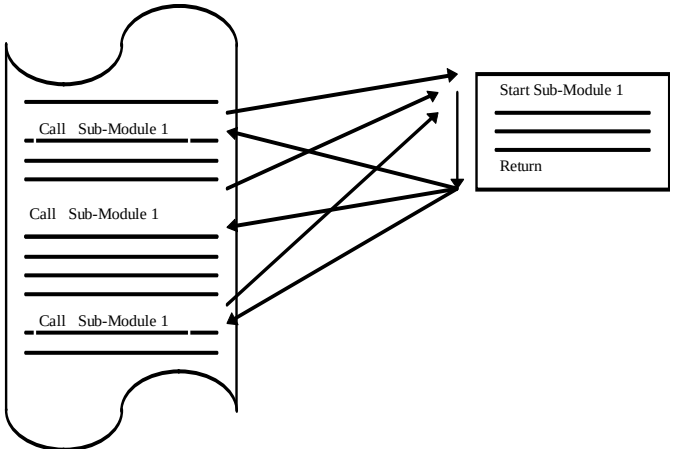


Figure 5-3 : Software Reusability

Arrays and Strings

6.1 Introduction

Consider a program that allows the user to enter the temperature for each day of a week and then prints the average temperature of that week.

```
#include <stdio.h>

int main(void)
{
  float SunTemp;
  float MonTemp;
  float TueTemp;
  float WedTemp;
  float ThuTemp;
  float FriTemp;
  float SatTemp;

  float Average;

  printf("Enter temperature for each day of the week of the day ");
  printf("separated by space :: ");
  scanf("%f %f %f %f %f %f %f",&SunTemp,&MonTemp,&TueTemp,&WedTemp,
                               &ThuTemp,&FriTemp,&SatTemp);
  getchar();
  Average=
  (SunTemp+MonTemp+TueTemp+WedTemp+ThuTemp+FriTemp+SatTemp)/7.0;
  printf("Average temperature for the week :: %7.2f",Average);
  return(0);
}
```

This program uses a collection of similar data elements (of data type `float`) where each data element has been given a unique variable name. If the number of data elements increase (e.g., to modify the program to provide the average temperature of a specific month), the programming effort increases significantly as the programmer has to declare individual data elements to store the temperature value for each day of the month.

If a collection of homogeneous values (i.e., of the same type) are to be used in a program, these are usually stored in ***arrays***. More specifically, an array is a series of variables that share the same basic name and are distinguished from one another by a numerical tag (also known as the array ***index***). To access or manipulate a specific variable in the array, the programmer has to specify the name of the array and the index value representing the

position of the variable in the array. An interesting aspect of using arrays is that the index can also be an expression (i.e., a variable, constant or a combination of variables, constants and operators) with an integer value. This makes it convenient for the programmer to write powerful array access routines.

6.2 Defining One Dimensional Arrays

An array is a collection of variables of a certain type, placed contiguously in memory. Like other variables, the array needs to be defined, so the compiler knows what is the type of the array elements and how many elements are contained in that array. A declaration for an array is given below,

```
float Temperature[7];
```

Here, the `float` specifies the type of the variables in the array, just as it does with simple variables, and the word `Temperature` is the name of the array. In `[7],` the number tells how many variables of type `float` are contained in this array. Each of the separate variables in the array is called an ***element***. The brackets tell the compiler that the variable being declared is actually an array. Figure 6-1 shows the array.

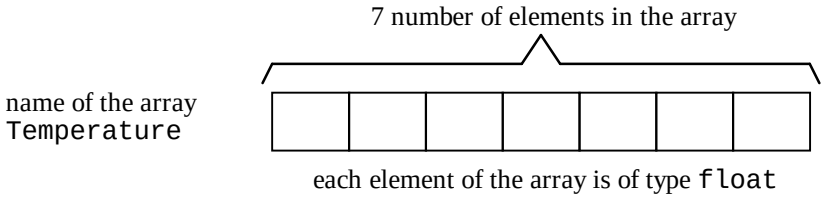
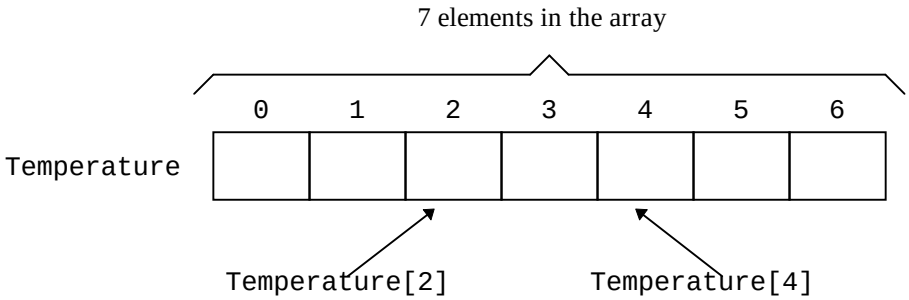


Figure 6-1 : Representation of `float Temperature[7];`

6.3 Using Arrays in Programs

Once the array has been declared, its individual elements may be accessed with subscripts or indexes. These are the numbers in brackets following the array name. Note, however, that this number has a different meaning when referring to an array element than it does when defining the array, when the number in brackets is the size of the array. When referring to an array element, this number specifies the element's position in the array. All

the array elements are numbered or indexed starting at 0. The element of the array `Temperture` with the index 2 would be referred to as `Temperature[2]`. Note that, because the numbering starts with 0, this is not the second element of the array, but the third. Thus, the index of the last array element is one less than the size of the array. This is shown in figure 6-2.



- The subscript of the first element of the array is 0
- The subscript of the last element of the array is one less than the array size

Figure 6-2 : Referencing Individual Array Elements

As mentioned earlier, the array subscript can be an integer constant, variable or an expression with an integral value.

The following program uses an array to store the temperature of every day of the week entered by the user. Then it calculates the average temperature for the week and prints it for the user.

```
#include <stdio.h>

int main(void)
{
      float Temperature[7];
      float Average;
      float Sum = 0.0;
      int DayOfWeek;
      for (DayOfWeek=0;DayOfWeek<7;DayOfWeek++)
      {
            printf("Enter temperature for day[%d] :: ",DayOfWeek);
            scanf("%f",&Temperature[DayOfWeek]);
      }
      for (DayOfWeek=0;DayOfWeek<7;DayOfWeek++)
      {
```

```
            Sum += Temperature[DayOfWeek];
      }
      Average = Sum/7.0;
      printf("Average Temperature :: %7.2f\n",Average);
      return(0);
}
```

In this program, the following section of the code reads data from the keyboard and places it in the keyboard buffer.

```
      for (DayOfWeek=0;DayOfWeek<7;DayOfWeek++)
      {
            printf("Enter temperature for day[%d] :: ",DayOfWeek);
            scanf("%f",&Temperature[DayOfWeek]);
      }
```

The `for` loop causes the process of asking for and receiving a temperature value from the user to be repeated seven times. The first time through the loop, the `DayOfWeek` variable has the value `0` and `scanf` statement causes the value typed in to be stored in array element `Temperature[0]` which is the first element of the array. This process is repeated till the value of `DayOfWeek` becomes more than `6`.

To put data into a specific array element, the `scanf` function require the address of that particular array element. This is why the `scanf` function has a pointer `&Temperature[DayOfWeek]` as an argument to the function. Here, `Temperature[DayOfWeek]` is a specific array element, depending upon the value of the variable `DayOfWeek`, and `&Temperature[DayOfWeek]` is its address.

To calculate average, the program needs to access each element of the array successively and add its value to the value of a variable representing the running total (i.e., the variable `Sum`). This is accomplished by the following program segment,

```
      for (DayOfWeek=0;DayOfWeek<7;DayOfWeek++)
      {
            Sum += Temperature[DayOfWeek];
      }
```

The statement `Average = Sum/7.0;` then calculates the average of the values entered by the user. A typical execution session of this program is given below,

```
Enter temperature for day[0] :: 12.35
Enter temperature for day[1] :: 11.15
Enter temperature for day[2] :: 12.03
Enter temperature for day[3] :: 11.20
Enter temperature for day[4] :: 11.55
Enter temperature for day[5] :: 11.45
Enter temperature for day[6] :: 12.10
Average Temperature :: 11.69
```

Arrays can also be used to store data when the number of data elements is not known in advance. For instance, consider a traffic monitoring system that needs to calculate the average speed of all the vehicles passing through a specific point in a day. The system should, therefore, incorporate some means to enter the speed of each vehicle passing through that point until it is signaled to stop entering data (e.g., when the system clock strikes midnight, or the user presses a key on the keyboard). The system then add all the speed values together and divides the sum by the total number of vehicles monitored to get the average speed.

Surely the designer of such a system cannot precisely determine the size of the array to store the speed value of each vehicle as the number of vehicles passing that particular point will be different for each day. It is, however, possible to declare a large array and then use only as many elements as required by the system. An example of such a program is given below,

```
#include <stdio.h>
#include <conio.h>
#define MAX 100

int main(void)
{
  float Speed[MAX];
  float Average;
  int Count = 0;
  float Sum = 0.0;
  int i;
  char More;
  do
  {
    printf("Enter speed for vehicle no. %d :: ",Count);
```

```
    scanf("%f",&Speed[Count]);
    getchar();
    Count++;
    printf("Press 'n' to exit, anyother key to enter another value ");
    More = getche();
    printf("\n");
  }
  while(More != 'n');
  for (i=0;i<Count;i++)
  {
    Sum += Speed[i];
  }
  Average = Sum/Count;
  printf("Average Speed of %d vehicles is %7.2f\n",Count,Average);
  return(0);
}
```

In this program, the statement `float Speed[MAX];` declares an array `Speed` to store floating point values and the size of the array is given by the identifier `MAX`. This identifier is given the value by the preprocessor directive `#define MAX 100`. This arrangement simplifies program modification as only the number `100` has to be changed to the required value if the size of the array needs to be modified.

The following program segment is used to enter data into the array,

```
  do
  {
    printf("Enter speed for vehicle no. %d :: ",Count);
    scanf("%f",&Speed[Count]);
    getchar();
    Count++;
    printf("Press 'n' to exit, anyother key to enter another value ");
    More = getche();
    printf("\n");
  }
  while(More != 'n');
```

The variable `Count`, which is initialised to `0`, is used an the array index, Each time a user enters a value, the variable `Count` is incremented. Thus, the value of `Count` always points to the array element into which data has to be entered. Moreover, the value of `Count`, at any instant is the total number of floating point values assigned to the respective array elements. The body of the loop is executed till the user presses `n` at the prompt

`Press 'n' to exit, anyother key to enter another value`

When the user presses `n` at the prompt mentioned above, the `do-while` loop is terminated and the following segment of the program is executed,

```
for (i=0;i<Count;i++)
{
  Sum += Speed[i];
}
```

Here, the variable `i` is used as the array index and the variable `Count` represents the total number of array elements that contain valid data. Value of each element of the array `Speed`, from `Speed[0]` to `Speed[Count-1]` is added to the variable `Sum` and the result is assigned back to the variable `Sum`. The average is then calculated by the statement

```
Average = Sum/Count;
```

An execution session of this program is given below,

```
Enter speed for vehicle no. 0 :: 55.3
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 1 :: 45.5
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 2 :: 67
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 3 :: 64
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 4 :: 53.2
Press 'n' to exit, anyother key to enter another value n
Average Speed of 5 vehicles is 57
```

Consider another execution session of this program

```
Enter speed for vehicle no. 0 :: 55.3
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 1 :: 45.5
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 2 :: 67
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 3 :: 64
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 4 :: 53.2
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 5 :: 55.3
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 6 :: 64
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 7 :: 53.2
Press 'n' to exit, anyother key to enter another value n
Average Speed of 8 vehicles is 57.19
```

6.4 Checking Array Bounds

As mentioned earlier, an array is a collection of variables of a certain type, placed contiguously in memory. Moreover, array elements are accessed with subscripts or indexes which specify the location of the elements in the array. If the value of the subscript is greater than or equal to the numerical value representing the size of the array, the program actually attempts to access memory outside the array. For instance, in a previous program, an array `Temperature` of seven floating point numbers was declared. The subscripts to be used to access elements of this array range from `0` to `6`. The expression `Temperature[7]` attempts to access memory outside the bounds of this array. This is shown in figure 6-3,

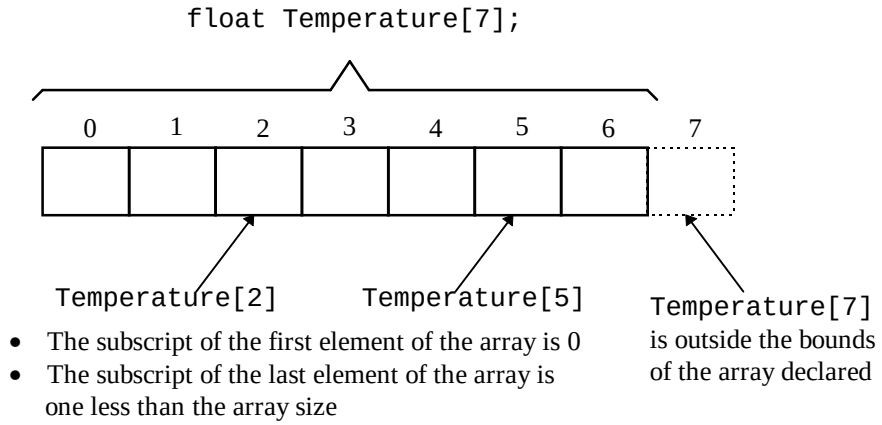


Figure 6-3 : Array Bounds

In C there is no mechanism to check and to warn if the array subscript is larger than the size of the array. Data entered with too large a subscript is placed in memory outside the array, probably overwriting other data or the program itself. This will lead to unpredictable results with no warning from the C system that the array bounds have been exceeded.

The responsibility of checking whether the program has exceeded array bounds or not lies with the programmer. All operations which involve the use of arrays must contain code

that insures that the data manipulation occurs within the array bounds. For instance, consider a modified version of the example shown above,

```
#include <stdio.h>
#include <conio.h>
#define MAX 10

int main(void)
{
  float Speed[MAX];
  float Average;
  int Count = 0;
  float Sum = 0.0;
  int i;
  char More;
  do
  {
    printf("Enter speed for vehicle no. %d :: ",Count);
    scanf("%f",&Speed[Count]);
    getchar();
    Count++;
    printf("Press 'n' to exit, anyother key to enter another value ");
    More = getche();
    printf("\n");
  }
  while((More != 'n') && (Count<MAX));
  for (i=0;i<Count;i++)
  {
    Sum += Speed[i];
  }
  Average = Sum/Count;
  printf("Average Speed of %d vehicles is %7.2f\n",Count,Average);
  return(0);
}
```

The test condition in the `do-while` loop (i.e., `(More != 'n') && (Count<MAX)`) not only checks for the value of the variable `More`, but also checks the value of the variable `Count`. As mentioned earlier, the value of variable `Count` determines the number of values entered into the array. If this value is greater than or equal to the value of the identifier `MAX`, the expression `(Count<MAX)` is false and the test expression becomes false. As a result, the body of the loop is not executed again. Thus the expression `(Count<MAX)` protects the array bounds from being crossed.

In this particular example, the value of `MAX` is defined as `10`. A typical test session in which the user attempts to input more than 10 values is shown below,

```
Enter speed for vehicle no. 0 :: 55.3
```

```
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 1 :: 45.5
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 2 :: 67
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 3 :: 64
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 4 :: 53.2
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 5 :: 55.3
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 6 :: 64
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 7 :: 53.2
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 8 :: 64
Press 'n' to exit, anyother key to enter another value y
Enter speed for vehicle no. 9 :: 53.2
Press 'n' to exit, anyother key to enter another value y
Average Speed of 10 vehicles is 57.47
```

Here, the program did not allow the user to enter data into more than 10 array elements.

6.5 Storage Classes and Array Initialisation

Arrays may be of storage class automatic, external or static but not register. In traditional C, only external and static arrays can be initialised using an array initialiser. In ANSI C, automatic arrays can also be initialised.

It should be noted that static and external arrays, if not initialised, their elements are automatically initialised to 0, whereas the automatic arrays retain whatever ***garbage values*** were already in the memory locations. For example, consider the following program,

```
#include <stdio.h>

int main(void)
{
  int AutoArray[3];
  static int StaticArray[3];
  int i;
  printf("Printing contents of an automatic array without initialisation\n");
  for (i=0;i<3;i++)
  {
    printf("%d ",AutoArray[i]);
  }
  printf("\nPrinting contents of a static array without initialisation\n");
  for (i=0;i<3;i++)
  {
    printf("%d ",StaticArray[i]);
```

```
  }
  return(0);
}
```

An execution session of this program provides the following output,

```
Printing contents of an automatic array without initialisation
2493 19802 314
Printing contents of a static array without initialisation
0 0 0
```

Consider a program that initialises an automatic array,

```
#include <stdio.h>

int main(void)
{
      int i;
      int DaysInMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
      for (i=0;i<12;i++)
      {
            printf("Month %2d has %d days\n",i+1,DaysInMonth[i]);
      }
      return(0);
}
```

The output of this program is given below,

```
Month  1 has 31 days
Month  2 has 28 days
Month  3 has 31 days
Month  4 has 30 days
Month  5 has 31 days
Month  6 has 30 days
Month  7 has 31 days
Month  8 has 31 days
Month  9 has 30 days
Month 10 has 31 days
Month 11 has 30 days
Month 12 has 31 days
```

Here, the statement

`int DaysInMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};`

declares an array, `DaysInMonth`, of 12 elements and initialises these with the integer values in the list on the other side of the assignment operator. The number of items in the list should match the size of the array.

Consider the following program, Here the list of numbers assigned to the array during initialisation contains only 10 elements, instead of 12,

```
#include <stdio.h>

int main(void)
{
      int i;
      int DaysInMonth[12] = {31,28,31,30,31,30,31,31,30,31};
      for (i=0;i<12;i++)
      {
            printf("Month %2d has %d days\n",i+1,DaysInMonth[i]);
      }
      return(0);
}
```

The output of this program is given below,

```
Month  1 has 31 days
Month  2 has 28 days
Month  3 has 31 days
Month  4 has 30 days
Month  5 has 31 days
Month  6 has 30 days
Month  7 has 31 days
Month  8 has 31 days
Month  9 has 30 days
Month 10 has 31 days
Month 11 has 0 days
Month 12 has 0 days
```

The compiler assigned each of the integer values in the list to the successive array elements. The last two elements, `DaysInMonth[10]` and `DaysInMonth[11]` are initialised to `0`.

The C compiler also has a mechanism to automatically determine the size of an array at initialisation, if not defined explicitly by the programmer. The compiler makes the array large enough to hold the number of values specified in the initialisation list. Consider the following example,

```
#include <stdio.h>

int main(void)
```

```
{
      float ShoppingBill[] = {72.25,100.25,88.80,90.0,70.5,105.5,95.75};
      int i;
      long ArraySize;
      ArraySize = sizeof(ShoppingBill)/sizeof(float);
      printf ("Shopping Bills for %ld days\n\n",ArraySize);
      for(i=0;i<ArraySize;i++)
      {
            printf("%7.2f Ruppees\n",ShoppingBill[i]);
      }
      return(0);
}
```

The output of this program is given below,

```
Shopping Bills for 7 days

  72.25 Ruppees
 100.25 Ruppees
  88.80 Ruppees
  90.00 Ruppees
  70.50 Ruppees
 105.50 Ruppees
  95.75 Ruppees
```

The statement

`float ShoppingBill[] = {72.25,100.25,88.80,90.0,70.5,105.5,95.75};`

causes the compiler to declares an array, `ShoppingBill`, of floating point numbers. The size of the array is made equal number of items in the intialisation list. It is possible for the programmer to determine the size of the array, as is done in the program by the expression

`sizeof(ShoppingBill)/sizeof(float)`. The expression `sizeof(ShoppingBill)` gives the size of the array in bytes. To determine the total number of elements in the array, it is divided by the expression `sizeof(float)` which provides the size of the type `float`. The resultant value is the number of floating point numbers that the array can hold.

6.6 Arrays and Pointers

A pointer is a symbolic way of using addresses. Since the hardware instructions of computing machines use addresses heavily, pointers allow the expression of concepts in a

way that is close to the way the machines express themselves. This makes programs with pointers efficient. Moreover, pointers provide an efficient way of dealing with arrays.

An array's name is also a pointer to the first element of an array. Thus if `MyArray[]` is an array, then

```
MyArray == &MyArray[0]
```

Both represent the memory address of the first element of the array. Both are pointer constants and they remain fixed for the duration of the program. However, they can be assigned as values to a pointer variable. The value assigned to this pointer can be changed, for example,

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
      int i, numbers[10], *numptr, data;
      char Temp[10];
      printf ("Address at the beginning of the array %u\n",numbers);
      numptr = numbers;
      for (i=0;i<10;i++)
      {
            printf("Enter number %d ",i);
            gets(Temp);
            numbers[i] = atoi(Temp);
            printf("%d\n",*(numptr+i));
      }
      for (i=0;i<10;i++)
      {
            printf("The value in location %d at address %u is ",
                              i,numptr+i);
            printf("%d\n",*(numptr+i));
      }
      return(0);
}
```

Going through the first iteration of the second loop, `0` is added to `numptr`. `numptr` still points to the first address of the array. In the next iteration, the value of `i` is incremented to `1` and `numptr + i` , when interpreted by C is translated to *add one storage unit*. For arrays, this means that address is increased to the address of the next element and not just one byte. This is another reason why it is important to declare a pointer variable with

respect to the type of variable a pointer points to. The address is not just enough, the computer needs to know how many bytes are used to store the variable. Thus it can be said that

```
numbers + 2 == &numbers[2]
*(numbers+2) == numbers[2]
```

It is important not to confuse `*(numbers+2)` with `*numbers + 2` . The indirection operator (`*`) has a higher precedence than `+`, so the latter expression means `(*numbers)+2`, i.e., `2` added to the value of the first element.

6.7 Arrays as Arguments to Functions

Consider the following example,

```
#include <stdio.h>
#include <stdlib.h>

void getData(int InArray[]);
void printData(int OutArray[]);

int main(void)
{
      int numbers[10];
      getData(numbers);
      printData(numbers);
      return(0);
}

void getData(int InArray[])
{
      char Temp[10];
      int i;
      for (i=0;i<10;i++)
      {
            printf ("Enter number %d ",i);
            gets(Temp);
            InArray[i] = atoi(Temp);
      }
}

void printData(int OutData[])
{
      int i;
      for (i=0;i<10;i++)
      {
            printf("Number[%d] = %d\n",i,OutData[i]);
      }
}
```

In the function declarations, `InArray[]` or `OutArray[]` does not create an array but a pointer. In the function calls, the argument `numbers` is a pointer to the first element of a 10 element array. The function calls pass pointers to both the functions. So these functions can be rewritten as

```
void getData(int *InArray);
void printData(int *OutArray);
```

respectively.

Thus when an array name is used as a function argument, a pointer is passed to the function. The function then uses this pointer to effect changes on the original array in the calling program. The above mentioned example can be rewritten as follows;

```
#include <stdio.h>
#include <stdlib.h>

void getData(int *InArray);
void printData(int *OutArray);

int main(void)
{
      int numbers[10];
      getData(numbers);
      printData(numbers);
      return(0);
}

void getData(int *InArray)
{
      char Temp[10];
      int i;
      for (i=0;i<10;i++)
      {
            printf ("Enter number %d ",i);
            gets(Temp);
            *(InArray + i) = atoi(Temp);
      }
}

void printData(int *OutData)
{
      int i;
      for (i=0;i<10;i++)
      {
            printf("Number[%d] = %d\n",i,*(OutData+i));
      }
```

```
}
```

It is also important to note that none of these functions have any information regarding the size of these arrays. This information can be hard coded in the programs as shown above or the last element of the array may contain a special value indicating the end of the array. It is also possible to pass this information as a separate argument to the function, e.g.,

```
void getData(int InArray[],int ArraySize);
```

The following program lets the user enter a variable number of integer values, not more than 10 integers, and calculate their mean and sum. In the end, these statistics are presented to the user.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

int getData(int *InArray);
int add(int *InArray,int Count);
float getMean(int Sum,int Count);

int main(void)
{
 int Count;
 int Numbers[MAX];
 int Sum;
 float Mean;
 Count = getData(Numbers);
 Sum = add(Numbers,Count);
 Mean = getMean(Sum,Count);
 printf("Count = %d :: Sum = %d :: Mean = %f\n",Count,Sum,Mean);
 return(0);
}

float getMean(int Sum, int Count)
{
 float Mean;
 Mean = ((float) Sum)/((float) Count);
 return(Mean);
}

int add(int *InData,int Count)
{
 int i;
 int Sum = 0;
 for (i=0;i < Count;i++)
 {
```

```
    Sum += *(InData+i);
  }
  return(Sum);
}

int getData(int *InData)
{
  char ExitChoice, Temp[10];
  int Count = 0;
  do
  {
    printf("Enter Number %d : ",Count);
    gets(Temp);
    *(InData+Count) = atoi(Temp);
    Count++;
    puts("Press 'x' to exit, anyother key to continue entering data");
    ExitChoice = getche();
    puts("");
  }
  while((ExitChoice != 'x') && (Count < 10));
  return(Count);
}
```

6.8 Multi-Dimensional Arrays

C language allows arrays of any type, including arrays of arrays. If two bracket pairs are used in a declaration after the identifier, a two dimensional array is declared. A two dimensional array can be regarded as an array of arrays. For example, consider the following declaration,

```
int Data[10][15];
```

Here `Data` is an array of `10` elements, where each of these `10` elements is an array of `15` integer numbers. Similarly, if three bracket pairs are used after the identifier, in the declaration, a three dimensional array is declared. For example,

```
float Sales[3][10][4];
```

Thus, the array dimension is increased by adding a bracket pair in the declaration. The following discussion focuses on two dimensional arrays. Whatever is valid for two dimensional arrays can be generalised to arrays of higher dimensions.

6.8.1 Two Dimensional Arrays

Consider a program that calculates the total yearly rainfall for three years. The program uses a two dimensional array to store the rainfall for each month of the last three years. The source code is given below,

```
#include <stdio.h>

int main(void)
{
  float rainfall[3][12];
  float YearlyRainfall[3];
  int i,j;

  for(i=0;i<3;i++)
  {
    for(j=0;j<12;j++)
    {
      printf("Enter rainfall[%d][%d] :: ",i,j);
      scanf("%f",&rainfall[i][j]);
    }
  }

  for(i=0;i<3;i++)
  {
    for(j=0;j<12;j++)
    {
      printf("%6.2f",rainfall[i][j]);
    }
    printf("\n");
  }

  for(i=0;i<3;i++)
  {
    YearlyRainfall[i] = 0.0;
    for(j=0;j<12;j++)
    {
      YearlyRainfall[i] += rainfall[i][j];
    }
  }

  for(i=0;i<3;i++)
  {
    printf("Total rainfall in year %d :: %7.2f\n",i,YearlyRainfall[i]);
  }
  return(0);
}
```

The statement `float  rainfall[3][12];` creates a two dimensional array in the memory. It can be visualised as a table with 3 rows and 12 columns (figure 6-4).

rainfall[0][0]	rainfall[0][1]	rainfall[0][2]	rainfall[0][3]	rainfall[0][4]	rainfall[0][5]	rainfall[0][6]	rainfall[0][7]	rainfall[0][8]	rainfall[0][9]	rainfall[0][10]	rainfall[0][11]
rainfall[1][0]	rainfall[1][1]	rainfall[1][2]	rainfall[1][3]	rainfall[1][4]	rainfall[1][5]	rainfall[1][6]	rainfall[1][7]	rainfall[1][8]	rainfall[1][9]	rainfall[1][10]	rainfall[1][11]
rainfall[2][0]	rainfall[2][1]	rainfall[2][2]	rainfall[2][3]	rainfall[2][4]	rainfall[2][5]	rainfall[2][6]	rainfall[2][7]	rainfall[2][8]	rainfall[2][9]	rainfall[2][10]	rainfall[2][11]

Figure 6-4 : Two Dimensional Array `rainfall[3][12]`

To access a particular element of the array, two index values need to be specified, one for the row and the other for the column. For instance, the following program segment is used to enter data into the array elements.

```
for(i=0;i<3;i++)
{
  for(j=0;j<12;j++)
  {
    printf("Enter rainfall[%d][%d] :: ",i,j);
    scanf("%f",&rainfall[i][j]);
  }
}
```

In this program segment, variables `i` and `j` are used as the array subscripts. Variable `i` is used to access a specific row of the array and variable `j` is used to access a particular column for that row. For each value of `i`, the value of `j` is varied from `0` to `11`. Thus, when the value of `i` is `1` and the value of `j` is `3`, the expression `rainfall[i][j]` gives the value of the element `rainfall[1][3]`. The statement `scanf("%f",&rainfall[i][j]);` allows the user to enter a floating point number into an array element specified by the values of variables `i` and `j`.

Another one dimensional array, `YearlyRainfall[3]`, is declared to store the total rainfall for each year. Each element of this array will hold the sum of all the columns of the array `rainfall` for that particular row. The following program segment performs this operation.

```
for(i=0;i<3;i++)
{
  YearlyRainfall[i] = 0.0;
```

```
  for(j=0;j<12;j++)
  {
    YearlyRainfall[i] += rainfall[i][j];
  }
}
```

For each value of `i`, i.e., every row, the respective array element of `YearlyRainfall` is initialised to `0` by the statement `YearlyRainfall[i] = 0.0;`. The loop after this statement performs a running total of all the elements of that specific row of the array `rainfall`. The following program segment then prints the total rainfall for each of the three years.

```
for(i=0;i<3;i++)
{
  printf("Total rainfall in year %d :: %7.2f\n",i,YearlyRainfall[i]);
}
```

A typical execution session of this program is shown below,

```
Enter rainfall[0][0] :: 10.5
Enter rainfall[0][1] :: 10.2
Enter rainfall[0][2] :: 9.8
Enter rainfall[0][3] :: 6.7
Enter rainfall[0][4] :: 6.1
Enter rainfall[0][5] :: 5.8
Enter rainfall[0][6] :: 4.5
Enter rainfall[0][7] :: 6.2
Enter rainfall[0][8] :: 7.5
Enter rainfall[0][9] :: 9.6
Enter rainfall[0][10] :: 9.9
Enter rainfall[0][11] :: 10.3
Enter rainfall[1][0] :: 10.6
Enter rainfall[1][1] :: 10.7
Enter rainfall[1][2] :: 10.2
Enter rainfall[1][3] :: 7.1
Enter rainfall[1][4] :: 6.7
Enter rainfall[1][5] :: 5.8
Enter rainfall[1][6] :: 4.9
Enter rainfall[1][7] :: 5.7
Enter rainfall[1][8] :: 7.0
Enter rainfall[1][9] :: 9.1
Enter rainfall[1][10] :: 9.4
Enter rainfall[1][11] :: 9.9
Enter rainfall[2][0] :: 10.7
Enter rainfall[2][1] :: 9.9
Enter rainfall[2][2] :: 9.5
Enter rainfall[2][3] :: 6.2
Enter rainfall[2][4] :: 6.2
Enter rainfall[2][5] :: 5.5
Enter rainfall[2][6] :: 4.2
Enter rainfall[2][7] :: 5.7
Enter rainfall[2][8] :: 7.1
Enter rainfall[2][9] :: 8.9
Enter rainfall[2][10] :: 9.2
Enter rainfall[2][11] :: 10.3
   10.50   10.20    9.80    6.70    6.10    5.80    4.50    6.20    7.50    9.60    9.90   10.30
   10.60   10.70   10.20    7.10    6.70    5.80    4.90    5.70    7.00    9.10    9.40    9.90
   10.70    9.90    9.50    6.20    6.20    5.50    4.20    5.70    7.10    8.90    9.20   10.30
Total rainfall in year 0 ::    97.10
Total rainfall in year 1 ::    97.10
Total rainfall in year 2 ::    93.40
```

To summarise, in order to find the total rainfall for a given year, the value of `i` is kept constant and the value of `j` is varied over its full range. This is accomplished by the inner `for` loop. Then the process is repeated for the next value of `i`, i.e., the next year. This is accomplished by the outer `for` loop. A nested loop structure like this is natural for handling two dimensional loops. One loop handles one subscript whereas the other loop handles the second subscript.

Consider another version of the same program. This program takes in the rainfall measured for each month in three years and then prints the average rainfall for each month. The source code of this program is shown below,

```
#include <stdio.h>

int main(void)
{
  float rainfall[3][12];
  float MonthlyAverage[12];
  float Sum;
  int i,j;

  for(i=0;i<3;i++)
  {
    for(j=0;j<12;j++)
    {
      printf("Enter rainfall[%d][%d] :: ",i,j);
      scanf("%f",&rainfall[i][j]);
    }
  }

  for(i=0;i<3;i++)
  {
    for(j=0;j<12;j++)
    {
      printf("%6.2f",rainfall[i][j]);
    }
    printf("\n");
  }

  for(j=0;j<12;j++)
  {
    Sum = 0.0;
    for(i=0;i<3;i++)
    {
      Sum += rainfall[i][j];
    }
    MonthlyAverage[j] = Sum/3.0;
  }

  for(i=0;i<12;i++)
  {
    printf("Average rainfall in Month %2d :: %7.2f\n",i,MonthlyAverage[i]);
  }
  return(0);
}
```

The following section of the program calculates the average rainfall for each month,

```
for(j=0;j<12;j++)
{
  Sum = 0.0;
  for(i=0;i<3;i++)
  {
    Sum += rainfall[i][j];
  }
  MonthlyAverage[j] = Sum/3.0;
}
```

In contrast to the previous program, the `for` loop with the subscript `j` as the loop control variable is now the outer loop and the `for` loop with the subscript `i` as the inner loop is the inner loop. Each time the outer loop cycles once, the inner loop cycles its complete allotment. Thus, this arrangement cycles through all the years (subscript `i`) before changing months (subscript `j`). As a result, the three year total for the first month is calculated first, then the three year total for the second month, and so on. A typical execution session of this program is shown below,

```
Enter rainfall[0][0] :: 10.5
Enter rainfall[0][1] :: 10.2
Enter rainfall[0][2] :: 9.8
Enter rainfall[0][3] :: 6.7
Enter rainfall[0][4] :: 6.1
Enter rainfall[0][5] :: 5.8
Enter rainfall[0][6] :: 4.5
Enter rainfall[0][7] :: 6.2
Enter rainfall[0][8] :: 7.5
Enter rainfall[0][9] :: 9.6
Enter rainfall[0][10] :: 9.9
Enter rainfall[0][11] :: 10.3
Enter rainfall[1][0] :: 10.6
Enter rainfall[1][1] :: 10.7
Enter rainfall[1][2] :: 10.2
Enter rainfall[1][3] :: 7.1
Enter rainfall[1][4] :: 6.7
Enter rainfall[1][5] :: 5.8
Enter rainfall[1][6] :: 4.9
Enter rainfall[1][7] :: 5.7
Enter rainfall[1][8] :: 7.0
Enter rainfall[1][9] :: 9.1
Enter rainfall[1][10] :: 9.4
Enter rainfall[1][11] :: 9.9
Enter rainfall[2][0] :: 10.7
Enter rainfall[2][1] :: 9.9
Enter rainfall[2][2] :: 9.5
Enter rainfall[2][3] :: 6.2
Enter rainfall[2][4] :: 6.2
Enter rainfall[2][5] :: 5.5
Enter rainfall[2][6] :: 4.2
Enter rainfall[2][7] :: 5.7
Enter rainfall[2][8] :: 7.1
Enter rainfall[2][9] :: 8.9
Enter rainfall[2][10] :: 9.2
Enter rainfall[2][11] :: 10.3
   10.50   10.20    9.80    6.70    6.10    5.80    4.50    6.20    7.50    9.60    9.90   10.30
   10.60   10.70   10.20    7.10    6.70    5.80    4.90    5.70    7.00    9.10    9.40    9.90
   10.70    9.90    9.50    6.20    6.20    5.50    4.20    5.70    7.10    8.90    9.20   10.30
Average rainfall in Month  0 :: 10.60
Average rainfall in Month  1 :: 10.27
```

```
Average rainfall in Month  2 ::  9.83
Average rainfall in Month  3 ::  6.67
Average rainfall in Month  4 ::  6.33
Average rainfall in Month  5 ::  5.70
Average rainfall in Month  6 ::  4.53
Average rainfall in Month  7 ::  5.87
Average rainfall in Month  8 ::  7.20
Average rainfall in Month  9 ::  9.20
Average rainfall in Month 10 ::  9.50
Average rainfall in Month 11 :: 10.17
```

6.8.2 Initialising Two Dimensional Arrays

The following program is a version of the previous program.

```
#include <stdio.h>

int main(void)
{
        float rainfall[3][12]={{10.5,10.2,9.8,6.7,6.1,5.8,4.5,6.2,7.5,9.6,9.9,10.3},
                               {10.6,10.7,10.2,7.1,6.7,5.8,4.9,5.7,7.0,9.1,9.4,9.9},
                               {10.7,9.9,9.5,6.2,6.2,5.5,4.2,5.7,7.1,8.9,9.2,10.3}};
        float MonthlyAverage[12];
        float Sum;
        int i,j;

        for(i=0;i<3;i++)
        {
                for(j=0;j<12;j++)
                {
                        printf("%6.2f",rainfall[i][j]);
                }
                printf("\n");
        }

        for(j=0;j<12;j++)
        {
                Sum = 0.0;
                for(i=0;i<3;i++)
                {
                        Sum += rainfall[i][j];
                }
                MonthlyAverage[j] = Sum/3.0;
        }

        for(i=0;i<12;i++)
        {
                printf("Average rainfall in Month %2d :: %7.2f\n",i,MonthlyAverage[i]);
        }
        return(0);
}
```

In this program, the array `rainfall[3][12]` is initialised in the following declaration

```
        float rainfall[3][12]={{10.5,10.2,9.8,6.7,6.1,5.8,4.5,6.2,7.5,9.6,9.9,10.3},
                               {10.6,10.7,10.2,7.1,6.7,5.8,4.9,5.7,7.0,9.1,9.4,9.9},
                               {10.7,9.9,9.5,6.2,6.2,5.5,4.2,5.7,7.1,8.9,9.2,10.3}};
```

For the initialisation, 3 embraced sets of numbers are included, all enclosed by one more set of braces. The data in the first interior set of braces is assigned to the first row of the array, the data in the second interior set is assigned to the second row, and so on.

The rules about mismatches between data and array sizes that are valid for one dimensional arrays also apply to multi-dimensional arrays. That is, if the first set of braces enclose 10 numbers, only the first 10 elements of the first row are affected. The last two elements in that row would get the standard default initialisation to zero. If there are too many numbers, that is an error, extra numbers do not automatically get assigned to elements of the next row.

6.8.3 Two Dimensional Arrays and Pointers

Consider the following declaration,

```
float AnArray[2][4];
```

`AnArray` being the name of the array is a pointer and it points to the first element of the array. In this case, the first element is itself an array of 4 floating point numbers. Therefore, `AnArray` points to the whole array `AnArray[0]`. As `AnArray[0]` is the name of the first array of floating point numbers, this makes `AnArray[0]` a pointer to its first element, the floating point number `AnArray[0][0]`. In short, `AnArray` points to `AnArray[0]`, which is an array of `float`, and `AnArray[0]` points to `AnArray[0][0]` which is a floating point number. This makes `AnArray` a pointer to a pointer.

The following program prints the addresses of `AnArray`, `AnArray[0]` and `AnArray[0][0]`.

```
#include <stdio.h>

int main(void)
{
      float AnArray[2][4];
      printf("The address AnArray        = %u\n",AnArray);
      printf("The address AnArray[0]     = %u\n",AnArray[0]);
      printf("The address &AnArray[0][0] = %u\n",&AnArray[0][0]);
      return(0);
}
```

The output of this program is given below,

```
The address AnArray        = 4064
```

```
The address AnArray[0]     = 4064
The address &AnArray[0][0] = 4064
```

This shows that the address of the first array in the array of arrays (`AnArray`) and the address of the first element in that array (`AnArray[0]`) are the same. Each is the address of the first element, i.e., each is numerically the same as `&AnArray[0][0]`. However, there is a difference. The pointer `AnArray[0]` is the address of a `float`, so it points to a 4-byte data object. The pointer `AnArray` is the address of an array of 4 `floats`, so it points to a 16 byte data object. Thus adding 1 to `AnArray` should produce an address 16 bytes larger, while adding 1 to `AnArray[0]` should produce an address 4 bytes larger. This is demonstrated in the following program.

```
#include <stdio.h>

int main(void)
{
      float AnArray[2][4];
      printf("The address AnArray        = %u\n",AnArray);
      printf("The address AnArray[0]     = %u\n",AnArray[0]);
      printf("The address &AnArray[0][0] = %u\n",&AnArray[0][0]);

      printf("\n\n");
      printf("The address AnArray+1         = %u\n",AnArray+1);
      printf("The address AnArray[0]+1      = %u\n",AnArray[0]+1);
      printf("The address &AnArray[0][0]+1  = %u\n",&AnArray[0][0]+1);
      return(0);
}
```

The output of this program is given below,

```
The address AnArray        = 4064
The address AnArray[0]     = 4064
The address &AnArray[0][0] = 4064


The address AnArray+1         = 4080
The address AnArray[0]+1      = 4068
The address &AnArray[0][0]+1  = 4068
```

Thus, adding 1 to `AnArray` moves the pointer from one array to the next array, while adding 1 to `AnArray[0]` moves the pointer from one array element to the next array element.

6.8.4 Functions and Multidimensional Arrays

There are several ways by which a function can be used to deal with two dimensional arrays. For example, a function can be written to process one dimensional arrays, and then this function is used to process each subarray of a two dimensional array. It is also possible to write a function that treats the complete two dimensional array as a one dimensional array and processes it as a one dimensional array. Alternatively, a function can be developed to explicitly deal with two dimensional arrays.

6.8.4.1 Applying One Dimensional Function to Subarrays

The following program uses a function that takes a one dimensional array of characters and its size, and converts each lowercase character into an uppercase character. The `main` function declares and initialises a two dimensional array of characters and passes each subarray successively to this function. In the end the `main` function prints the contents of the two dimensional array.

```
#include <stdio.h>
#include <conio.h>

#define ROW 3
#define COLUMN 5

void makeUpperCase(char InData[],int Size);

int main(void)
{
	int i,j;
	char Data[ROW][COLUMN];
	for(i=0;i<ROW;i++)
	{
		for(j=0;j<COLUMN;j++)
		{
			Data[i][j] = getche();
		}
		printf("\n");
	}

	for (i=0;i<ROW;i++)
	{
		makeUpperCase(Data[i],COLUMN);
	}

	printf("\n\n");
	for (i=0;i<ROW;i++)
	{
		for(j=0;j<COLUMN;j++)
		{
```

```
                putchar(Data[i][j]);
            }
            putchar('\n');
      }

      return(0);
}

void makeUpperCase(char InData[],int Size)
{
      int i;
      for(i=0;i<Size;i++)
      {
            if ((InData[i] <= 'z') && (InData[i] >= 'a'))
            {
                  InData[i] -= 32;
            }
      }
}
```

A `for` loop in the function `main` uses the `makeUpperCase` function to process the subarray `Data[0]`, then the subarray `Data[1]` and finally subarray `Data[2]`. Also passed to the function `makeUpperCase` is the size of each subarray. An execution cycle of this program provides the following output,

```
Hello
HowDo
YouDo

HELLO
HOWDO
YOUDO
```

6.8.4.2 Applying a One-Dimensional Function to the Whole Array

In the preceding example, the two dimensional array `Data` was taken as an array of 3 arrays of 5 characters each. Alternatively the two dimensional array `Data` can also be taken as an array of 15 characters. Consider the following version of the example shown above,

```
#include <stdio.h>
#include <conio.h>

#define ROW 3
#define COLUMN 5

void makeUpperCase(char InData[],int Size);

int main(void)
```

```
{
	int i,j;
	char Data[ROW][COLUMN];
	for(i=0;i<ROW;i++)
	{
		for(j=0;j<COLUMN;j++)
		{
			Data[i][j] = getche();
		}
		printf("\n");
	}

	makeUpperCase(Data[0],COLUMN * ROW);

	printf("\n\n");
	for (i=0;i<ROW;i++)
	{
		for(j=0;j<COLUMN;j++)
		{
			putchar(Data[i][j]);
		}
		putchar('\n');
	}

	return(0);
}

void makeUpperCase(char InData[],int Size)
{
	int i;
	for(i=0;i<Size;i++)
	{
		if ((InData[i] <= 'z') && (InData[i] >= 'a'))
		{
			InData[i] -= 32;
		}
	}
}
```

In this particular case, the `makeUpperCase` function call receives `Data[0]` as the first argument and an integer value 3×5, i.e., 15 as the second argument. The first argument initialises the pointer `InData` in `makeUpperCase` to the address of `Data[0][0]`. Thus, `InData[0]` corresponds to `Data[0][0]` and `InData[4]` corresponds to `Data[0][4]`. However, `InData[5]` corresponds to the element following `Data[0][4]` which is `Data[1][0]`, the first element of the next subarray. Also note that the definition of the function `makeUpperCase` remains unchanged. The second parameter is used to indicate the size of the complete array, rather than just one subarray. As a result only one call to the function `makeUpperCase` is necessary, as opposed to 3 calls in the previous example.

A typical execution session of this program produces the following output,

```
Point
ersar
egood

POINT
ERSAR
EGOOD
```

6.8.4.3 Apply a Two Dimensional Function

Both of the approaches discussed so far lose track of the column and row information. In this particular application, this information may be unimportant, but in certain other applications it may be necessary to retain this information within the called function. For instance a function that is supposed to process the information column-wise rather than row-wise should have the row and column information available. This can be accomplished by declaring the right kind of formal variable so that the function can pass the array properly. In this case, the pointer to the first element of the two dimensional array is passed to the called function. There it is assigned to a variable which is a pointer to an array of elements equal to the number of columns in the original two dimensional array.

Consider the following version of the original example,

```
#include <stdio.h>
#include <conio.h>

#define ROW 3
#define COLUMN 5

void makeUpperCase(char InData[][5]);

int main(void)
{
      int i,j;
      char Data[ROW][COLUMN];
      for(i=0;i<ROW;i++)
      {
            for(j=0;j<COLUMN;j++)
            {
                  Data[i][j] = getche();
            }
            printf("\n");
      }
```

```
	makeUpperCase(Data);

	printf("\n\n");
	for (i=0;i<ROW;i++)
	{
		for(j=0;j<COLUMN;j++)
		{
			putchar(Data[i][j]);
		}
		putchar('\n');
	}

	return(0);
}

void makeUpperCase(char InData[][5])
{
	int i;
	int j;
	for(i=0;i<ROW;i++)
	{
		for(j=0;j<COLUMN;j++)
		{
			if ((InData[i][j] <= 'z') && (InData[i][j] >= 'a'))
			{
				InData[i][j] -= 32;
			}
		}
	}
}
```

Here, `Data` is the pointer to the first sub array of 5 characters. In the definition of function `makeUpperCase`, it is assigned to `InData[][5]`, which is a pointer to an array of 5 characters. The length of a row is built into the function, but the number of rows is left open.

Output of a typical execution session of this program is given below,

```
ptrsa
relot
offun

PTRSA
RELOT
OFFUN
```

6.9 Strings

C has no special variable type for strings. Instead, strings are stored in a one dimensional array of type `char`. By convention, a string in C is terminated by the end-of-string sentinel `\0`, or null character. It is useful to think of strings having a variable length, delimited by `\0`, but with a maximum length determined by the size of the string. The size of the string must include the storage needed for the end-of-string sentinel. As with all arrays, it is the job of the programmer to make sure that string bounds are not overrun.

String constants are written between double quotes. For example, `"abc"` is a character array of size 4, with the last element being the null character `\0`. This string may be visualised as shown in figure 6-5. Note that string constants are different from character constants. For example, `"a"` and `'a'` are not the same. The array `"a"` has two elements, the first with a value `'a'` and second with a value `'\0'`.

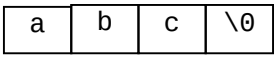


Figure 6-5 : A String

String variables, as discussed above are arrays of type `char`. The size of the array must be kept one more than the longest string that will be stored in the array. This extra one byte is used to store the null terminating character at the end of string. The following program reads a string from the keyboard and prints it on the display.

```
#include <stdio.h>

int main(void)
{
      char Name[50];
      printf("Enter your name ");
      scanf("%s",Name);
      printf("Hello %s\n",Name);
      return(0);
}
```

The statement `char Name[50];` declares an array `Name` of 50 characters. The statement `scanf("%s",Name);` reads characters from the keyboard and stores them the array `Name`. When the user enters a whitespace character, the `scanf` function inserts a `\0` in the array.

%s is the format specifier for strings. A typical execution session of this program is shown below,

```
Enter your name Omar
Hello Omar
```

6.9.1 String Input Output

As mentioned earlier, scanf function scans only till a whitespace character and inserts a null terminating character in the array. Thus a multiword string cannot be entered into an array using the scanf function. Consider the following execution cycle of the above mentioned program,

```
Enter your name Omar Bashir
Hello Omar
```

The solution to this problem is to use another C library function, gets. The purpose of gets is get a string from the keyboard and assign its contents to the specified character array. It is not as versatile a function as scanf as it only specialises in reading strings. It is terminated only when the return/enter key is pressed. Therefore, tabs and spaces are acceptable parts of the input string. The gets function is a part of the matching pair. The other, puts, is used to display a specified string on the monitor. Following is a version of the above mentioned program that uses gets and puts functions to read and print character strings.

```
#include <stdio.h>

int main(void)
{
      char Name[50];
      printf("Enter your name ");
      gets(Name);
      puts("Hello");
      puts(Name);
      return(0);
}
```

A typical execution session of this program is given below,

```
Enter your name Omar Bashir
Hello
Omar Bashir
```

As shown in the program, `gets` function takes only one parameter, i.e., a pointer to a character. As `Name` is the name of an array of characters, it is the pointer to the first character of the array. `gets` function reads characters until it reaches a newline character (`'\n'`), which the user generates by pressing the enter key. It takes all the characters before (but not including) the newline character, attaches a null terminating character at the end and places the string in the memory location specified by the pointer passed to it as the argument.

The `puts` function takes only one argument, that is the pointer to the string that is to be displayed. `puts` starts printing characters from the address it is passed till it encounters the null terminating character. If, for some reason, the null terminating character of a string gets overwritten by some other character, `puts` will continue printing characters from successive memory locations until it encounters null somewhere in the memory location. Another characteristic of `puts` is that each string printed by `puts` is displayed on a new line. For a string that is being printed, when puts finds a null terminating character, it replaces it with a newline character and then prints it.

6.9.2 Initialising Strings

Since strings are arrays of characters, these can be initialised in a manner similar to arrays. The following example shows how a string can be initialised in a program,

```
#include <stdio.h>

int main(void)
{
      char Str[] = {'T','h','i','s',' ','i','s',' ','a',' ','s','t',
                    'r','i','n','g','\0'};
      puts(Str);
      return(0);
}
```

The output of this program is

```
This is a string
```

Here `Str[]` is an array of characters, each element of which is initialised by the respective character in the array initialiser. Each character in the initialiser has to be surrounded by a pair of single quote marks and the last character in the initialiser must be a null terminating character. The same program may be written as follows,

```
#include <stdio.h>

int main(void)
{
      char Str[] = "This is a string";
      puts(Str);
      return(0);
}
```

The output of this program is as follows,

```
This is a string
```

Here the string Str[] is initialised by a string constant. The complete string constant is surrounded by a pair of double quotes. A null terminating character is not required at the end as the string format causes this to happen automatically.

6.9.3 String Handling Functions

C inherently has no special string handling operators. Rather, C employs a large set of useful string handling library functions. These functions have been written in C and are all quite short. The prototypes of these functions are declared in the header file `string.h`. This file, therefore needs to be included in programs where these functions are to be used. Some of the most widely used string handling functions are described in the following paragraphs.

6.9.3.1 `strlen`

`strlen` takes, as an argument, a pointer to a string and returns the number of characters in the string except the null terminating character. Consider the following example,

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char AString[50];
  int SizeOfArray;
  int SizeOfString;
  printf("Enter a string :: ");
  gets(AString);
  SizeOfArray = sizeof(AString);
  SizeOfString = strlen(AString);
  printf("The array is %d characters long\n",SizeOfArray);
  printf("The string \"%s\" is %d characters long\n",AString,SizeOfString);
  return(0);
}
```

An execution session of this program is given below,

```
Enter a string :: Hello World
The array is 50 characters long
The string "Hello World" is 11 characters long
```

The statement `char AString[50];` declares an array of 50 characters. The function call `gets(AString);` allows the user to enter a string which is stored in the character array `AString`. For the above mentioned example, the state of the array is shown in figure 6-6. In the statement `SizeOfArray = sizeof(AString);` the `sizeof` operator determines the length of the array `AString`. This value is assigned to the variable `SizeOfArray`. Alternatively, in the statement `SizeOfString = strlen(AString);` the `strlen` function determines the number of characters preceding the first null terminating character in the array. This value is assigned to the variable `SizeOfString`.

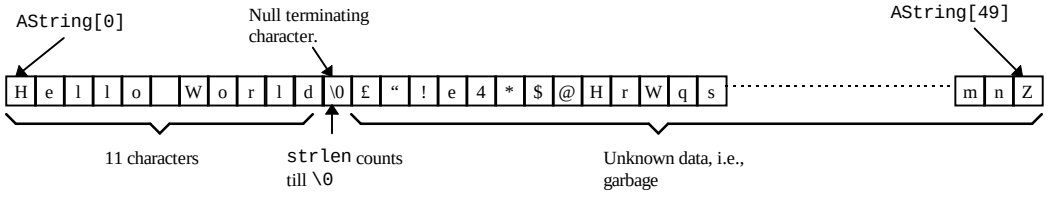


Figure 6-6 : `strlen` Operation

Note that the value of the variable `SizeOfString` may change for different execution cycles of the program, whereas the value of the variable `SizeOfArray` is same for each execution cycle. This is because the size of the array remains constant for each execution cycle of the program. Consider another execution cycle,

```
Enter a string :: Good Morning
The array is 50 characters long
The string "Good Morning" is 12 characters long
```

6.9.3.2 `strcat`

`strcat` function is used to concatenate two strings. This function takes two strings as arguments. A copy of the second string is attached on to the end of the first string. The resultant concatenated string is assigned to the first argument. The second string is not altered. Consider the following example,

```
#include <stdio.h>
#include <string.h>

int main(void)
{
      char String1[15];
      char String2[15];
      printf("Enter String1 : ");
      gets(String1);
      printf("Enter String2 : ");
      gets(String2);
      strcat(String1,String2);
      printf("Operation : strcat(String1,String2)\n");
      printf("String1 = %s\n",String1);
      printf("String2 = %s\n",String2);
      return(0);
}
```

An execution cycle of this program is shown below,

```
Enter String1 : Omar
Enter String2 : Bashir
Operation : strcat(String1,String2)
String1 = OmarBashir
String2 = Bashir
```

`strcat` function does not check to see if the concatenated string will fit in the character array specified as the first argument. Thus, if a string that is larger in size than the first

character array is written into that array, the program may malfunction. One option with the programmers is to use `strlen` function and `sizeof` operator to insure that the concatenation operation is only allowed when the size of the concatenated string is less than the size of the first character array. Consider a modified version of the program shown above,

```
#include <stdio.h>
#include <string.h>

int main(void)
{
      char String1[15];
      char String2[15];
      printf("Enter String1 : ");
      gets(String1);
      printf("Enter String2 : ");
      gets(String2);
      printf("Operation : strcat(String1,String2)\n");
      if((strlen(String1) + strlen(String2)) < sizeof(String1))
      {
            strcat(String1,String2);
            printf("String1 = %s\n",String1);
            printf("String2 = %s\n",String2);
      }
      else
      {
            printf("Concatenated string will be too big\n");
      }
      return(0);
}
```

In this program, the expression `(strlen(String1) + strlen(String2))` determines the total length of the concatenated string, i.e., the number of characters that the concatenated string will contain minus the null terminating character. The expression `sizeof(String1)` determines the size of the character array `String1`. If the total number of characters in the final concatenated string will be less than the size of the character array that has been declared to hold it (i.e., the first argument of the `strcat` function call), the program executes the following statements,

```
            strcat(String1,String2);
            printf("String1 = %s\n",String1);
            printf("String2 = %s\n",String2);
```

The null terminated string contained in `String2` is appended to the null terminated string contained in `String1` and the resultant is placed in `String1`. After this operation, the contents of `String1` and `String2` are displayed.

Alternatively, if `String1` is not big enough to store the concatenated string, the following statement is executed,

```
printf("Concatenated string will be too big\n");
```

Two execution sessions of this program are shown below,

```
Enter String1 : Omar
Enter String1 : Bashir
Operation : strcat(String1,String2)
String1 = OmarBashir
String2 = Bashir
```

```
Enter String1 : Zulfiqar
Enter String1 : Kirmani
Operation : strcat(String1,String2)
Concatenated string will be too big
```

6.9.3.3 `strcmp`

`strcmp` takes pointers to two strings as arguments, compares them and returns an integer value based on the comparison. The value returned is zero if the contents of both the strings are the same. Otherwise a non-zero value is returned. Consider the following program. This program keeps asking the user to enter a string till the time that string has the same contents as a string predefined in the program.

```
#include <stdio.h>
#include <string.h>
#define SOLUTION "Ulan Bator"

int main(void)
{
      char Answer[40];
      do
      {
            printf("Name the capital of Mongolia : ");
            gets(Answer);
```

```
            if(strcmp(Answer,SOLUTION) == 0)
            {
                  puts("CORRECT");
            }
            else
            {
                  puts("WRONG :: Try again");
            }
      }
      while(strcmp(SOLUTION,Answer) != 0);
      return(0);
}
```

In this program, the value of the identifier `SOLUTION` has been defined as a string constant `"Ulan Bator"`. The program ask the user the following question,

```
Name the capital of Mongolia :
```

The statement `gets(Answer);` stores the user's response as a null terminated string in the array `Answer`. The expression `strcmp(Answer,SOLUTION)` returns a zero if the contents of `Answer` match with the string constant `"Ulan Bator"`. In that case, the message `CORRECT` is displayed to the user and the program terminates. If the string entered by the user is other than `"Ulan Bator"`, the result of the expression `strcmp(Answer,SOLUTION)` is a non-zero number. The message `WRONG :: Try again` is printed and another iteration of the `do-while` loop is executed, asking the user to enter the answer again. This process is repeated till the time the user enters a correct value.

A typical execution session of this program is shown below,

```
Name the capital of Mongolia : Moscow
WRONG :: Try again
Name the capital of Mongolia : Peking
WRONG :: Try again
Name the capital of Mongolia : Ulan Bator
CORRECT
```

`strcmp` compares strings, not arrays. Thus, although the array `Answer` can contain 40 characters and `"Ulan Bator"` contains only 11 characters (including the null terminating character), the comparison operation is performed only for the part of the array `Answer` up to its first null terminating character. Thus `strcmp` can be used to

compare strings stored in arrays of different sizes. Moreover, `strcmp` compares the ASCII values of the string characters. This makes `strcmp` case sensitive. Thus, for `strcmp`, "`Ulan Bator`" is different from "`ulan bator`", which is different from "`ULAN BATOR`".

Most implementations of `strcmp` function return the difference in ASCII code between the two characters. Others return a negative number if the first string comes before the second alphabetically, 0 if they are the same and a positive number if the first string follows the second alphabetically. In general, `strcmp` skips characters until it finds the first pair of disagreeing characters. It then returns the difference in the ASCII code of the corresponding characters of the two strings. Consider the following example,

```
#include <stdio.h>
#include <string.h>

int main(void)
{
        printf("strcmp(\"A\",\"A\") = %d\n",strcmp("A","A"));
        printf("strcmp(\"A\",\"B\") = %d\n",strcmp("A","B"));
        printf("strcmp(\"C\",\"A\") = %d\n",strcmp("C","A"));
        printf("strcmp(\"Switch\",\"Stitch\") = %d\n",strcmp("Switch","Stitch"));

        return(0);
}
```

The output of this program is given below,

```
strcmp("A","A") = 0
strcmp("A","B") = -1
strcmp("C","A") = 2
strcmp("Switch","Stitch") = 3
```

In most of the cases, strings are only compared to determine whether they contain the same elements or not. In such cases, the programmers only check if the return from the `strcmp` function is a 0 or a nonzero number. However, in applications such as sorting a list of string, the programmers need to check if the return value was zero, a positive number or a negative number.

6.9.3.4 `strcpy`

`strcpy` function takes pointers to two strings as arguments. It copies the contents of the string pointed to by the second argument into the string pointed to by the first argument.

The contents of the string pointed to by the first argument are overwritten. The contents of the string pointed to by the second argument remain unchanged. Consider the following program,

```
#include <stdio.h>
#include <string.h>

int main(void)
{
      char Original[] = "This is a string";
      char Copy[20];
      printf("The original string : \"%s\"\n",Original);
      printf("Copying the contents of Original into Copy\n");
      strcpy(Copy,Original);
      printf("The copied string : \"%s\"\n",Copy);
      printf("The original string : \"%s\"\n",Original);

      return(0);
}
```

The output of this program is given below,

```
The original string : "This is a string"
Copying the contents of Original into Copy
The copied string : "This is a string"
The original string : "This is a string"
```

The statement `strcpy(Copy,Original);` copies the contents of the string pointed to by `Original` to the string pointed to by `Copy`. The contents of the string pointed to by `Original` remain unchanged whereas the contents of the string pointed to by `Copy` are overwritten by the contents of the string pointed to by `Original`. It should be noted that `strcpy` does not check the size of the destination array before copying the contents of the source string to the destination. It is, therefore the responsibility of the programmer to ensure that the destination array has enough room to accommodate the incoming string.

6.9.4 Accessing Individual String Elements

As mentioned earlier, a string is a null terminated array of characters. In several applications, the ability to access and manipulate individual characters in a string is required. Characters in a string can be accessed as elements of the array holding the

string. However, the elements of interest in that array are generally the characters placed before the null terminating character.

The following program asks the user to enter a character string. Once the user has enter the string, the program sends the pointer to the string to a function that converts all the lowercase characters in the string to uppercase.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void string2UpperCase(char []);

int main(void)
{
      char String[40];
      puts("Enter a character string ");
      gets(String);
      string2UpperCase(String);
      puts(String);
      return(0);
}

void string2UpperCase(char InString[])
{
      int i = 0;
      while(InString[i] != '\0')
      {
            InString[i] = toupper(InString[i]);
            i++;
      }
}
```

In this program, the function `string2UpperCase` takes in a pointer to a character array. The loop `while(InString[i] != '\0')` examines each character of the array, starting from the first character, and if the character is not the null character, the body of the loop is executed. Inside the body of the loop, the statement

`InString[i] = toupper(InString[i]);` uses the `toupper` function to convert the input character to an uppercase character if it is a lowercase character. All other characters are returned without any change. The prototype of the `toupper` function is declared in `ctype.h`. The returned character is assigned back to the same element of the array `InString`. Thus, the function `string2UpperCase` converts all the characters of the string to uppercase.

An alternate implementation of the `string2UpperCase` function would use the `strlen` function to determine the length of the string, minus the null terminating character. The function can then employ a `for` loop to iterate as many times as the number of characters in the string. Within the body of the loop, a character from the string is accessed and is converted to uppercase if it is a lowercase character. Alternate implementation of this function is shown below,

```
void string2UpperCase(char InString[])
{
      int i;
      int Length;
      Length = strlen(InString);
      for(i=0;i<Length;i++)
      {
            InString[i] = toupper(InString[i]);
      }
}
```

A typical execution session of this program is given below,

```
Enter a character string
Hello World.
HELLO WORLD.
```

6.9.5 Arrays of Strings

A string is an array of characters. An array of strings should therefore be an array of arrays, i.e., a two dimensional array. An array of strings is declared as follows,

```
char StringArray[10][40];
```

The order of subscripts in an array declaration is important. The first subscript gives the number of items (i.e., strings) in the array, while the second subscript gives the maximum length each string in the array can have. Thus, for the declaration shown above, `StringArray` can have 10 strings where each string can contain a maximum of 40 characters, including the null terminating character.

Consider the following program. It allows the users to enter 10 strings. Then it converts characters in all the string to uppercase and prints them on the screen.

```
#include <stdio.h>
#include <ctype.h>

#define MAX 10
#define LEN 40

void string2Uppercase(char InString[]);

int main(void)
{
      char StringArray[MAX][LEN];
      int i;

      for(i=0;i<MAX;i++)
      {
            printf("Enter string[%d] :: ",i);
            gets(StringArray[i]);
      }

      for(i=0;i<MAX;i++)
      {
            string2Uppercase(StringArray[i]);
      }

      for(i=0;i<MAX;i++)
      {
            printf("String[%d] = %s\n",i,StringArray[i]);
      }
      return(0);
}

void string2Uppercase(char InString[])
{
      int i;
      for(i=0;i<strlen(InString);i++)
      {
            InString[i] = toupper(InString[i]);
      }
}
```

In the function `main`, separate loops are used to accept strings in the array (one at a time) from the user, convert them into uppercase and then print them. The function `string2Uppercase` performs the same operation described in the section 6.9.4. A typical execution session of this program is shown below,

```
Enter string[0] :: IBM PC
Enter string[1] :: Apple Macintosh
Enter string[2] :: Sun Microsystems
Enter string[3] :: Microsoft Windows
```

```
Enter string[4] :: Xerox Copiers
Enter string[5] :: ICL Mainframes
Enter string[6] :: Newfield Automation
Enter string[7] :: Computer Associates
Enter string[8] :: Texas Instruments
Enter string[9] :: Intel chips

String[0] = IBM PC
String[1] = APPLE MACINTOSH
String[2] = SUN MICROSYSTEMS
String[3] = MICROSOFT WINDOWS
String[4] = XEROX COPIERS
String[5] = ICL MAINFRAMES
String[6] = NEWFIELD AUTOMATION
String[7] = COMPUTER ASSOCIATES
String[8] = TEXAS INSTRUMENTS
String[9] = INTEL CHIPS
```

6.9.6 Conversion Between Strings and Numerical Data

In several cases, data that is fed to programs from files or other sources is inherently in the form of ASCII text strings. In such cases, numerical information, that must be managed and processed by the programs, needs to be converted into their appropriate data types and assigned to suitable variables for processing. C provides three functions that can be used to convert numerical information in text format to appropriate data types. These are the `atoi`, `atol` and `atof` functions. `atoi` stands for *ASCII to integer*, `atof` stands for *ASCII to floating point* and `atol` stands for *ASCII to long integer*.

All the three functions accept one argument, i.e., pointer to the string that needs to be converted. `atoi` and `atol` return an integer value and a long integer value, respectively, corresponding to the string entered. Thus `"1234"`, which is a string of 4 characters, is converted to `1234`, a single `int` or a `long` value (depending upon whether `atoi` or `atol` function was used). These functions ignore leading blank spaces, process a leading algebraic sign (`+` or `-`), if any, and process digits up to the first non-numeric digit. Thus, `"34+25"` is converted to `34`. `atof` function can accept strings with decimal points, preceding algebraic signs and representations in scientific notation (e.g., `"1.6e-9"`), and converts them into floating point numbers.

Consider the following example. It accepts an integer, a long integer and a floating point number as ASCII strings. Then it uses these functions to convert them into their respective types. The returned values are assigned to suitable variables and the values of these variables are then printed.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
       char Integer[10];
       char LongInteger[10];
       char FloatingNumber[10];

       int AnInt;
       long ALong;
       float AFloat;

       printf("Enter an integer :: ");
       gets(Integer);
       printf("Enter a long integer :: ");
       gets(LongInteger);
       printf("Enter a floating point number :: ");
       gets(FloatingNumber);

       AnInt = atoi(Integer);
       ALong = atol(LongInteger);
       AFloat = atof(FloatingNumber);

       printf("Converted Values are %d :: %ld :: %e\n",AnInt,ALong,AFloat);
       return(0);
}
```

A typical execution session of this program is shown below,

```
Enter an integer :: -15
Enter a long integer :: 76549
Enter a floating point number :: -3.3e-16
Converted Values are -15 :: 76549 :: -3.299999e-16
```

Conversion can also be accomplished in the reverse direction, i.e., numerical values can be transformed into null terminated strings. Three functions, `itoa`, `ltoa` and `gcvt` can be used to perform the conversion to null terminated strings from integers, long integer and floating point numbers, respectively.

`itoa` function call takes three arguments. The first is the integer value that is to be converted to a null terminated string, the second is the pointer to the string where the

string representation of the integer value has to be stored, and the third is the radix, i.e, number system in which the value should be represented as a string. `ltoa` function call also takes 3 argument. In this case, the second and the third arguments are the same as for the `itoa` function, but the first argument is a long integer value. If the value of `radix` is 10, the string contains the decimal representation of the value of the first argument. Alternatively, if the value of `radix` is 16, the string contains the hexadecimal representation of the value of the first argument.

`gcvt` function call also takes three arguments. The first argument is the floating point value that has to be converted, the second argument is the number of digits that need to be represented and the third argument is the pointer to the string where the string representation of the floating point number has to be stored. The value of the floating point number in the first argument is rounded off so that it can be accommodated in the number of digits specified in the second argument.

The following program demonstrates the application of these functions,

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int AnInt;
    long ALong;
    float AFloat;

    char Integer10[10];
    char Integer16[10];
    char Long10[10];
    char Long16[10];
    char FloatString[15];

    printf("Enter an integer, a long integer and a floating point number\n");
    printf("separated by spaces :: ");
    scanf("%d %ld %f",&AnInt,&ALong,&AFloat);

    itoa(AnInt,Integer10,10);
    itoa(AnInt,Integer16,16);
    printf("%d as a string at base 10 is %s and at base 16 is %s\n",
             AnInt,Integer10,Integer16);

    ltoa(ALong,Long10,10);
    ltoa(ALong,Long16,16);
    printf("%ld as a string at base 10 is %s and at base 16 is %s\n",
            ALong,Long10,Long16);

    gcvt(AFloat,8,FloatString);
    printf("%f as a string is %s\n",AFloat,FloatString);
    return(0);
}
```

In this program, the function call `itoa(AnInt,Integer10,10);` converts the decimal value of `AnInt` into its string representation because the `radix`, i.e., the third argument, is specified as 10. The resultant string representation is stored in the location pointed to by `Integer10`. The function call `itoa(AnInt,Integer16,16);` converts the hexadecimal value of `AnInt` into its string representation as the radix is specified as 16. The resultant string is saved in the location pointed to by `Integer16`. The function calls `ltoa(ALong,Long10,10);` and `ltoa(ALong,Long16,16);` perform the same operations on the value of `ALong` and store the resultant strings in location pointed to by `Long10` and `Long16`. The function call `gcvt(AFloat,8,FloatString);` converts the 8 most significant digits of the value of `AFloat` to a string representation and store that string in the location pointed to by `FloatString`.

A typical execution session of this program is given below,

```
Enter an integer, a long integer and a floating point number
separated by spaces :: 56 65531 -10.987654
56 as a string at base 10 is 56 and at base 16 is 38
65531 as a string at base 10 is 65531 and at base 16 is fffb
-10.987654 as a string is -10.987654
```

Structures

7.1 Introduction

Simple variables can hold one piece of information at a time and arrays can hold a number of pieces of information of the same data type. These two data storage mechanisms can handle a great variety of data manipulation situations. But often programs, particularly data processing programs, need to operate on data items of different types packaged together as a single unit. In this case, neither a variable nor an array is adequate.

For instance, consider a program that is supposed to manage data of students in a course. Typical information elements for a specific student would include name, domicile of the province and the student's age. It may be desirable to package these information elements so that they may be managed and processed efficiently and conveniently. For a number of students, an array of such an information package may be required.

Even multidimensional arrays cannot solve this problem, since all the elements of an array must be of the same data type. It may be possible for the programmer to use several different arrays, a string array for names, another string array for course and an integer array for age. Consider the following example,

```
#include <stdio.h>
#include <string.h>

void enterData(char Names[][40],char Domicile[][40],int *Age);
void findNShow(char Names[][40],char Domicile[][40],int *Age,char *Query);

int main(void)
{
	char Names[5][40];
	char Domicile[5][40];
	int Age[5];
	char Query[40];
	enterData(Names,Domicile,Age);
	do
	{
		printf("Enter name to search, * to exit to operating system : ");
		gets(Query);
		findNShow(Names,Domicile,Age,Query);
	}
```

```
	while(strcmp(Query,"*") != 0);
	return(0);
}

void findNShow(char Names[][40],char Domicile[][40],int Age[5],char *Query)
{
	int Found = 0;
	int i = 0;
	while((! Found) && (i < 5))
	{
		if (strcmp(Names[i],Query) == 0)
		{
			printf("Name	: %s\n",Names[i]);
			printf("Domicile	: %s\n",Domicile[i]);
			printf("Age	: %d\n",Age[i]);
			Found = 1;
		}
		else
		{
			i++;
		}
	}
	if (! Found)
	{
		printf("No matching record for Name = %s\n",Query);
	}
}

void enterData(char Names[][40],char Domicile[][40],int Age[5])
{
	int i;
	printf("Entering Data\n");
	printf("======== ====\n");
	for(i=0;i<5;i++)
	{
		printf("Entering data for person %d\n",i);
		printf("Name	: ");
		gets(Names[i]);
		printf("Domicile : ");
		gets(Domicile[i]);
		printf("Age	: ");
		scanf("%d",&Age[i]);
		getchar();
		printf("\n\n");
	}
}
```

This program allows the user to enter data (names, domicile and ages) for five students. The individual information elements gets stored in arrays `Names`, `Domicile` and `Age`. The user can search for a record by specifying the full name of the student. A typical execution session of this program is shown below,

```
Entering Data
======== ====
Entering data for person 0
Name	: Omar Bashir
Domicile : Punjab
Age	: 32
```

```
Entering data for person 1
Name     : Salman Khalid
Domicile : Sind
Age      : 30

Entering data for person 2
Name     : Waqar Haider
Domicile : NWFP
Age      : 33

Entering data for person 3
Name     : Amir Abbas
Domicile : Federal
Age      : 34

Entering data for person 4
Name     : Qamar Abbas
Domicile : Baluchistan
Age      : 35

Enter name to search, * to exit to operating system : Amir Abbas
Name     :  Amir Abbas
Domicile :  Federal
Age      :  34
Enter name to search, * to exit to operating system : Amir Kaleem
No matching record for Name = Amir Kaleem
Enter name to search, * to exit to operating system : Qamar Abbas
Name     : Qamar Abbas
Domicile : Baluchistan
Age      : 35
Enter name to search, * to exit to operating system : *
```

This approach does solve the problem of managing several different data elements of potentially different data types used to define an entity in a program. However, this approach is cumbersome to implement and maintain. For instance, if the number of information elements describing a student increase, the number of arguments being passed to each function increase as well.

To solve such problems, C provides a special data type known as the ***structure***. A structure consists of a number of data items grouped together. These data items need not be of the same type. Moreover structures are used as the basis of more complex data types, such as the linked lists, queues and stacks.

7.2 Basics of Structures

To understand the basic concepts used in defining and using structures, a simple example is given below,

```
#include <stdio.h>

struct Student
{
      char Name[25];
      char Domicile[25];
      int Age;
};

int main(void)
{
      struct Student AStudent;
      printf("Enter Name      : ");
      gets(AStudent.Name);
      printf("Enter Domicile  : ");
      gets(AStudent.Domicile);
      printf("Enter Age       : ");
      scanf("%d",&(AStudent.Age));

      printf("\n\nStudent Details\n");
      printf("======= =======\n");
      printf("Name     : %s\n",AStudent.Name);
      printf("Domicile : %s\n",AStudent.Domicile);
      printf("Age      : %d\n",AStudent.Age);
      return(0);
}
```

The program declares a simple structure `Student`, which contains three data items. Two of these are arrays of characters and one is an integer variable. Data items in a structure may be referred to as its ***members*** or ***fields***. The `main` function declares a variable, `AStudent`, of this structure and allows the user to assigns values to the fields of this structure. The `main` function then prints the value of each field of the structure variable.

A typical execution session of this program is shown below,

```
Enter Name      : Saif Imran
Enter Domicile  : NWFP
Enter Age       : 28


Student Details
======= =======
Name      : Saif Imran
Domicile  : NWFP
Age       : 28
```

This program highlights the three fundamental aspects of using structures,

1. declaring the structure type or template,
2. defining structure variables,

3. accessing members or fields of the structure.

7.2.1 Declaring a Structure Type or Template

The fundamental data types used in C, such as `int`, `long` and `float`, are predefined by the compiler. Thus, all variables of type `long` will always consist of 4 bytes, and the compiler will always interpret the contents of these four bytes in a certain way. However, this is not true for a structure, as a structure is a data type whose format is defined by the programmer. The programmer must, therefore, specify to the compiler, the number and types of fields a structure is composed of. This specification or declaration should be made any variable of that structure type is used.

In the above mentioned example, the following statement declares the structure type,

```
struct Student
{
      char Name[25];
      char Domicile[25];
      int Age;
};
```

This statement declares a new data type called `struct Student`. Each variable of this type will consist of three members, an array of characters called `Name`, another array of characters called `Domicile` and an integer variable called `Age`. Note that this statement does not define any variables and, therefore, it does not set aside any storage in memory. It merely informs the compiler what is the format of the data type `struct Student`. Thus, this is also known as the structure template or a structure plan.

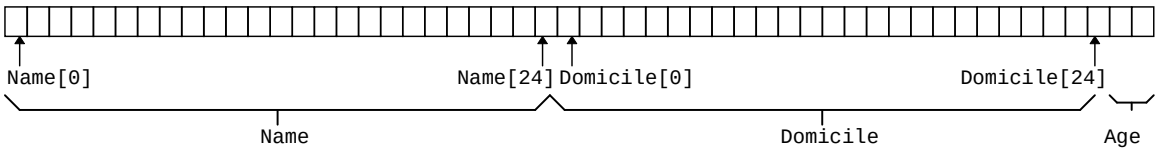
The keyword `struct` introduces this declaration. The name `Student` is called the ***tag***. It names the kind of structure being defined. *It should be noted that a tag is a type name and not a variable name*. The members of the structure are surrounded by braces and the entire statement is terminated by a semicolon.

7.2.2 Defining Structure Variables

Structure type declaration merely specifies to the compiler the format and the size of a structure variable. With a structure type declaration, a compiler does not actually allocate any memory for structure variables. For the above mentioned example, a structure variable is defined as shown below,

```
struct Student AStudent;
```

This statement causes the computer to create a variable `AStudent`. A specified by the structure declaration, space is allocated for two character arrays of 25 bytes each and an integer variable of 2 bytes. This package is named as `AStudent`. Figure 7-1 shows the memory allocation for the structure.




```
struct Student
{
      char Name[25];
      char Domicile[25];
      int Age;
};
```

Figure 7-1 : Memory Allocation for a Structure

In this declaration, `struct Student` plays the same role that `int` or `float` does in a declaration. For example, two variables of the `struct Student` type can be declared, or even a pointer to that kind of structure,

```
struct Student AStudent, BStudent, *StudentPtr;
```

The structure variables `AStudent` and `BStudent` would each have their own `Name`, `Domicile` and `Age` members. The pointer `StudentPtr` can point to `AStudent`, `BStudent` or any other `Student` structure.

7.2.3 Accessing Structure Members

Structures use the ***dot operator*** (also known as the ***membership operator***), ".". For instance, the expression `AStudent.Age` provides access to the `Age` member of the structure `AStudent`. `Astudent.Age` can be used exactly like any other integer variable. Similarly, `AStudent.Name` can be used exactly like any array of the type `char`. For instance, the statement `printf("Age : %d\n",AStudent.Age);` prints the value of the `Age` member of structure `AStudent`. Similarly the statement `scanf("%d",&(AStudent.Age));` requires the address of a location of an integer, which is provided by the expression `&(AStudent.Age)`.

If there are two variables of the type `Student`, then the following statement would print the value of the member `Age` for each of these structures,

```
printf("Age1 : %d\nAge2 : %d\n ",AStudent.Age,BStudent.Age);
```

Similarly, the following statement would allow a user to enter values for member `Age` for these two variables of type `Student`,

```
scanf("%d %d",&AStudent.Age,&BStudent.Age);
```

The dot operator has higher precedence than the address operator. Therefore, the expression `&AStudent.Age` is the same as the expression `&(AStudent.Age)`.

7.3 Initialising Structures

Like simple variables and arrays, structure variables can be initialised, i.e., given specific values, at the beginning of the program. The format used is quite similar to that used to initialise arrays. A structure variable in a declaration can be initialised by following it with an assignment operator and a comma separated list of constants contained within braces. If not enough values are used to assign all the members of the structure, the remaining members are assigned the value zero by default.

Consider the following version of the previous example. In this example, the variable `AStudent` of the type `Student` is initialised at declaration.

```
#include <stdio.h>

struct Student
{
      char Name[25];
      char Domicile[25];
      int Age;
};

int main(void)
{
      struct Student AStudent = {"Tahir Majeed","FATA",29};

      printf("\n\nStudent Details\n");
      printf("======= =======\n");
      printf("Name     : %s\n",AStudent.Name);
      printf("Domicile : %s\n",AStudent.Domicile);
      printf("Age      : %d\n",AStudent.Age);
      return(0);
}
```

The `Name` member of the structure `AStudent` is initialised to `"Tahir Majeed"`, the `Domicile` member is initialised to `"FATA"` and the `Age` member is initialised to `29`. The output of this program is shown below,

```
Student Details
======= =======
Name     : Tahir Majeed
Domicile : FATA
Age      : 29
```

7.4 Assigning a Structure Variable to Another Structure Variable of Same Type

In the original C language, it was impossible to assign values of one structure variable to another variable using a simple assignment statement. Thus, value of each element of one structure variable (i.e., the source) had to be assigned to the corresponding element of the second structure variable (i.e., the destination). In ANSI compatible versions of C, it is possible to assign values of one structure variable to another variable using a simple assignment statement.

Consider the following example. Here, two structures (`AStudent` and `BStudent`) of type `Student` are declared and initialised. Then the value of `AStudent` is assigned to `BStudent` using a simple assignment statement.

```
#include <stdio.h>

struct Student
{
      char Name[25];
      char Domicile[25];
      int Age;
};

int main(void)
{
      struct Student AStudent = {"Tahir Majeed","FATA",29};
      struct Student BStudent = {"Ahmed Karim","NWFP",28};

      printf("\n\nStudent Details\n");
      printf("======= =======\n");
      printf("AStudent : %s %s %d\n",AStudent.Name,AStudent.Domicile,
                                     AStudent.Age);
      printf("BStudent : %s %s %d\n",BStudent.Name,BStudent.Domicile,
                                     BStudent.Age);

      BStudent = AStudent;

      printf("\n\nStudent Details\n");
      printf("======= =======\n");
      printf("AStudent : %s %s %d\n",AStudent.Name,AStudent.Domicile,
                                     AStudent.Age);
      printf("BStudent : %s %s %d\n",BStudent.Name,BStudent.Domicile,
                                     BStudent.Age);

      return(0);
}
```

Output of this program is given below,

```
Student Details
======= =======
AStudent : Tahir Majeed FATA 29
BStudent : Ahmed Karim NWFP 28


Student Details
======= =======
AStudent : Tahir Majeed FATA 29
BStudent : Tahir Majeed FATA 29
```

The statement `BStudent = AStudent;` causes the values of all the members of `AStudent` to be assigned to the corresponding members of `BStudent`.

7.5 Nested Structures

Just as there can be arrays of arrays, there can also be structures that contain other structures. This allows a powerful way to create complex data types. The structure that contains other structures will be referred to as ***composite structure*** whereas the structure contained within composite structure is referred to as the ***component structure***.

Consider the following version of the initial example. In this example, the structure `Student`, instead of having member `Age`, has a variable of another structure `DateOfBirth` as its member. The structure `DateOfBirth` has three integers as its members representing day, month and year.

```
#include <stdio.h>

struct DateOfBirth
{
       int Year;
       int Month;
       int Day;
};

struct Student
{
       char Name[25];
       char Domicile[25];
       struct DateOfBirth BirthDay;
};

int main(void)
{
       struct Student AStudent;
       printf("Enter Name                      : ");
       gets(AStudent.Name);
       printf("Enter Domicile                  : ");
       gets(AStudent.Domicile);
       printf("Enter Date of Birth (dd/mm/yyyy) : ");
       scanf("%d/%d/%d",&(AStudent.BirthDay.Day),&(AStudent.BirthDay.Month),
                        &(AStudent.BirthDay.Year));

       printf("\n\nStudent Details\n");
       printf("======= =======\n");
       printf("Name          : %s\n",AStudent.Name);
       printf("Domicile      : %s\n",AStudent.Domicile);
       printf("Date of Birth : %d/%d/%d\n",AStudent.BirthDay.Day,
                                           AStudent.BirthDay.Month,
                                           AStudent.BirthDay.Year);
       return(0);
}
```

In this example, to access a member of the component structure, the dot operator has to be used twice. First to access the component structure within the variable of the composite structure and the second to access the specific variable of the component structure. For instance, the expression `AStudent.BirthDay.Year` provides access to the `variable Year` which is the member of the structure variable `BirthDay` which in turn is a member of the structure variable `AStudent`. A typical execution session of this program is given below,

```
Enter Name                      : Amir Ali
Enter Domicile                  : Kashmir
Enter Date of Birth (dd/mm/yyyy) : 23/8/1966


Name          : Amir Ali
Domicile      : Kashmir
Date of Birth : 23/8/1966
```

The following example is another version of this program, where the composite structure variable is initialised upon declaration.

```
#include <stdio.h>

struct DateOfBirth
{
      int Year;
      int Month;
      int Day;
};

struct Student
{
      char Name[25];
      char Domicile[25];
      struct DateOfBirth BirthDay;
};

int main(void)
{
      struct Student AStudent = {"Amir Ali","Kashmir",{1966,8,23}};

      printf("\n\nStudent Details\n");
      printf("======= =======\n");
      printf("Name          : %s\n",AStudent.Name);
      printf("Domicile      : %s\n",AStudent.Domicile);
      printf("Date of Birth : %d/%d/%d\n",AStudent.BirthDay.Day,
                                          AStudent.BirthDay.Month,
                                          AStudent.BirthDay.Year);
      return(0);
}
```

The output of this program is given below,

```
Student Details
======= =======
Name          : Amir Ali
Domicile      : Kashmir
Date of Birth : 23/8/1966
```

The statement

`struct Student AStudent = {"Amir Ali","Kashmir",{1966,8,23}};`

initialises the respective members of the variable `AStudent`. Note that nested braces contain the values used to initialise the members of the component structure.

7.6 Arrays of Variables of a Structure

Consider the following program. This program uses a array of variables of the structure `Student`.

```
#include <stdio.h>

struct DateOfBirth
{
       int Year;
       int Month;
       int Day;
};

struct Student
{
       char Name[25];
       char Domicile[25];
       struct DateOfBirth Birthday;
};

int main(void)
{
       struct Student Class[5];
       int i;
       /* Entering data */
       for (i=0;i<5;i++)
       {
              printf ("Entering data for student[%d]\n",i);
              printf ("Enter name                  : ");
              gets   (Class[i].Name);
              printf ("Enter domicile              : ");
              gets   (Class[i].Domicile);
              printf ("Enter date of birth [d/m/y] : ");
              scanf  ("%d/%d/%d",&Class[i].Birthday.Day,
                                 &Class[i].Birthday.Month,
                                 &Class[i].Birthday.Year);
              getchar();
```

```
        }

        printf ("============****==============\n");

        /* Printing Data */
        for (i=0;i<5;i++)
        {
              printf ("STUDENT[%d] :: ",i);
              printf ("%s of %s born on %d/%d/%d\n",Class[i].Name,
                                                    Class[i].Domicile,
                                                    Class[i].Birthday.Day,
                                                    Class[i].Birthday.Month,
                                                    Class[i].Birthday.Year);
        }

        return(0);
}
```

The statement

```
struct Student Class[5];
```

declares an array `Class` of five elements of the type `struct Student`. Each element of the array `Class` is a structure of `Student` type. Thus `Class[0]` is a structure of type `Student` and is the first element of the array `Class`, and `Class[1]` is a structure of type `Student` and is the second element of the array `Class`. This can be visualised as shown in figure 2.

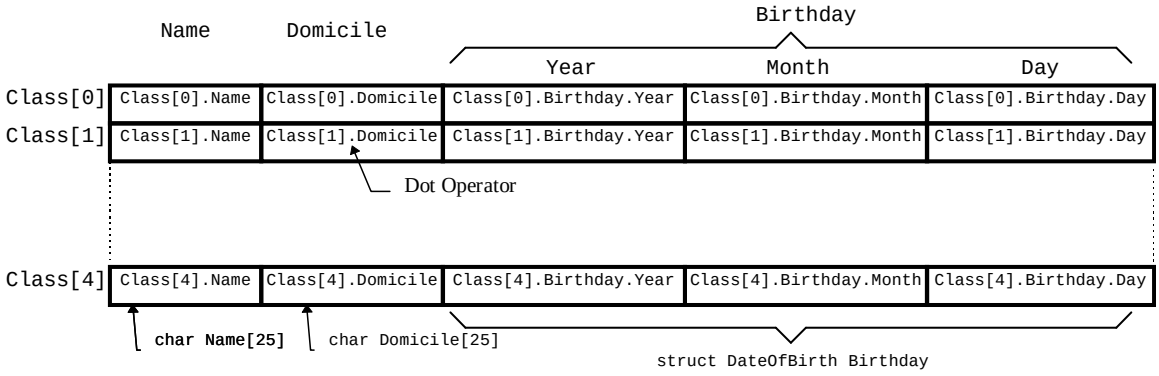


Figure - 2 : Declaration `struct Student Class[5]`

Members of an array of structures are identified by applying the same rule as used for individual structures, i.e., referring to the structure variable name, followed by a subscript, followed by the dot operator and ending with the name of the desired structure element. Thus `Class[2].Name` gives access the element `Name` of the third variable in the array `Class`, where `Class[2]` is a variable name, just as `Class[1]` is another variable name.

7.7 Pointers to Variables of a Structure

Pointers can also be used to contain the addresses of structures. Consider the following example,

```
#include <stdio.h>
#include <string.h>

struct Date
{
 int Year;
 int Month;
 int Day;
};

struct Student
{
 char FirstName[25];
 char SecondName[25];
 struct Date DateOfBirth;
};

int main(void)
{
 struct Student AStudent;
 struct Student *PtrStudent;
 strcpy(AStudent.FirstName,"Farooq");
 strcpy(AStudent.SecondName,"Omar");
 AStudent.DateOfBirth.Day = 26;
 AStudent.DateOfBirth.Month = 11;
 AStudent.DateOfBirth.Year  = 1965;

 PtrStudent = &AStudent;
 printf("STUDENT DETAILS\n======= =======\n");
 printf("NAME : %s %s\n",PtrStudent->FirstName,PtrStudent->SecondName);
 printf("DoB  : %d/%d/%d\n",PtrStudent->DateOfBirth.Day,
                            PtrStudent->DateOfBirth.Month,
                            PtrStudent->DateOfBirth.Year);
 getchar();
 return(0);
}
```

The statement `struct Student AStudent;` creates a structure variable `AStudent` of the type `struct Student` whereas the statement `struct Student *PtrStudent;` creates a pointer `PtrStudent` to a variable of the type `struct Student`. The statement

```
PtrStudent = &AStudent;
```

assigns the address of the variable `AStudent` of the structure to the pointer variable `PtrStudent`. Elements of the variable of structure pointed to by a pointer can be accessed by using the ***arrow operator*** or the ***indirect membership operator*** (`->`). This operator is

formed by typing a hyphen (-) followed by the greater than symbol (>). A pointer to a structure followed by the arrow operator works the same as a structure name followed by the dot operator. Thus `AStudent.FirstName` is the same as `PtrStudent->FirstName`. It is important to remember that `PtrStudent` is a pointer but `FirstName` is a member of the structure pointed to.

Similarly `DateOfBirth` is a variable of the structure `Date` and is, therefore, a member of the structure pointed to by `PtrStudent`. Thus, the expression `PtrStudent->DateOfBirth.Year` provides the value of the `Year` element of the structure variable `DateOfBirth`, which is a member of the structure pointed to by `PtrStudent`.

In addition to the arrow operator, the indirection operator can be used in conjunction with the dot operator to access individual members of a structure variable pointed to. For instance, `PtrStudent->DateOfBirth.Year` can be replaced by the expression `(*PtrStudent).DateOfBirth.Year`. Parenthesis are needed around `*PtrStudent` as the dot operator has a higher precedence than the indirection operator and will operate in the expression before the indirection operator.

Thus, the example shown above can be re-written as follows,

```
#include <stdio.h>
#include <string.h>

struct Date
{
 int Year;
 int Month;
 int Day;
};

struct Student
{
 char FirstName[25];
 char SecondName[25];
 struct Date DateOfBirth;
};

int main(void)
{
 struct Student AStudent;
 struct Student *PtrStudent;
 strcpy(AStudent.FirstName,"Farooq");
 strcpy(AStudent.SecondName,"Omar");
 AStudent.DateOfBirth.Day = 26;
```

```
 AStudent.DateOfBirth.Month = 11;
 AStudent.DateOfBirth.Year  = 1965;

 PtrStudent = &AStudent;
 printf("STUDENT DETAILS\n======= =======\n");
 printf("NAME : %s %s\n",(*PtrStudent).FirstName,(*PtrStudent).SecondName);
 printf("DoB  : %d/%d/%d\n",(*PtrStudent).DateOfBirth.Day,
                           (*PtrStudent).DateOfBirth.Month,
                           (*PtrStudent).DateOfBirth.Year);
 getchar();
 return(0);
}
```

The output of both the programs is

```
STUDENT DETAILS
======= =======
NAME : Farooq Omar
DoB  : 26/11/1965
```

7.8 Communicating Variables of a Structure to Functions

In C, structures can be passed as arguments to functions and can be returned from them.

Consider the following example,

```
#include <stdio.h>

struct Student
{
      char Name[50];
      int  RollNumber;
      int  Class;
      char Section;
};

struct Student getData(void);
void printData(struct Student);

int main(void)
{
      struct Student AStudent;
      AStudent = getData();
      printData(AStudent);
      return(0);
}

void printData(struct Student StudentData)
{
      printf("*************************************************\n");
      printf("                      Student Data\n");
      printf("*************************************************\n");
      printf("Name        : %s\n",StudentData.Name);
      printf("Roll Number : %d\n",StudentData.RollNumber);
      printf("Class       : %d\n",StudentData.Class);
      printf("Section     : %c\n",StudentData.Section);
```

```
      printf("************************************************\n");
}

struct Student getData(void)
{
      struct Student StudentData;
      printf("Enter student's name         : ");
      gets(StudentData.Name);
      printf("Enter student's roll number  : ");
      scanf("%d",&(StudentData.RollNumber));
      getchar();
      printf("Enter student's class        : ");
      scanf("%d",&(StudentData.Class));
      getchar();
      printf("Enter student's section      : ");
      StudentData.Section = getchar();
      return(StudentData);
}
```

A variable of the structure `Student` (`AStudent`) is declared in function `main`. The members of the structure are initialised by the structure returned by the function `getData`. In function `getData`, another variable of the structure `Student` (`StudentData`) is declared. Subsequent statements prompt the user to enter data from the keyboard to initialise the respective fields of the structure. In the end the structure is returned to the calling function.

In function `main`, the function `printData` is then called. This function call takes as an argument a variable of the structure `Student`. The values of the fields of the structure passed in the function call are copied to the respective fields of another variable (`StudentData`) of the same structure `Student` defined as the formal argument. The `printData` function then prints the values of the respective fields of the structure variable `StudentData`. A typical session of this program produces the following output on the display,

```
Enter student's name         : Shameer Omar
Enter student's roll number  : 32
Enter student's class        : 2
Enter student's section      : b
************************************************
                  Student Data
************************************************
Name        : Shameer Omar
Roll Number : 32
Class       : 2
Section     : b
```

```
************************************************
```

When a structure is passed as an argument to a function, it is passed by value. This means that a local copy is made for use in the body of the function. If a member of the structure is an array, then the array gets copied as well. If the structure has many members or members that are large arrays, then passing a structure as an argument can be inefficient. For most applications, functions are written so as to take the address of the structure as an argument instead. Consider the following version of the program written above,

```
#include <stdio.h>

struct Student
{
      char Name[50];
      int  RollNumber;
      int  Class;
      char Section;
};

void getData(struct Student*);
void printData(struct Student*);

int main(void)
{
      struct Student AStudent;
      getData(&AStudent);
      printData(&AStudent);
      return(0);
}

void printData(struct Student *StudentData)
{
      printf("************************************************\n");
      printf("                      Student Data\n");
      printf("************************************************\n");
      printf("Name        : %s\n",StudentData->Name);
      printf("Roll Number : %d\n",StudentData->RollNumber);
      printf("Class       : %d\n",StudentData->Class);
      printf("Section     : %c\n",StudentData->Section);
      printf("************************************************\n");
}

void getData(struct Student *StudentData)
{
      printf("Enter student's name         : ");
      gets(StudentData->Name);
      printf("Enter student's roll number  : ");
      scanf("%d",&(StudentData->RollNumber));
      getchar();
      printf("Enter student's class        : ");
      scanf("%d",&(StudentData->Class));
      getchar();
      printf("Enter student's section      : ");
```

```
      StudentData->Section = getchar();
}
```

In the program shown above, only one variable of the structure `Student` (`AStudent`) is declared in the entire program. Function `main` passes the pointer to `AStudent` to the functions `getData` and `printData`. Therefore, these functions, when in execution, are actually accessing the fields of the same structure variable (`AStudent`) declared in function `main`. This eliminates the overhead associated with local copies of structures created when the structures are passed as arguments to functions.

7.9 Dynamic Memory Allocation

In the following example, an array of structure variables is used to hold the details of students on a course.

```
#include <stdio.h>

#define MAX_STUDENTS 5

struct Date
{
   int Year;
   int Month;
   int Day;
};

struct Student
{
   char Name[25];
   char RollNumber[25];
   struct Date DateOfBirth;
};

int  AddAStudent(struct Student *StudentList,int Count);
void EnterStudentDetails(struct Student *AStudent);
void ListStudents(struct Student *StudentList,int Count);
void ShowOneStudent(struct Student *AStudent);

int main(void)
{
   struct Student Students[MAX_STUDENTS];
   int Count = 0;
   char Choice;
   do
   {
      printf("STUDENTS DATABASE\n");
      printf("=================\n");
      printf("1. Add a student\n");
      printf("2. List all students\n");
      printf("9. Exit\n");
      Choice = getchar();
      getchar();
      switch (Choice)
      {
         case '1':
            if (Count < MAX_STUDENTS)
            {
               Count = AddAStudent(Students,Count);
```

```
            }
            else
            {
               puts("No more space");
            }
            break;
         case '2':
            ListStudents(Students,Count);
            break;
         case '9':
            break;
         default:
            puts("Command Not Recognised");
      }
   }
   while (Choice != '9');
   return(0);
}

int AddAStudent(struct Student *StudentList,int Count)
{
   EnterStudentDetails(StudentList+Count);
   Count++;
   return(Count);
}

void ListStudents(struct Student *StudentList,int Count)
{
   int i;
   if (Count <= 0)
   {
      printf("List is empty\n");
   }
   else
   {
      for (i = 0;i < Count;i++)
      {
         ShowOneStudent(StudentList+i);
         printf("Press ENTER to continue...");
         getchar();
      }
   }
}

void ShowOneStudent(struct Student *AStudent)
{
   printf("NAME           : %s\n",AStudent->Name);
   printf("Roll Number    : %s\n",AStudent->RollNumber);
   printf("Date of Birth  : %02d/%02d/%04d\n", AStudent->DateOfBirth.Day,

AStudent->DateOfBirth.Month,

AStudent->DateOfBirth.Year);
}

void EnterStudentDetails(struct Student *AStudent)
{
   printf("Enter Name                       : ");
   gets(AStudent->Name);
   printf("Enter Roll Number                : ");
   gets(AStudent->RollNumber);
   printf("Enter Date of Birth (dd/mm/yyyy) : ");
   scanf("%d/%d/%d",&(AStudent->DateOfBirth.Day),
                    &(AStudent->DateOfBirth.Month),
                    &(AStudent->DateOfBirth.Year));
   getchar();
}
```

The statement `struct Student Students[MAX_STUDENTS]` creates an array of elements of the structure `Student`. As the value of `MAX_STUDENTS` is 5 and the size of each element of the

array is 56 bytes, this statement allocates 280 bytes even when data for only one or two students is entered in the array. If the size of the array is increased to 100 elements, then upon the execution of the program, 5600 bytes are allocated for the array even when data is not entered for a single record.

In such cases it is desirable to use dynamic memory allocation techniques. These techniques allow the allocation of required amount of memory during program execution. C provides two functions for dynamic memory allocation. `malloc()` (memory allocation) is used to allocate memory of specified size whereas `calloc()` (contiguous allocation) allocates contiguous space in memory for an array of specified number of elements. Both these functions are implemented in the standard library and their prototypes are specified in `stdlib.h`.

(a) `malloc`

`malloc` takes in a single argument of the type `size_t` and returns a pointer of type `void *` if the call is successful and memory has been allocated. If the call in unsuccessful and memory is not allocated, this function returns `NULL`. Allocated memory is not initialised. The prototype of the function is given below,

```
void* malloc(size_t);
```

Consider the following version of the program shown above.

```
#include <stdio.h>
#include <alloc.h>

#define MAX_STUDENTS 5

struct Date
{
   int Year;
   int Month;
   int Day;
};

struct Student
{
   char Name[25];
```

```
   char RollNumber[25];
   struct Date DateOfBirth;
};

void InitialiseArray(struct Student *StudentList[]);
void DeleteList(struct Student *StudentList[]);
void DeleteAStudent(struct Student *StudentList[]);
void AddAStudent(struct Student *StudentList[]);
void EnterStudentDetails(struct Student *AStudent);
void ListStudents(struct Student *StudentList[]);
void ShowOneStudent(struct Student *AStudent);

int main(void)
{
   struct Student *Students[MAX_STUDENTS];
   char Choice;
   InitialiseArray(Students);
   do
   {
      printf("STUDENTS DATABASE (uses malloc)\n");
      printf("==============================\n");
      printf("1. Add a student\n");
      printf("2. List all students\n");
      printf("3. Delete a student\n");
      printf("9. Exit\n");
      Choice = getchar();
      getchar();
      switch (Choice)
      {
        case '1':
             AddAStudent(Students);
             break;
        case '2':
             ListStudents(Students);
             break;
        case '3':
             DeleteAStudent(Students);
             break;
        case '9':
             DeleteList(Students);
             break;
        default:
             puts("Command Not Recognised");
      }
   }
   while (Choice != '9');
   return(0);
}

void InitialiseArray(struct Student* List[])
{
   int i;
   for (i=0;i < MAX_STUDENTS;i++)
   {
      List[i] = NULL;
   }
}

void DeleteList(struct Student* List[])
{
   int i;
```

```
  for (i=0;i < MAX_STUDENTS;i++)
  {
    if (List[i] != NULL)
    {
      free(List[i]);
      List[i] = NULL;
      printf("Location %d deleted\n",i);
    }
    else
    {
      printf("Location %d not occupied\n",i);
    }
  }
}

void AddAStudent(struct Student *StudentList[])
{
  int i = 0;
  while((i < MAX_STUDENTS) && (StudentList[i] != NULL))
  {
    i++;
  }
  if (i < MAX_STUDENTS)
  {
    printf("Entering student details at %d\n",i);
    StudentList[i] = (struct Student*)
                     malloc(sizeof(struct Student));
    if (StudentList[i] != NULL)
    {
      EnterStudentDetails(StudentList[i]);
      printf("Student details entered at %d\n",i);
    }
    else
    {
      printf("LESS PHYSICAL MEMORY SPACE\n");
    }
  }
  else
  {
    printf("No more space\n");
  }
}

void ListStudents(struct Student *StudentList[])
{
  int i;
  for (i = 0;i < MAX_STUDENTS;i++)
  {
    printf("Location # : %d\n",i);
    ShowOneStudent(StudentList[i]);
    printf("Press ENTER to continue...");
    getchar();
  }
}

void ShowOneStudent(struct Student *AStudent)
{
  if (AStudent != NULL)
  {
    printf("NAME           : %s\n",AStudent->Name);
    printf("Roll Number    : %s\n",AStudent->RollNumber);
```

```
    printf("Date of Birth  : %02d/%02d/%04d\n",
            AStudent->DateOfBirth.Day,
            AStudent->DateOfBirth.Month,
            AStudent->DateOfBirth.Year);
  }
  else
  {
    printf("Location empty\n");
  }
}

void DeleteAStudent(struct Student* List[])
{
  int Index;
  ListStudents(List);
  printf("Enter Index to Delete : ");
  scanf("%d",&Index);
  getchar();
  if ((Index < 0) || (Index >= MAX_STUDENTS))
  {
    printf("Index Out of Range\n");
  }
  else
  {
    if (List[Index] == NULL)
    {
      printf("Location %d already empty\n",Index);
    }
    else
    {
      free(List[Index]);
      List[Index] = NULL;
      printf("Location %d deleted\n",Index);
    }
  }
}

void EnterStudentDetails(struct Student *AStudent)
{
  printf("Enter Name                       : ");
  gets(AStudent->Name);
  printf("Enter Roll Number                : ");
  gets(AStudent->RollNumber);
  printf("Enter Date of Birth (dd/mm/yyyy) : ");
  scanf("%d/%d/%d",&(AStudent->DateOfBirth.Day),
                   &(AStudent->DateOfBirth.Month),
                   &(AStudent->DateOfBirth.Year));
  getchar();
}
```

In this program, only an array of pointers to structures is initialised, as opposed to an array of structure variables. Therefore, only the amount of memory required for the array of addresses is allocated. All unused elements of the pointer array are initialised to `NULL`. Once the user wishes to add details of a student in the program (in function `AddAStudent`), a structure variable is created dynamically (using

malloc) and its pointer is assigned to an unused pointer element of the array. Similarly, once the details of a student are to be removed from the program, the memory for the appropriate structure variable is deallocated and then the element in the pointer array pointing to that memory location is assigned a NULL. To release (or deallocate) memory dynamically allocated for a structure variable, function free is called (in function DeleteAStudent). The call for this function takes, as argument, pointer to the memory location that has to be released. If the pointer passed as argument to this function is pointing towards NULL, the function call has no effect.

(b) calloc

calloc takes two arguments, both of type size_t. In ANSI C, size_t must be an unsigned integral type. The first argument specifies the number of elements for which contiguous memory space has to be allocated, and the second argument specifies the size of each element. If the calloc function call is successful, a pointer of the type void* is returned that is pointing to the base of the allocated array in memory. If the calloc function call is unsuccessful, NULL is returned. The space is initialised with all bits set to 0.

Consider the following example,

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>

int   CreateArray(float **Array);
void  FillData(float *Array,int Size);
void  PrintData(float *Array,int Size);
float GetAverage(float *Array,int Size);
void pause(void);

int main(void)
{
  float *ArrayPointer = NULL;
  int   DataSize;
  float Average;
  DataSize = CreateArray(&ArrayPointer);
  if (DataSize > 0)
  {
    printf("Array of %d elements created\n",DataSize);
```

```
    FillData(ArrayPointer,DataSize);
    pause();
    PrintData(ArrayPointer,DataSize);
    pause();
    Average = GetAverage(ArrayPointer,DataSize);
    printf("Average = %9.2f\n",Average);
    free(ArrayPointer);
    pause();
  }
  else
  {
    printf("Array not created\n");
  }
  return(0);
}

void pause(void)
{
  puts("Press ENTER to continue...");
  getchar();
}

int CreateArray(float* *Array)
{
  int DataSize;
  printf("Enter data size : ");
  scanf("%i",&DataSize);
  getchar();
  *Array = (float*) calloc(DataSize,sizeof(float));
  if (*Array == NULL)
  {
    DataSize = 0;
  }
  return(DataSize);
}

void FillData(float *Array,int DataSize)
{
  int i;
  char DataString[15];
  for (i=0;i<DataSize;i++)
  {
    printf("Enter Data[%d] : ",i);
    gets(DataString);
    Array[i] = atof(DataString);
  }
}

void PrintData(float *Array,int DataSize)
{
  int i;
  for (i=0;i<DataSize;i++)
  {
    printf("Data[%d] = %9.2f\n",i,*(Array+i));
  }
}

float GetAverage(float *Array,int DataSize)
{
  float Sum = 0.0;
  float Average;
```

```
  int i;
  for (i=0;i<DataSize;i++)
  {
    Sum += *(Array+i);
  }
  Average = Sum/DataSize;
  return(Average);
}
```

The function `CreateArray` takes, as argument, a pointer to a pointer variable to the floating point variable. The user is asked to specify the size of the array of floating point numbers that is required. The function call `calloc(DataSize,sizeof(float))` attempts to create an array of number of elements specified by the `DataSize` variable where the size of each element is specified by the expression `sizeof(float)` and returns a pointer to memory created. This pointer is type casted to `float*` and is assigned to the pointer variable pointed to by the `Array` argument. Once the array has been created and the pointer to its first element assigned to a suitable pointer variable, its elements can be assigned values and these values accessed and processed as in normal arrays.

At the end of the program, the statement `free(ArrayPointer);` causes the space in memory pointed to by `ArrayPointer` to be deallocated and returned to the system.

The function call `malloc(DataSize*(sizeof(float)))` would also create a memory space of the same size as the function call `calloc(DataSize,sizeof(float))` would. However, the former does not initialise the space in memory that it makes available. If there is no reason to initialise the array to zero, then the use of either `malloc` or `calloc` is acceptable. In a large program, `malloc` may take less time.