

UNIT - I

1. OVERVIEW OF C

1.1 HISTORY OF C

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today because it is a structured, high-level, machine independent language. It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Jacopini and Edsger Dijkstra popularized this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. Both BCPL and B were "type less" system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating system in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C running under a variety of operating system and hardware platforms.

During 1970s, C had developed into what is now known as “traditional C”. The language became more popular after publication of the book ‘The C Programming Language’ by Brian Kernigham and Dennis Ritchie in 1978. The book was so popular that the language came to known as “K&R C” among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.

During 1990s, C++, a language entirely based on C, Underwent number of improvements and changes and became an ANSI/ISO approved language in November 1977. C++ added several new features to features to make it not only a true object-oriented language but also a more versatile language. During the same period, Sun Microsystems of USA created a new language JAVA modelled on C and C++.

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and JAVA were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig.1.1.

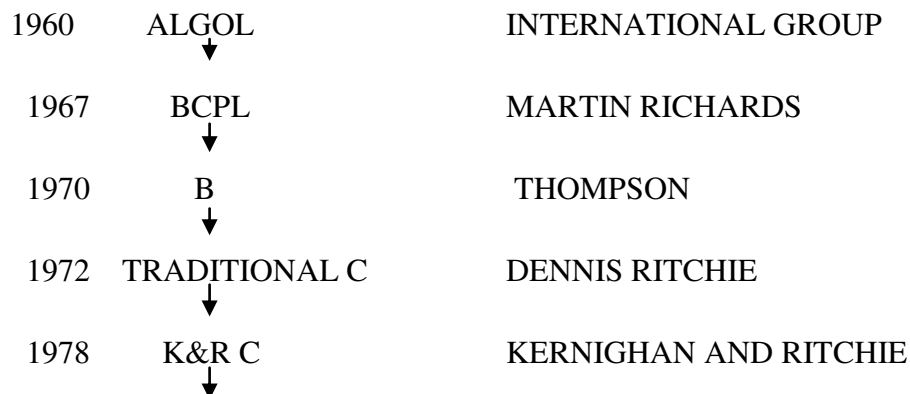




Fig 1.1 History of ANSI C

Although C99 is an improved version, still many commonly available compilers do not support all of the new features incorporated in C99. We therefore discuss all the new features added by C99 in an appendix separately so that the readers who are interested can quickly refer to the new material and use them wherever possible.

1.2 IMPORTANCE OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of assembly language with the features of high-level language and therefore it is well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C while it takes more than 50seconds in an interpreter BASIC.

There are only 32 keywords in ANSI C and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would

make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Before discussing specific features of C, we shall look at some sample C programs, and analyze and understand how they work.

1.3 Sample Program 1: Printing a Message

Consider a very simple program, given in fig1.2

```
main()
{
/*.....printing begins.....*/
    Printf("I see, I remember");
/*.....printing ends.....*/
}
```

Fig 1.2 A program to print one line of text

This program when executed will produce the following output:

I see, I remember

Let us have a close look at the program. The first line informs the system that the name of the program is **main** and execution begins at this line. The `main()` is a special function used by the C system to tell the computer where the program starts. Every program must have exactly one main function. If we use more than one **main** function, the compiler cannot understand which one marks the beginning of the program.

The empty pair of parentheses immediately following **main** indicates that the function `main` has no arguments (or parameters).

The opening brace “{” in the second line marks the beginning of the function **main** and the closing brace “}” in the last line indicates the end of the function. In this case, the closing brace

also marks the end of the program. All the statements between these two braces form the function body. The function contains a set of instructions to perform the given task

In this case, the function body contains three statements out of which only the **printf** line is an executable statement. The lines beginning with `/*` and ending with `*/` are known as comment lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between `/*` and `*/` is ignored by the compiler. In general, a comment can be inserted wherever blank spaces can occur—at the beginning, middle or end of a line —“but never in the middle of a word”.

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

```
/*=====/*=====*/=====*/
```

is not valid and therefore results in an error.

Since comments do not affect the execution speed and the size of a compiled program, we should use them liberally in our programs. They help programmers and other users in understanding the various functions and operations of program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the **printf()** function, the only executable statement of the program.

```
printf(“I see, I remember”);
```

Printf is a predefined standard C function for printing output. Predefined means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The concepts of compilation and linking are explained later in this chapter. The `printf` function causes everything between the starting and ending quotation marks to print out. In this case, the output will be:

```
I see remember, I remember
```

Note that the print line ends with a semicolon. Every statement in c should end with a semicolon (;) mark. Suppose we want to print the above quotation in two lines as

I see,

I remember!

This can be achieved by adding another printf function as shown below:

```
printf("I see,\n");
```

```
printf("I remember !");
```

The information contained between the parentheses is called the argument of the function. This argument of the first **printf** is "I see, \n" and the second is "I remember!" These arguments are simply strings of characters to be printed out.

Notice that the argument of the first printf contains a combination of two characters \ and n at the end of the string. This combination is collectively called the newline character. A newline character instructs the computer to go the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character \n causes the string "I remember!" to be printed on the next line. No space is allowed between \ and n.

If we omit the newline character from the first **printf** statement, then the output will again be a single line as shown below.

I see, I remember!

This is similar to the output of the program in fig 1.2 however; note that there is no space between, and I.

Note: some authors recommended the inclusion of the statement

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

At the beginning of all programs that use any input/output library functions. However, this is not

necessary for the functions **printf** and **scanf** which have been defined as a part of the C language. See chapter 4 for more on input and output functions.

Before we proceed to discuss further examples, we must not one important point. C does make a distinction between uppercase and lowercase letters. For example, **printf** and **PRINTF** are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output string like “I SEE” and “I REMEMBER”.

The above example that printed **I see, I remember** is one of the simplest programs. Fig 1.3 highlights the general format of such simple programs. All programs need a main function.

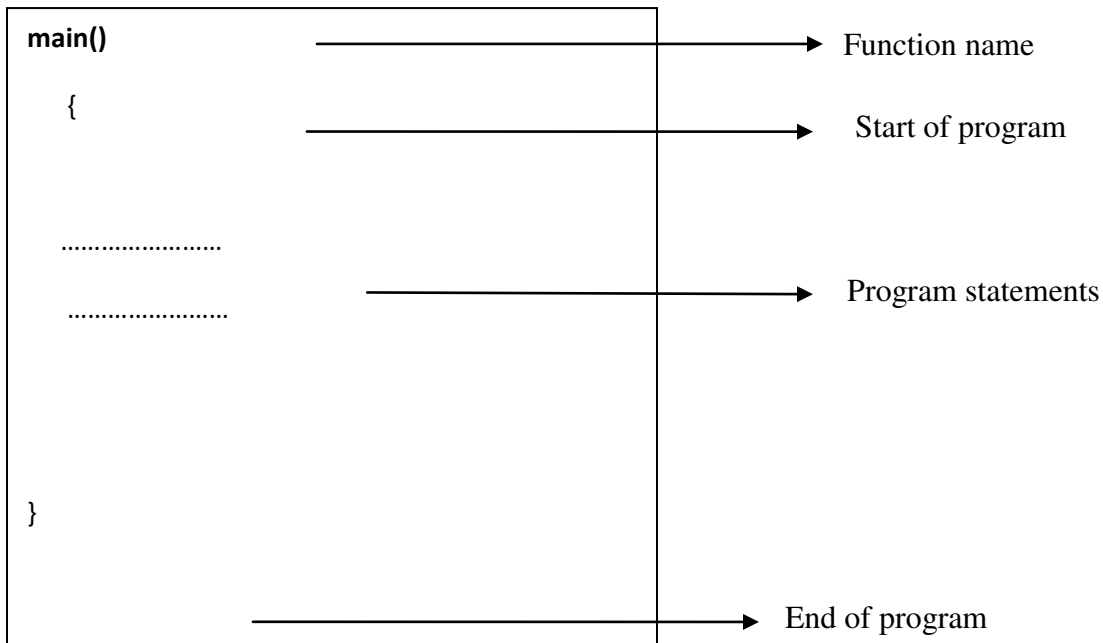


Fig.1.3 Format of simple C programs

THE MAIN FUNCTION

The main is a part of every C program. C permits different forms of main statement. Following forms are allowed.

- `main()`
- `intmain()`

- voidmain()
- main(void)
- void main(void)
- int main(void)

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyword void inside parentheses. We may also specify the keyword int or void before the word main. The keyword void means that the function does not return any information to the operating system and int means that the function returns an integer value to the operating system. When int is specified, the last statements in the program must be “return 0”. For the sake of simplicity, we use the first form in our programs.

1.4 SAMPLE PROGRAM

2: *ADDING TWO NUMBERS*

Consider another program, which performs addition two numbers and displays the result. The complete program is shown in Fig.1.4

```

/* program ADDITION                                Line-1*/
/* written by EBG                                   Line-2*/
main()                                              /* Line-3*/
{
/* Line-4*/
    int number;                                     /* Line-5*/
    float number                                    /* Line-6*/
/* Line-7*/
    number = 100;                                   /* Line-8*/
/* Line-9*/
    amount = 30.75 + 75.35;                         /* Line-10*/

```



```

printf(“%d\n”,number);           /* Line-11*/

printf(“%5.2f”,amount);         /* Line-12*/

}

```

Fig 1.4 Program when executed will produce the following output:

100

106.10

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are variable names that are used to store numeric data. The numeric data may be either in integer form or in real form. In C, all variables should be declared to tell the compiler what the **variables names** are and what **type of data** they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

int number;

float amount;

tell the compiler that **number** is an **integer(int)** and **amount** is a floating(**float**)point number. Declaration statements must appear at the beginning of the functions as shown in Fig.1.4. all declaration statements end with a semicolon; C supports many other data types and they are discussed in detail in Chapter 2.

The words such as **int** and **float** are called the keywords and cannot be used as variable names. A list of keywords is given in chapter 2.

Data is stored in a variable by assigning a data value to it. This is done in lines 8 and 10. In line-8, an integer value 00 is assigned to the integer variable **number** and in line-10; the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount**. The statements

```
number = 100;
```

```
amount = 30.75 + 75.35;
```

are called the assignment statements. Every assignment statement must have a semicolon at the end. The next statement is an output statement that prints the value of **number**. [The print statement

```
printf(“%d\n”, number);
```

contain two arguments. The first argument “%d” tells the compiler that the value of the second argument **number** should be printed as a decimal integer. Note that these arguments are separated by a comma. The last statement of the program

```
printf(“%5.2f”, amount);
```

prints out the value of **amount** in floating point format. The format specification %5.2f tells the compiler that the output must be in floating point, with five places in all and two places to the right of decimal point.

1.5 SAMPLE PROGRAM

3: INTEREST CALCULATION

The program in fig.1.5 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in fig 1.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = value at start of year (1+ interest rate)

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement

```
amount = value;
```

Makes the value at the end of the current year as the value at start of the next year.

```

/*.....INVESTMENT PROBLEM.....*/

#define PERIOD 10

#define PRINCIPAL 5000.00

/*.....MAIN PROGRAM BEGINE.....*/

main()

{ /*.....DECLARATION STATEMENTS .....*/

int year;

float amount, value, inrate;

/*.....ASSIGNMENT STATEMENTS.....*/

amount = principal;

inrate = 0.11;

year = 0;

/*.....COMPUTATION STATEMENTS.....*/

/*.....COMPUTATION USING WHILE LOOP.....*/

while(year <= period)

{

printf(“%2d %8.2f\n”,year, amount);

value = amount + inrate * amount;

year = year + 1;

amount = value;

}

```

```

/*.....WHILE LOOP ENDS.....*/

}

/*.....PROGRAM ENDS.....*/

```

Fig 1.5 Program for investment problem

Let us consider the new features introduced in this program. The second and third lines begin with #define instruction. A #define value to a symbolic constant for use in the program whenever a symbolic name is encountered, the compiler substitutes the value associated with the name automatically. To change the value, we have to simply change the definition. In this example, we have defined two symbolic constants PERIOD and PRINCIPAL and assigned values 10 and 5000.00 respectively. These values remain constant throughout the execution of the program.

0	5000.00
1	5550.00
2	6160.50
3	7590.35
4	8425.29
5	9352.07
6	10380.00
7	11522.69
8	12790.00
9	14197.11

Fig 1.6 Output of the investment program

The #define directive

A **#define** is pre-processor compiler directive and not a statement. Therefore **#define** lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names. **#define** instruction are usually

placed at the beginning before the **main()** function. Symbolic constants are not declared in declaration section. Pre-processor are discussed in chapter 14.

We must note that the defined constants are not variables. We may not change their values within the program by using an assignment statement. For example, the statement

PRINCIPAL = 10000.00; is illegal.

The declaration section declares **year** as integer and **amount**, **value** and **inrate** as floating point numbers. Note all the floating-point variables are declared in one statement. They can also be declared as

float amount;

float value;

float inrate;

When two or more variables are declared in one statement, they are separated by a comma.

All computations and printing are accomplished in a **while** loop. **While** is mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of year is less than or equal to the value of PERIOD, four statements that follow **while** are executed. Note that these four statements are grouped by braces. We exit the loop when **year** becomes greater than PERIOD the concept and types of loops are discussed in chapter6.

C supports the basic four arithmetic operators| (-, +,*, /) along with several others. They are discussed in chapter 3.

1.6 SAMPLE PROGRAM 4: USE OF SUBROUTINES

So far, we have used only **printf** function that has been provided for us by the C system. The program shown in fig 1.7 uses a user-defined function. A function defined by the user is equivalent to subroutine in FORTRAN or subprogram in BASIC.

Fig 1.7 presents a very simple program that uses a **mul()** function. The program will print the following output.

MULTIPLICATION OF 5 AND 10 IS 50

```
/*.....PROGRAM USING FUNCTION.....*/

int mul (int a, int b);

/*.....DECLARATION.....*/

main()

{

    int a, b, c;

    a = 5;

    b = 10;

    c = mul (a,b);

    printf ("multiplication of %d and %d is %d", a,b,c);

}

/*.....MAIN PROGRAM ENDS MUL() FUNCTION STARTS.....*/

int mul (int x, int y)

int p;

{

    p = x*y;

    return(p);

}

/*.....MUL ( ) FUNCTION ENDS .....*/
```

Fig 1.7 A program using a user defined function.

The `mul ()` function multiplies the values of `x` and `y` and the result is returned to the `main()` function when it is called in the statement

```
c = mul (a, b);
```

The `mul ()` has two arguments `x` and `y` that are declared as integers. The values of `a` and `b` are passed on to `x` and `y` respectively when the function `mul ()` is called. User-defined function are considered in detail chapter 9.

1.7 SAMPLE PROGRAM: USE OF MATH FUNCTIONS

We often use standard mathematical functions such as `cos`, `sin`, `exp`, etc. We shall see now the use of a mathematical function in a program. The standard mathematical functions are defined kept as a part of C **math library**. If we want to use any of these mathematical functions, we must add an `#include` instruction in the program,. Like `#define`, it is also a compiler directive that instructs the compiler to link the specified mathematical functions from the library. The instruction is of the form

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

`math.h` is the filename containing the required function. Figure 1.8 illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20,....., 180 and prints out the results with headings.

```
/*.....PROGRAM USING COSIN FUNCTION.....*/
```

```
#define <math.h>
```

```
#define PI 3.1416
```

```
#define MAX 180
```

```
main ( )
```

```
{
```

```
int angle ;
```

```

float x,y;

angle = 0;

printf(" angle    cos(angle)\n\n");

while(angle <= MAX)

{

x = (PI/(MAX)*angle;

y = cos(x);

printf("%15d %13.4f\n", angle, y);

angle = angle+ 10;

}

}

```

OUTPUT

Angle	cos(angle)
0	1.0000
10	0.9848
20	0.9397
30	0.8660
40	0.7660
50	0.6428
60	0.5000
70	0.3420

80	0.1736
90	-0.0000
100	-0.1737
110	-0.3420
120	-0.5000
130	-0.6428
140	-0.7660
150	-0.8660
160	-0.9367
170	-0.9848
180	-1.0000

Fig 1.8 Program using a math function

Another **#include** instruction that is offer required is

#include <stdio.h>

stdio.h refers to the standard I/O header file containing standard input and output functions.

The #include directive

As mentioned earlier, C programs are divided into modules or functions. Some functions are written by users, like us, many others are stored in the C library. Library functions are grouped category wise and stored in different files known as header files. If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

This is achieved by using the pre-processor directive **#include** as follows:

#include<filename>

filename is the name of the library file that contains the required function definition. Pre-processor directives are placed at the beginning of a program.

A list of library function and header files containing them are given in Appendix III.

1.8 BASIC STRUCTURE OF C PROGRAMS

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in fig 1.9.

Documentation section

Linksection

Definition section

Globla Declaration Section

main () Function section

{

Declaration part

Executable part

}

Subprogram section

Function 1

Function 2

..... (User-defined functions)

.....

Function n

Fig 1.9 An overview of a C program

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one main () function section. This contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon (;).

The subprogram section contains all the user-defined functions that are called in the main function. User-defined functions are generally placed immediately after the main function, although they may appear in any order.

All sections, except the main function section may be absent when they are not required.

1.9 PROGRAMMING STYLE

Unlike some other programming languages (COBOL, FORTRAN, etc..) C is a free-form language. That is, the C compiler does not care where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of fig 1.5.

Since C is free-form language, we can group statements together on one line. The statements

```
a = b;
```

```
x = y + a;
```

```
z = a + x;
```

Can be written on one line as

```
a=b;x=y+1;z=a+x;
```

The program `main()`

```
{  
    Printf("hello C");  
}
```

May be written in one line like

```
main ( ) {printf("Hello C");}
```

However, this style makes the program more difficult to understand and should not be used. In this book, each statement is written on a separate line. The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

1.10 EXECUTING A 'C' PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program;
2. Compiling the program:
3. Linking the program with function that are needed form the C library; and
4. Executing the program

Figure 1.10 illustrates the process of creating, compiling and executing a C program. Although there steps remain the same irrespective of the operating system, system commands for implementing steps and conventions for naming files may differ on different system.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channelled through the operating system. The operating system, which is a interface between the hardware and the user, handles the execution of user programs.

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers). We shall discuss briefly the procedure to follow in executing C programs under both these operating systems in the following sections.

1.11 UNIX SYSTEM

Creating the Program

Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered *int* a file. The file name can consist of letter, digits and special characters, followed by a dot and a letter C. Examples of valid file names are:

hello.c

program.c

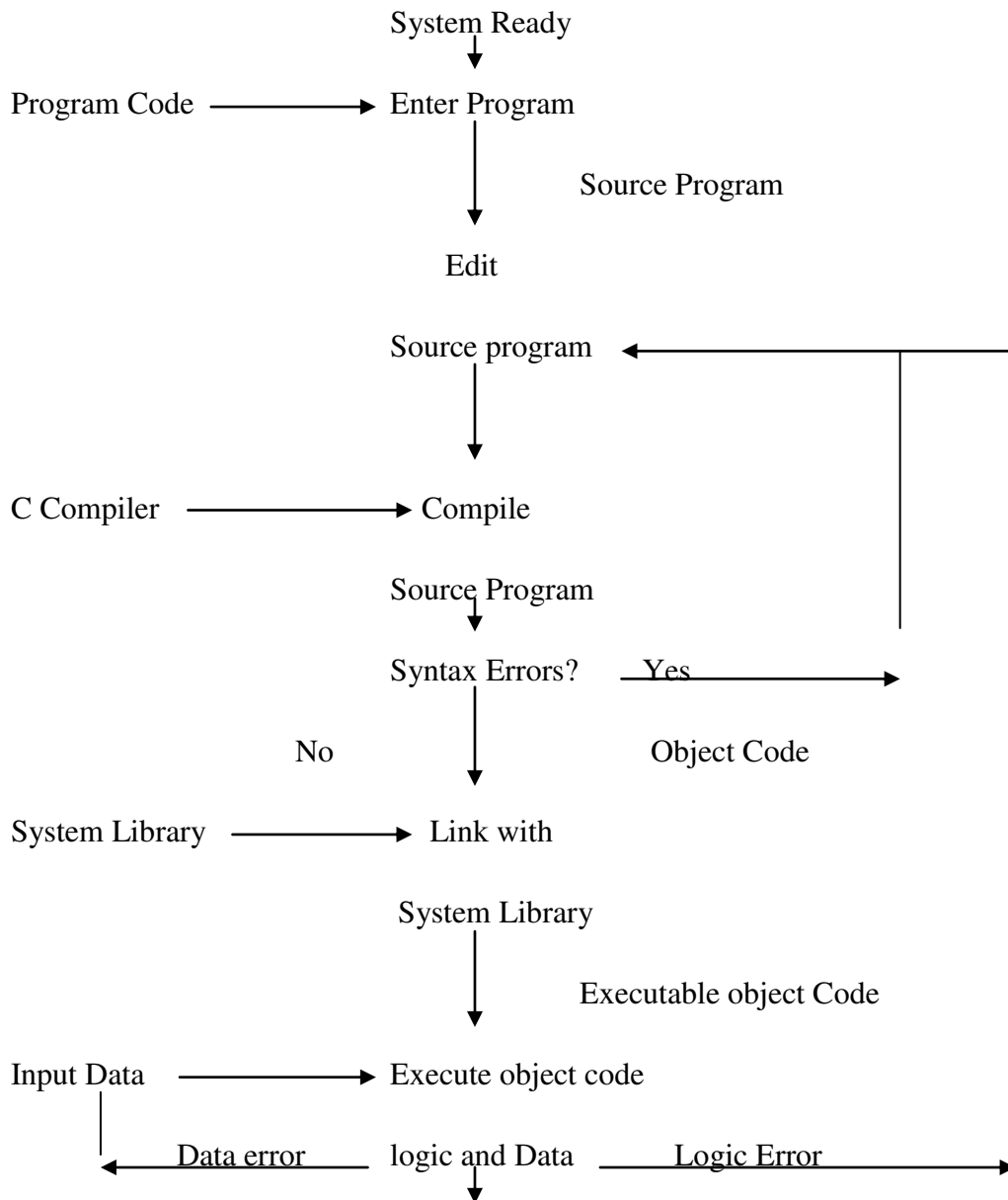
ebgl.c

The file created with the help of text editor, either **ed** or **vi**. The command for calling the editor and creating the file is

ed filename

If the file existed before, it is loaded. If it does not yet exist, the file has to create so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system; manual)

When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the source program, since it represents the original form of the program.



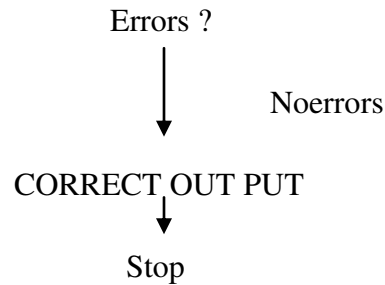


Fig 1.10 Process of compiling and running a C program

COMPILING AND LINKING

Let us assume that the source program has been created in a file named `ebgl.c`. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is `cc ebgl.c`

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name `ebgl.o`. This program is known as *object code*.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using `exp()` function, then the object code of this function should be brought from the **math library** of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the `cc` command is used.

If any mistakes in the syntax and semantics of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the executable object code and is stored automatically in another file named **a.out**.

Note that some systems use different compilation command for linking mathematical functions.

`cc filename -l`

is the command under UNIPLUS SYSTEM V operating system.

Executing the program

Executing is simple task. The command

a.out

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program logic or data. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Your Own Executable File:

Note that the linker always assigns the same name a.out. when we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

`mv. a.out name`

We may also achieve this by specifying an option in the cc command as follows:

This will store the executable object code in the file name and prevent the old file a.out from being destroyed.

Multiple Source Files

To compile and link multiple source program files, we must append all the files names to the cc command.

`cc filename-1.c..filename-n.c`

These files will be separately compiled into object files called

filename-i.o

and then linked to produce an executable program file a.out as shown in fig 1.11.

It is also possible to compile each file separately and link them later. For example, the commands

```
cc -c mod1.c
```

```
cc -c mod2.c
```

Will compile the source files mod1.c and mod2.c into objects files mod1.o and mod2.o. they can be linked together by the command `cc mod1.o mod2.o`

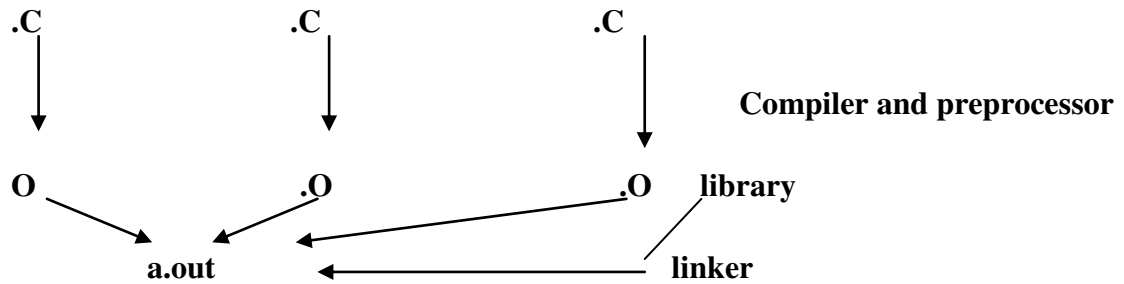


Fig 1.11 compilation of multiple files

We may also combine the source files and object files as follows:

```
cc mod1.c mod2.o
```

Only mod1.c is compiled and then linked with the object file mod2.o. this approach is useful when one of the multiple source files need to be changed and recompile or an already existing object files is to be used along with the program to compiled.

1.12 MS-DOS SYSTEM

The program can be cre3ated using any word processing software IN NON-DOCUMENT MODE. THE FILE NAMED SHOULD END WITH THE CHARACTERS. “c” like program.c, pay.c,etc. Then the command

```
MSC pay.c
```

Under MS-DOS operating system would load the program stored in the file pay.c and generate the object code. This code is stored in another file under name pay.obj. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the command

```
LINK pay.obj
```

Which generates the executable code with the filename pay.exe? Now the command

```
Pay
```

Would execute the program and give the results.

2 CONSTANTS, VARIABLES, AND DATA TYPES

KEY TERMS

Identifiers, constant, String constant, Variable, Scanf

2.1 INTRODUCTION

A programming language is designed to help process certain kinds of data consisting of numbers, characters and strings and to provide useful output known as information. The task of processing of data is accomplished by executing a sequence of precise instructions called a program. These instructions are formed using certain symbols and words according to some rigid rules known as syntax rules (or grammar). Every program instruction must conform precisely to the syntax rules of the language.

Like any other language, C has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to C programming language.

2.2 CHARACTER SET

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be

used on most personal, micro, mini and mainframe computers. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The entire character set is given in table 2.1.

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

Trigraph Characters

Many non-English keyboards do not support all the characters mentioned in Table 2.1. ANSI C introduces the concept of “trigraph” sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 2.2. For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraphs `??(and??)`.

Table 2.1 C Character Set

Letters	Digits
Uppercase A.....Z	All decimal digits 0.....9
Lowercase a.....z	
Special characters	
, comma	& ampersand
. Period	^ caret
; Semicolon	* asterisk
: Colon	– minus sign
?question mark	+ plus sign
'Apostrophe	< opening angle bracket
"Quotation mark	(or less than sign)
! Exclamation mark	> closing angle bracket
Vertical bar	(or greater than sign)
/ slash	(left parenthesis
\ Back slash) right parenthesis
~tilde	[left bracket
_under score] right bracket
\$ dollar sign	{left brace
% percent sign	} right brace
# Number sign	

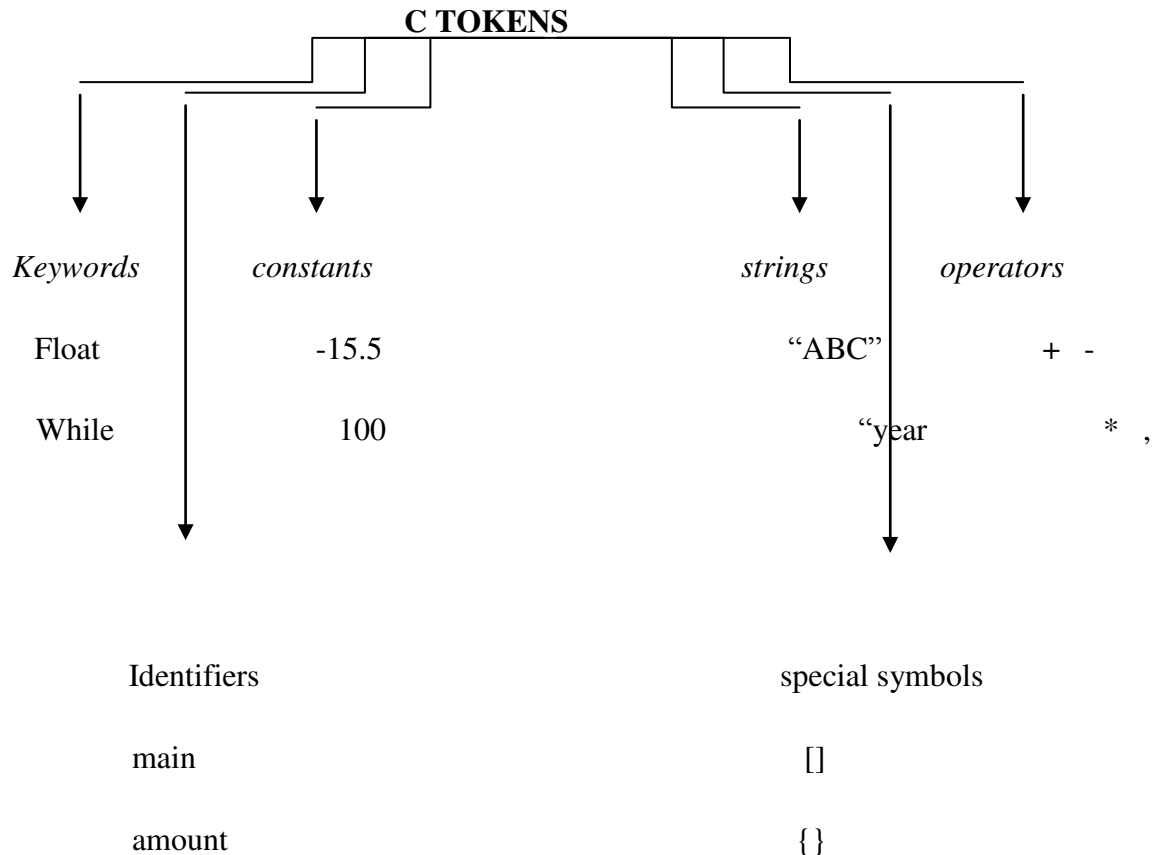
White spaces
Blank space
Horizontal tab
Carriage return
New line
Form feed

Table 2.2 ANSI C trigraph sequences

Trigraph sequence	Translation
??=	# number sign
??{	[left bracket
??)]	right bracket
??<	{left brace
??>	} right brace
??!	Vertical bar
??/	\ back slash
??/	^ caret
??-	~tilde

2.3 C TOKEN

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig.2.1. C programs are written using these tokens and the syntax of the language.



2.4 KEYWORDS AND IDENTIFIERS

Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of ANSI C is listed in Table 2.3. All keywords must be written lowercase. Some compilers may use additional keywords that must be identified from the C manual.

NOTE: C99 adds some more keywords; see the Appendix "C99 Features".

Table 2.3 ANSI C KEYWORD

Auto	double	int	struct
Break	else	long	switch
Char	extern	return	union
Const	float	short	unsigned
Continue	for	signed	void
Default	goto	sizeof	volatile
Do	if	static	while

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

Rules for identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letter, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

2.5 CONSTANTS

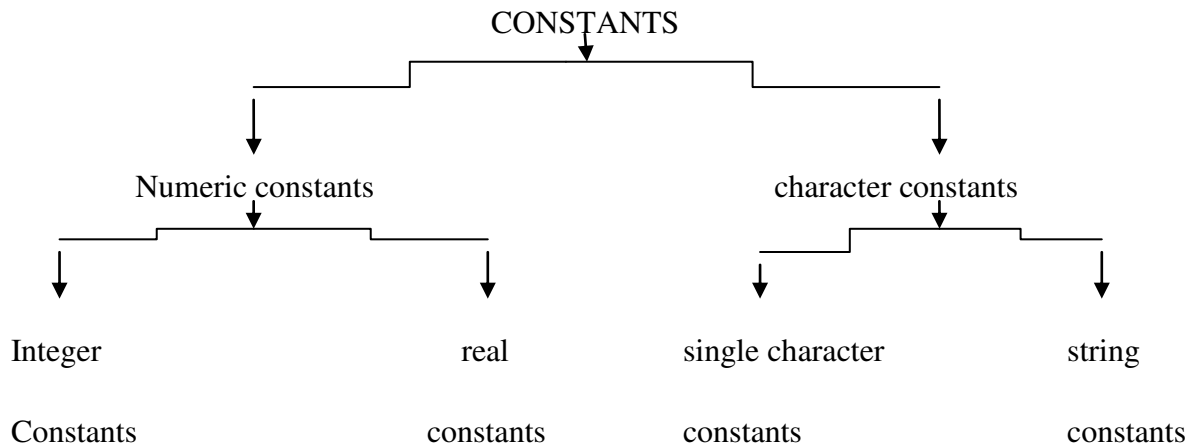
Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in fig 2.2.

INTEGER CONSTANTS

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, decimal, integer, octal integer and hexadecimal integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123 -321 0 654321 +78



Embedded spaces, commas, and non-digit characters are not permitted between digits. For example 15 750 20,000 \$1000 are illegal.

NOTE: ANSI C supports unary plus which was not defined earlier.

An octal integer constant consists of any combination of digits from the set 3 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 05511

A sequence of digits preceded by ox or oX is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letter A through F represents the number 10 through 15. Following are examples of valid hex integers:

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U,L and UL to the constants. Examples:

56789U	or 56789u	(unsigned integer)
98761234UL	or 98761234ul	(unsigned integer)
9876543L	or 9876543I	(long integer)

The concepts of unsigned and long integers are discussed in detail in section 2.7.

PROGRAM 2.1: Representation of integer constants on a 16- bit computer.

The program in fig 2.3 illustrates the use of integer constants on a 16-bit machine. The output in Fig 2.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

Program

```
main()  
{  
    printf("integer values\n\n");  
    printf("%d %d\n", 32767, 32767+,32767+10);  
    printf("\n");  
    printf("Long integer values\n\n");  
    printf("%ld %ld\n", 32767L,3276L+1L,32767L+10L);
```

}

Output:

Integer values

32767 -32768 -32759

Long integer values

32767 32768 3777

Fig 2.3: Representation of integer constants on 16-bit machine

Real constants

Integer numbers are inadequate to represent quantities that vary continuously. Such as distance, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called real (or floating point) constants. Further examples of real constants are:

0.0083 -0.75 435.36 +247.0

These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. That is,

215. .95 -.71 +.5

are all valid real numbers.

A real number may also be expressed in *exponential (or scientific) notation*. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2 . The general form is:

mantissa e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. The letter e separating the

mantissa and the exponent can be written in either lowercase or uppercase. “Since the exponent causes the decimal point to “float”, this notation is said to represent a real number in floating point form. Examples of legal floating-point constants are:

0.65d4 12e-2 1.5e+5 3.18E3 -1.2E-1

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single – precision and I or L to extend double precision further. Some examples of valid and invalid numeric constants are given in table 2.4.

Table 2.4: Examples of Numeric constants

CONSTANT	VALID?	REMARKS
698354L	Yes	represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
\$255	No	\$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. Example of character constants are:

```
'5' 'X' ',' ' '
```

Note that the character constant '5' is not the same as the number 5. The last constant is a blank space.

Character constants have integer values known as ASCII Values. For example, the statement

```
printf("%d", 'a');
```

Would print the number 97, the ASCII Value of the letter a. Similarly, the statement

```
printf("%c", '97');
```

would out the letter 'a'. ASCII values for all characters are given in appendix II.

Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants. They are discussed in chapter 8.

String constants

A string constant is a sequence of characters enclosed in double quotes. The characters may be letter, numbers, special characters and blank space. **Example is:**

```
"Hello!" "1987" "WELLDONE" "?...!" "5+3" "X"
```

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs.

BACKSLASH CHARACTER CONSTANTS

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of backslash character

constants is given in table 2.5. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as escape sequences.

Constant	meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\''	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

Table.2.5

2.6 VARIABLES

A *variable* is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. In chapter 1, we used several variables. For instance, we used

the variable amount in sample program 3 to store the value of money at the end of each year (after adding the interest earned during that year).

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

average

height

total

counter_1

class_strength

As mentioned earlier, variable names may consist of letters, digits, and the underscore(_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers. (In C99, at least 63 characters are significant.)
3. Uppercase and lowercase are significant. That is, the variable Total is not the same as total or TOTAL.
4. It should not be a keyword.
5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
Mark	sum1	distance

Invalid examples include:

123 (area)
 % 25th

Further examples of variable names and their correctness are given in Table 2.6.

Table 2.6 Examples of Variable Names

Variable name	valid ?	Remark
First_tag	valid	
Char	Not valid	char is a keyword
Price\$	Not valid	Dollar sign is illegal
Group one	Not valid	Blank space is not Permitted
average_number	valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

If only the first eight characters are recognized by a compiler, the the two names

Average_height

Average_weight

Mean the same thing to the computer. Such names can be rewritten as

avg_height and avg_weight

Or

.ht_average and wt_average

Without changing their meanings.

2.7 DATA TYPES

C Language is rich in its data types. Storage representations and machine instructions to handle constants differ from machine to machine. The varieties of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSI C supports three classes of data types:

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered.

All C compilers support five fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), double-precision floating point (**double**) and void. Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given. Fig2.4. the range of the basic four types is given in table2.7. we discuss briefly each one of them in this section.

NOTE: C99 adds three more data types, namely `_Bool`, `_Complex`, and `_Imaginary`. See the Appendix "C99 Features"

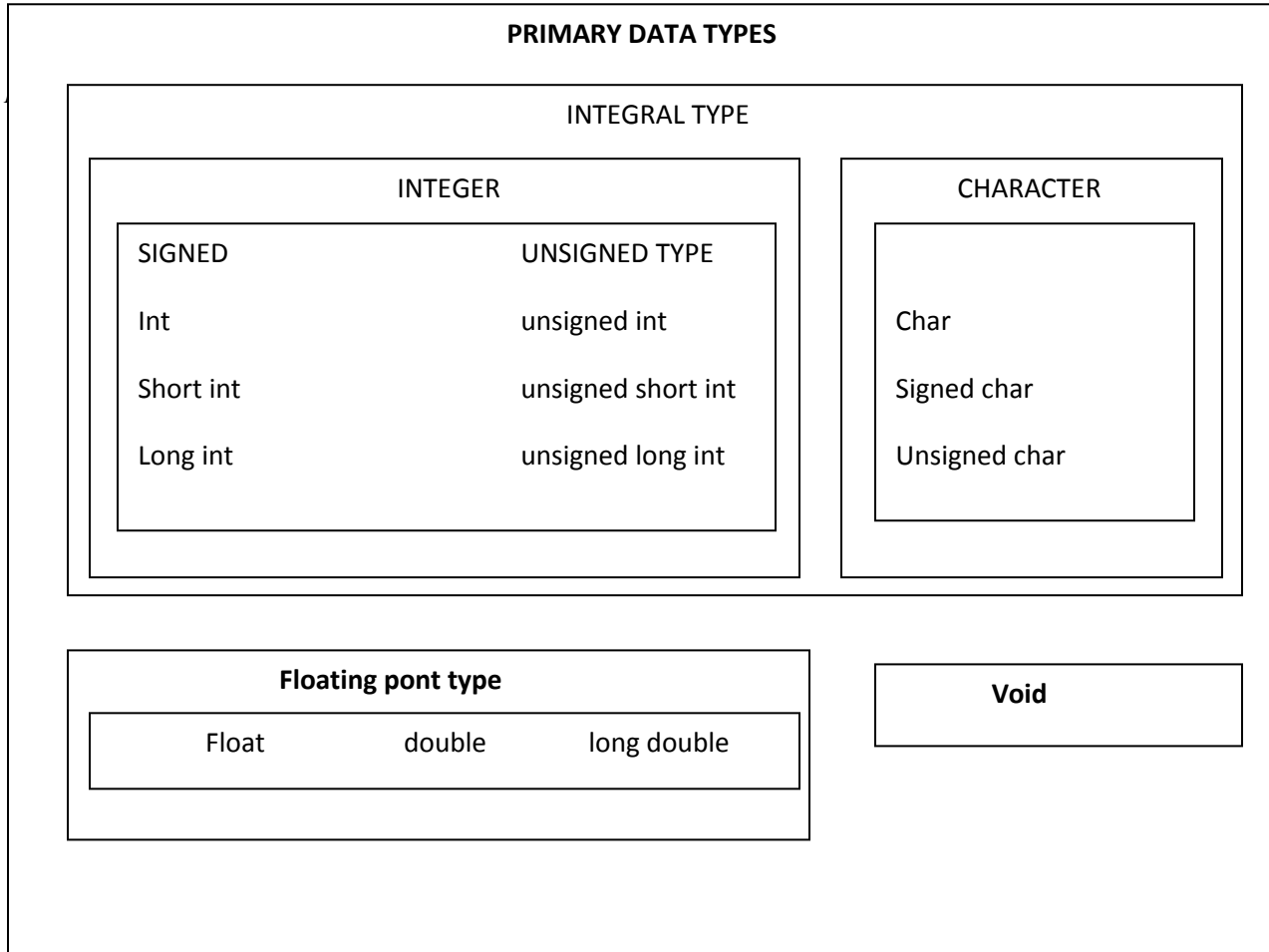


Fig 2.4 primary dat types in C

Table 2.7 Size and Range of Basic Data types on 16-bit machine

Data type	Range of Values
Char	-128 to 127
Int	-32768 to 32767
Float	3.4e-38 to 3.4e+e38
Double	1.7e-308 to 1.7e+308

INTEGER TYPES

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range -32768 to +32767 (that is, -2^{15} to $+2^{15}-1$). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely short int, int, and long int, in both signed and unsigned form. ANSI C defines these types so that they can be organized from the smallest to the largest, as shown in Fig.2.5. For example, short int represents fairly small integer values and requires half the amount of storage as regular int number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16-bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

We declare long and unsigned integers to increase the range of values. The use of qualifier signed on integers is optional because the default declaration assumes a signed number. Table 2.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

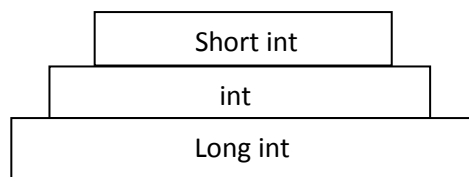


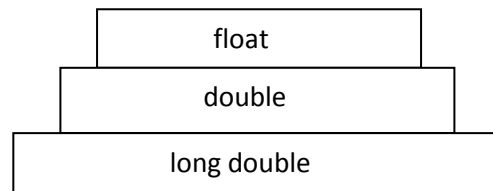
Table 2.8 Size and Range of Data Types on a16-bit machine

Type	size (bits)	range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32,768 to 32,767
unsigned int	16	0 to 65535
short int or		
signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or		
signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E -38/ to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 1.1E+4932

Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits precision. Floating point numbers are defined in C by the keyword float. When the accuracy provided by a float number is not sufficient, the type double data type number uses

64 bits giving a precision of 14 digits. These are known as double precision numbers; Remember that double type represents the same data type that float represents, but with a greater precision. To extend the precision further, we may use long double which uses 80 bits. The relationship among floating types is illustrated 2.6



2. 8 Declaration of variables

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows:

Data - type v1,v2,....,vn

v1,v2,vn are the names of variables. Variables are separated by commas. A declaration must end with a semicolon. For example, valid declarations are:

int count;

int number, total;

double ratio;

int and **double** are the keywords to represent integer type and real type data values respectively.

Table 2.9 shows various data types and their keyword equivalents.

Table 2.9 Data Types and Their Keywords

Type	size(bits)	range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E -38/ to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 1.1E+4932

The program segment given in Fig.2.7 illustrates declaration variables. `main()` is the beginning of the program. The opening brace `{` signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the main function. The importance of place of declaration will be dealt in detail later while discussing functions.

Note:

C99 permits declaration of variables at any point within a function or block, prior to their use.

```

main() /*.....program name.....*/
{
    /*.....declaration.....*/

    float        x, y;

    int          code;

    short int    count;

    long int     amount;

    double       deviation;

    unsigned     n;

    char         c;

    /*.....computation.....*/

    .....

    .....

    .....

}

/*.....program ends.....*/

```

Fig 2.7 Declaration of variables

When an adjective (qualifier) **short, long, or unsigned** is used without a basic data type specifier, C compilers treat the data type as an **int**. If we want to declare a character variable as unsigned, then we must do so using both the terms like **unsigned char**.

Default values of Constants

Integer constants, by default, represent int type data. We can override this default by specifying unsigned or long after the number (by appending U or L) as shown below:

Literal	Type	Value
+111	int	111
-222	int	-222
45678U	unsigned int	45,678
-56789L	unsigned int	-56,789
987654UL	unsigned long int	9,87,6564

Similarly, floating point constants, by default represent double type data. If we want the resulting data type to be float or long double, we must append the letter f or F to the number for float and letter I or L for long double as shown below:

Literal	Type	Value
0.	Double	0.0
.0	double	0.0
12.0	double	12.
1.234	double	1.234
-1.2f	float	-1.2
1.23456789L	long double	1.23456789

USER-DEFINED TYPE DECLARATION

C supports a feature known as “known definition” that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form:

typedef type identifier;

Where *type* refers to an existing data type and “identifier” refers to the “new” name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is ‘new’ only in the name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

typedef int units;

typedef float marks;

Here, units symbolize **int** and **marks** symbolizes **float**. They can be later used to declare variables as follows:

units batch1, batch2;

marks name1[50], name2[50];

Batch1 and batch2 are included as int variable and name1[50] and name2[50] are declared as 50 element floating point array variables. The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

enum identifier {value1, value 2, ..valuen};

The “identifier” is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as enumeration constants). After this definition, we can declare variables to be of this ‘new’ type as below:

enum identifier v1, v2, ...,vn;

The enumerated variables v1, v2,...,vn can only have one of the values value1, value2,...valuen.

The assignments of the following types are valid:

v1 = value3;


```
v5 = value1;
```

An example:

```
enum day {Monday,Tuesday,...Sunday};
```

```
enum day week_st, week_end;
```

```
week_st = Monday;
```

```
week_end = Friday;
```

```
if(week_st == Tuesday)
```

```
week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value 1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example

```
enum day {Monday,Tuesday,...Sunday};
```

Here the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. Example:

```
enum day {Monday,Tuesday,...Sunday} week_st, week_end;
```

2.9 DECLARATION OF STORAGE CLASS

Variables in C can have not only data type but also storage class that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized. Consider the following example:

```
/* Example of storage classes */
```

```
int m;
```

```

main()
{
int i;

float balance;

.....

.....

function1();

}

function( )
{
int i;

float sum;

.....

.....

}

```

The variable **m** which has been declared before the **main** is called *global* variable. It can be used in all the functions in the program. It need not be declared in other functions. A global variable is also known as an external variable.

The variables **i**, **balance** and **sum** are called local variables because they are declared inside a function. Local variables are visible and meaningful only inside the functions in which they are declared. They are not known to other function. Note that the variable **i** has been declared in both the function. Any change in the value of **i** in one function does not affect its value in the other. C provides a variety of storage class specifies that can be used to declare explicitly the

scope and lifetime of variables. The concepts of scope and lifetime are important only in multifunction and multiple file programs and therefore the storage classes are considered in detail later when functions are discussed. For now, remember that there are four storage class specifiers (**auto**, **register**, **static**, and **extern**) whose meanings are given in table 2.10

The storage class is another qualifier (like **long** or **unsigned**) that can be added to a variable declaration as shown below:

auto int count;

register char ch;

static int x;

extern long total;

Static and external (**extern**) variables are automatically initialized to zero. Automatic (**auto**) variables contain undefined values (known as ‘garbage’) unless they are initialized explicitly.

Table 2.10 Storage Classes and their meaning

Storage class	meaning
auto	local variable known only to the function in which it is declared. Default is auto.
static	local variable which exists and retains its value even after the control is transferred to the calling function.
extern	global variable known to all functions in the file.
register	local variable which is stored in the register.

Variables are created for use in program statements such as,

value = amount + inrate * amount;

While (year <= PERIOD)

```
{  
.....  
.....  
year = year + 1;  
}
```

In the first statement, the numeric value stored in the variable `inrate` is multiplied by the value stored in `amount` and the product is added to `amount`. The result is stored in the variable `value`. This process is possible only if the variables `amount` and `inrate` have already been given values. The variable `value` is called the target variable. While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal (=) sign of a computational statement) must be assigned values before they are encountered in the program. Similarly, the variable `year` and the symbolic constant `PERIOD` in the while statement must be assigned values before this statement is encountered.

Assignment Statement

Values can be assigned to variables using the assignment operator `=` as follows:

```
variable_name = constant;
```

We have already used such statements in chapter 1. Further example are:

```
initial_value = 0;
```

```
final_value = 100;
```

```
balance = 75.84;
```

```
yes = 'x';
```

C permits multiple assignments in one line. For example

```
initial_value = 0; final_value = 100;
```

are valid statements.

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The statement

```
year = year + 1;
```

means that the 'new value' of year is equal to the 'old value' of year plus 1.

During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

It is also possible to assign a value to a variable at the time the variable is declared. This takes the following form:

```
data-type variable_name = constant;
```

Some examples are:

```
int final_value = 100;
```

```
char yes = 'x';
```

```
double balance = 75.84;
```

The process of giving initial values to variables is called initialization. C permits the initialization of more than one variable in one statement using multiple assignment operators. For example the statements

```
p = q = s = 0;
```

```
x = y = z = MAX;
```

are valid. The first statement initializes the variables p, q, and s to zero while the second initializes x, y, and z with MAX. Note that MAX is a symbolic constant defined at the beginning.

Remember that external and static variables are initialized to zero by default. Automatic variables that are not initialized explicitly will contain garbage.

Program 2.2: program in Fig.2.8 shows typical declarations, assignments and values stored in various types of variables.

The variables x and p have been declared as floating-point variables. Note that the way the value of 1.234567890000 that we assigned to x is displayed under different output formats. The value of x is displayed as 1.234567880630 under %12f format, while the actual value assigned is 1.234567890000. This is because the variable x has been declared as a float that can store values only up to six decimal place.

The variable m that has been declared as **int** is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an **int** variable can store is only 32767. However, the variable k (declared as unsigned) has stored the value 54321 correctly. Similarly, the **long int** variable n has stored the value 1234567890 correctly.

The value 9.87654321 assigned to y declared as double has been stored correctly but the value is printed as 9.876543 under %f format. Note that unless specified otherwise, the printf function will always display a float or double value to six decimal place. We will later the output formats for displaying numbers.

Program

```
main( )
{
/*.....Declarations.....*/

float x,p;

double y,q;

unsigned k;

/*.....declartions and assignemnts.....*/

int m =54321 ;
```

```

long int n = 1234567890 ;

/*.....ASSIGNMENTS.....*/

x = 1.234567890000 ;

y = 9.87654321 ;

k = 54321 ;

p = q= 1.0 ;

/*.....PRINTING.....*/

printf("m = %d\n", m) ;

printf("n = %1d\n", n) ;

printf("x = %.121f\n", x) ;

printf("x = %f\n", x) ;

printf("y = %.121f\n",y) ;

printf("y = %1f\n", y) ;

printf("k = %u p = %f q = %.121f\n", k, p, q) ;

}

```

Output

m = -11215

n = 1234567890

x = 1.234567880630

x = 1.234568

y = 9.876543210000

y = 9.876543

k = 54321 p = 1.000000q = 1.000000000000

Fig 2.8 examples of assignments

Reading Data from Keyboard

Another way of giving values to variables is to input data through keyboard using the `scanf` function. It is a general input function available in C and is very similar in concept to the `printf` function. It works much like an `INPUT` statement in BASIC. The general format of `scanf` is as follows:

```
scanf("control string", &variable1,&variable1,&variable2,...);
```

The control string contains the format of data being received. The ampersand symbol & before each variable name is an operator that specifies the variable name's address. We must always use this operator, otherwise unexpected results may occur. Let us look at an example:

```
scanf("%d", &number);
```

When this statement is encountered by the computer, the execution stops and waits for the value of the variable `number` to be typed in. Since the control string `"%d"` specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. Once the number is typed in and 'Return' Key is pressed, the computer then proceeds to the next statement. Thus, the use of **`scanf`** provides an interactive feature and makes the program 'user friendly'. The value is assigned to the variable `number`.

Program 2.3 The program in Fig 2.9 illustrates the use of **`scanf`** function.

The first executable statement in the program is a **`printf`**, requesting the user to enter an integer number. This is known as "prompt message" and appears on the screen like

Enter an integer number

As soon as the user types in an integer number, the computer proceeds to compare the value with 100. If the value typed in is less than 100, then a message

Your number is smaller than 100

is printed on the screen. Otherwise, the message

Your number contains more than two digits

is printed. Outputs of the program run for two different inputs are also shown in fig 2.9.

program

```
main( )  
  
{  
  
int number;  
  
printf("Enter an integer number\n");  
  
scanf ("%d", &number);  
  
if (number < 100 )  
  
printf("Your number is smaller than 100\n\n");  
  
else  
  
printf("your number contains more than two digits\n");  
  
}
```

Output

Enter an integer number

54

Your number is smaller than 100

Enter an integer number

108 your number contains more than two digits

Fig.2.9 use of scanf function for interactive computing

Some compilers permit the use of the prompt message as a part of the control string in scanf, like

```
scanf("Enter a number %d",&number);
```

We discuss more about scanf in chapter 4.

In Fig 2.9 we have used a decision statement if....else to decide whether the number is less than 100. Decision statements are discussed in depth in chapter 5.

2.11 DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant "pi". Another example is the total number of students whose mark-sheets are analyzed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. we face two problems in the subsequent use of such programs. These are

1. Problem in modification of the program and
2. Problem in understanding the program.

Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same

program. We may forget what a certain number meant, when we read the program some days later.

Assignment of such constants to a symbolic name frees us from these problems. For example, we may use the name STRENGTH to define the number of students and PASS_MARK to define pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names STRENGTH AND PASS_MARK in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is defined as follows:

```
#define symbolic_name value of constant
```

Valid examples of constant definitions are:

```
#define STRENGTH 100
```

```
#define PASS_MARK 50
```

```
#define MAX 200
```

```
#define PI 3.14159
```

Symbolic names are sometimes called constant identifiers. Since the symbolic names are constants (not variables), they do not appear in declarations. The following rules apply to define statement which defines a symbolic constant:

1. Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letter. This only a convention, not a rule.)
2. No blank space between the pound sing '#' and the word define is permitted.
3. '#' must be the first character in the line.
4. A blank space is required between #define and symbolic name and between the symbolic name and the constant.
5. #define statements must not end with a semicolon.
6. After definition, the symbolic name should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal;.

7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. #define statements may appear anywhere in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

#define statement is a preprocessor compiler directive and is much more powerful than what has been mentioned here. More advanced types of definitions will be discussed later. Table 2.11 illustrates some invalid statements of #define.

Table 2.11 Examples of Invalid #define statements

Statement	Remark
#define X=2.5 ‘=’	sign is not allowed
#define MAX 10	No white space between # and define
#define N 25;	No semicolon at the end
#define N 5, M 10	A statement can define only one name
#define ARRAY	define should be in lowercase letters
#define PRICES\$ 100 \$	symbol is not permitted in name

2.12 DECLARING A VARIBALE AS CONSTANT

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier const at the time of initialization. Example: **const int class_size = 40;**

const is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the int variable **class_size** must not be modified by the program. However, it can be used on the right_hand side of an assignment statement like any other variable.

2.13 DECLARING A VARIABLE AS VOLATILE

ANSI standard defines another qualifier volatile that could be used to tell explicitly the compiler that variable’s value may be changed at any time by some external sources (from outside the program). For example: **volatile int date;**

The value of date may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When we declare a variable as volatile, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

Remember that the value of a variable declared as volatile can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variables as both const and volatile as shown below:

```
volatile const int location = 100;
```

Note: C99 adds another qualifier called restrict. See the Appendix “C99 Features”.

2.14 OVERFLOW AND UNDERFLOW OF DATA

Problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine. Since floating-point values are rounded off to the number of significant digits allowed (or specified), an overflow normally results in the largest possible real value, whereas an underflow results in zero.

Integers are always exact within the limits of the range of the integral data types used. However, an overflow which is a serious problem may occur if the data type does not match the value of the constant. C does not provide any warning or indication of integer overflow. It simply gives incorrect results. (Overflow normally produces a negative number.) We should therefore exercise a greater care to define correct data types for handling the input/output values.

Case studies

1. A program to calculate the average of a set of N numbers is given in Fig.2.11

Program

```
#define N 10          /* SYMBOLIC CONSTNT*/
```

```
main()
```

```

{
    int count;          /* declaration of variables */

    float sum, average, number ;

    sum = 0;

    count = 0;

while (count < n )
{
    scanf("%f", &number);

    sum = sum + number;

    count = count + 1;

}

    average = sum/n;

    printf("n= %d sum = %f", n, sum);

    printf(" average = %f", average);\

}

```

Output

1

2.3

4.67

1.42

7

3.67

4.08

2.2

4.25

8.21

N = 10 sum = 38.799999 Average = 3.880

Fig.2.11 Average of N numbers

The variable number is declared as float and therefore it can take both integer and real numbers. Since the symbolic constant N is assigned the value of 10 using the #define statement, the program accepts ten values and calculates their sum using the while loop. The variable count counts the number of values and as soon as it becomes 11, the while loop is exited and then the average is calculated.

Notice that the actual value of sum is 38.8 but the value displayed is 38.799999. In fact, the actual value that is displayed is quite dependent on the computer system. Such an inaccuracy is due to the way the floating point numbers are internally represented inside the computer.

3 OPERATORS AND EXPRESSIONS

3.1 INTRODUCTION

C supports a rich set of built-in operators. We have already used several of them, such as =, +, -, *, & and <. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operators can be classified into a number of categories. They include.

1. Arithmetic operators

2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

An expression is sequence of operands and operators that reduces to a single value. For example.

$$10 + 15$$

is an expression whose value is 25. The value can be any type other than void.

3.2 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. They are listed in Table 3.1, the operators +, -, * and / all work the same way as they do in other languages. These can operate on any built-in type allowed in C. The unary minus operator, in effect. Multiplies its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.

Table 3.1 Arithmetic Operator

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The modulo division operation products the remainder of an integer division. Examples of use of arithmetic operators are:

a-b	a+b
a*b	a/b
a%b	-a*b

Here a and b are variables and are known as operands. The modulo division operator % cannot be used on floating point data. Note that C does not have an operator for exponentiation. Older versions of C does not support unary plus but ANSI C supports it.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers. The expression is called an integer value. The largest integer value depends on the machine, as pointed out earlier. In the above examples, if a and b are integers, then for a=14 and b=4 we have the following results:

$$a-b = 10$$

$$a+b = 18$$

$$a*b = 56$$

$$a/b = 3 \text{ (decimal part truncated)}$$

$$a\%b = 2 \text{ (remainder of division)}$$

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one them is negative, the direction of truncation is implementation dependent. That is,

$$6/7 = 0 \text{ and } -6/-7 = 0$$

But -6/7 may be zero or -1. (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

$$-14\%3 = -2$$

$$-14\%-3 = -2$$

$$14\%-3 = 2$$

Program 3.1

The Program in fig.3.1 shows the use of integer arithmetic to convert a given number of days into months and days.

Program

```
main()
{
    int months, days;

    printf("Enter days\n");

    scanf("%d",&days);

    months = days/30;

    days = days%30;

    printf("Months = %d Days = %d",months,days);

}
```

Output

Enter days

265

Months = 8 days = 25

Enter days

364

Months = 12 Days = 4

Enter days

45

Months = 1 Days = 15

Fig.3.1 Illustration of integer arithmetic

The variables months and days are declared as integers. Therefore, the statement

$$\text{months} = \text{days}/30$$

Truncates the decimal part and assigns the integer part to months. Similarly, the statement

$$\text{days} = \text{days}\%30$$

assigns the remainder part of the division to days. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If x,y, and z are floats, then we will have:

$$x = 6.0/7.0 = 0.857143$$

$$y = 1.0/3.0 = 0.33333$$

$$z = -2.0/3.0 = -0.66667$$

The operator % cannot be used with real operands.

Mixed – mode Arithmetic

When one of the operands is real and the other is integer, the expression is called mixed – mode arithmetic expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus

$$15/10.0 = 1.5$$

whereas

$$15/10 = 1$$

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

3.3 RELATIONAL OPERATORS

We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two items, and so on. These comparisons can be done with the help of relational operators. We have already used the symbol ‘<’ meaning ‘less than’. An expression such as

$$a < b \text{ or } 1 < 20$$

containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. It is one if the specified relation is true and zero if the relation is false. For example

$$10 < 20 \text{ is true}$$

but

$$20 < 10 \text{ is false}$$

C supports six relational operators in all. These operators and their meanings are shown in Table 3.2

Table 3.2

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and takes the following form:

ae-1 relational operator ae-2

ae-1 and ae-2 are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values:

4.5<= 10 TRUE

4.5<-10 FALSE

-35>=0 FALSE

10<7+5 TRUE

a+b = c+d TRUE only if the sum of values of a and b is equal to the sum of values of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational expressions are used in decision statements such as if and while to decide the course of action of a running program. We have already used the while statement in chapter 1. Decision statements are discussed in detail in Chapter 5 and 6.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

- > is complement of <=
- < is complement of >=
- == is complement of !=

We can simplify an expression involving the not and the less than operators using the complements as shown below:

Actual one	Simplified one
!(x<y)	x>=y
!(x>y)	x<=y
!(x!=y)	x==y
!(x<=y)	x>y
!(x>=y)	x<y
!(x==y)	x!=y

3.4 LOGICAL OPERATORS

In addition to the relational operators, C has the following three logical operators.

&& meaning logical AND

|| meaning logical OR

! Meaning logical NOT

The logical operators && and II are used when we want to test more than one condition and make decisions. An example is:

$$a > b \ \&\& \ x == 10$$

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression. Like the simple relational expressions, a logical expression also yields a value of one or zero, according to the truth table shown in Table 3.3 The logical expression given above is true only if $a > b$ is true and $x == 10$ is true. If either (or both) of them are false, the expression is false.

Table 3.3 Truth table

op-1	op-2	Value of the expression	
		op-1 && op-2	Op-1 II op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)
2. if (number < 0 II number > 100)

We shall see more of them when we discuss decision statements.

Note

Relative precedence of the relational and logical operators is as follows:

Highest	!
	>>= < <=
	== !=
	&&
Lowest	

It is important to remember this when we use these operators in compound expressions.

3.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='. In addition, C has a set of 'shorthand' assignment operators of the form

v op = exp;

Where v is a variable, exp is an expression and op is a C binary arithmetic operator. The operator op= is known as the shorthand assignment operator.

The assignment statement

v op = exp;

is equivalent to

v = v op (exp);

with v evaluated only once. Consider an example

x += y+1;

This is same as the statement

x = x + (y+1);

The shorthand operator += means 'add y+1 to x' or 'increment x by y+1'. For y=2, the above statement becomes

`x +=3;`

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 3.4.

Table 3.4 Shorthand Assignment Operators

Statement with simple Assignment operator	Statement with Shorthand operator
<code>a = a+1</code>	<code>a += 1</code>
<code>a = a-1</code>	<code>a -= 1</code>
<code>a = a*(n+1)</code>	<code>a*=n+1</code>
<code>a = a/(n+1)</code>	<code>a/=n+1</code>
<code>a = a%b</code>	<code>a%=b</code>

The use of shorthand assignment operators has three advantages;

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

These advantages maybe appreciated if we consider slightly more involved statement like

`value(5*j-2) += delta;`

It is easier to read and understand and is more efficient because the expression `5*j-2` is evaluated only once.

Program 3.2

Program of Fig. 3.2 prints a sequence of squares of numbers. Note the use of the shorthand operator `*=`.

The Program attempts to print a sequence of squares of numbers starting from 2. The statement

```
a * = aj;
```

which is identical to

```
a = a*a;
```

replaces the current value of a by its square. When the value of a becomes equal or greater than N (=100) the while is terminated. Note that the output contains only three values 2,4 and 16.

Program

```
#define      N      100

#define      A      2

main()

{

int a;

a = a;

while (a < n)

{

printf(“%d\n”,a);

a *= a;

}

}
```

Output

2

4

16

Fig. 3.2 Use of shorthand operator *=

3.6 INCREMENT AND DECREMENT OPERATORS

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

++ and –

The operator ++ adds 1 to the operand, while – subtracts 1. Both are unary operators and take the following form:

++m; or m++;

--m; or m--;

++m is equivalent to $m = m + 1$; (or $m += 1$;))

--m is equivalent to $m = m - 1$; (or $m -= 1$;))

We use the increment and decrement statements in for and while loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement.

Consider the following:

m = 5;

y = ++m;

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
```

```
y = m++;
```

then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.

Similar is the case, when we use ++(or --) in subscripted variables. That is, the statement

```
a[i++] = 10;
```

is equivalent to

```
a[i] = 10;
```

```
i = i + 1;
```

The increment and decrement operator can be used in complex statements Examples;

```
m = n++ -j+10;
```

Old value of n is used in evaluating the expression n is incremented after the evaluation. Some compilers require a space on either side of n+= or ++n.

Rules for ++ and -- Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++(or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++(or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associativity of ++ and -- operators are the same as those of unary + and unary -.

3.7 CONDITIONAL OPERATOR

A ternary operator pair “?:;” is available in C to construct conditional expressions of the form

$$\text{exp1} \text{ ? exp2 : exp3}$$

where exp1,exp2, and exp3 are expressions.

The operator? : works as follows: exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either exp2 or exp3) is evaluated. For example, consider the following statements.

```
a = 10;
b = 15;
x = (a>b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the if . else statements as follows:

```
If (a > b)
    x = a;
else
    x = b;
```

3.8 BITWISE OPERATORS

C has a distinction of supporting special operators known as bitwise operator for manipulation of data at bit level. These operators are used of testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. Table 3.5 lists the bitwise operators and their meanings. They are discussed in detail in Appendix I.

Table 3.5 Bitwise Operators

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right

3.9 SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, sizeof operator, pointer operators (& and*) and member selection operators (. and ->). The comma and sizeof operators are discussed in this section while the pointer operators are discussed in Chapter 11. Member selection operators which are used to select members of a structure are discussed in Chapters 10 and 11. ANSI committee has introduced two pre-processor operators known as ‘string-izing’ and ‘token-pasting’ operators (# and ##). They will be discussed in Chapter 14.

The comma Operator

The comma operator can be used to link the related expressions together. Comma-linked lists of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression. For example, the statement

```
value = (x =10, y = 5, x +y);
```

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15(i.e. 10+5) to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

in for loop;

```
for ( n=3, m=10, n<=m; n++, m++)
```

in while loops

```
while (c = getchar(), c!= '10')
```

Exchanging values t = x, x =y, y = t;

The sizeof Operator

The sizeof is a compile time operator and, when used with an operand, it return the number of bytes the operand occupies. The operand may be a variable a constant or a data type qualifier.

Examples:

```
m = sizeof (sum);  
n = sizeof (long int);  
k = sizeof (235L);
```

The sizeof operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Program 3.3

In Fig.3.3 the program employs different kinds of operators. The results of their evaluation are also shown for expression.

Notice the way the increment operator ++ works when used in an expression. In the statement

```
c = ++a -b;
```

new value of a (=16) is used thus giving the value 6 to c. That is, a is incremented by 1 before if is used in the expression. However, in the statement

```
d = b++ + a;
```

the old value of b(=10) is used in the expression. Here, b is incremented by 1 after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement

```
printf('`a%%b = %d/n`, a%b);
```

The program also illustrate that the expression

```
C >d ? 1 : 0
```

Assumes the value 0 when c is less than d and 1 when c is greater than d.

Program

```
main ()
{
    int a, b, c, d;

    a = 15;

    b = 10;

    c = ++a -b;

    printf('`a = %d b= %d c = %d\n`, a,b,c);

    d = b++ +a;

    printf('`a =%d b=%d d=%d\n`, a, b, d);

    printf('`a/b = %d\n`, a/b);

    printf('`a%%b = %d\n`,a%b);

    printf('`a* = b = %d\n`, a*b);

    printf("`%d\n", (c>d) ? 1 : 0);
```



```
printf(“%d\n”, (c<d) ? 1 :0);  
}
```

Output

a = 16 b = 10 c = 6

a = 16 b = 11 d = 26

a/b = 1

a%b = 5

a *=b 176

0

1

Fig.3.3 Further illustration of arithmetic operators

3.10 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the languages. We have used a number of simple expressions in the examples discussed so far, C can handle any complex mathematical expressions, some of the examples of C expressions are shown in Table 3.6. Remember that C does not have an operator for exponentiation.

Table 3.6 Expressions

Algebraic expression	C expression
$a \times b - c$	$a * b - c$
$(m+n)(x+y)$	$(m+n)*(x+y)$
$\left(\frac{ab}{c}\right)$	$a*b/c$
$3x^2+2x+1$	$3*x*x+2*x+1$
$\left(\frac{x}{y}\right) + c$	$x/y+c$

Expressions are evaluated using an assignment statement of the form:

$$\text{variable} = \text{expression}$$

variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Examples of evaluation statements are

$$x = a * b - c;$$

$$y = b / c * a;$$

$$z = a - b / c + d;$$

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables, a,b,c and d must be defined before they are used in the expressions.

Program 3.4

The program in Fig. 3.4 illustrates the use of variables in expressions and their evaluation.

Output of the program also illustrates the effect of presence of parentheses in expressions. This is discussed in the next section.

Program

```
main ()
{
    float a,b,c,x,y,z;
    a = 9;
    b = 12;
    c = 3;
    x = a-b / 3 +c *2 -1;
    y = a-b / (3 +c) * (2-1);
    z = a-(b / (3+c) *2) -1;
    printf('`x=%f\n`,x);
    printf('`y =%f\n`,y);
    printf('`z = %f\n`,z);
}
```

Output

```
x = 10.000000
y = 7.0000000
z = 4.0000000
```

Fig. 3.4 illustrations of evaluation of expressions

3.12 PRECEDENCE OF ARITHMETIC OF OPERATORS

An Arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High priority */%

Low priority +-
^

The basic evaluation procedure includes two left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program of Fig.3.4

$$x = a-b/3 + c*2-1$$

When a = 9, b=12 and c =3 the statement becomes

$$x = 9 - 12/3 + 3*2-1$$

and is evaluated as follows

First pass

Step 1: $x = 9-4+3*2-1$

Step 2: $x = 9-4+6-1$

Second pass

Step 3: $x = 5+6-1$

Step 4: $x = 11-1$

Step5: $x = 10$

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

$$9-12/(3+3)*(2-1)$$

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

$$\text{Step1: } 9-12/6*(2-1)$$

$$\text{Step2: } 9-12/6*1$$

Second pass

$$\text{Step3: } 9-2*1$$

$$\text{Step4: } 9-2$$

Third pass

$$\text{Step5: } 7$$

The time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e., equal to the number of arithmetic operations).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis. For example

$$9-(12/(3+3)*2)-1=4$$

whereas

$$9-(12/3)+3*2)-1=-2$$

While parentheses allow us to change the order of priority, we may also use them to improve understand ability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right is evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
- Associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

Program 3.5

Write a C Program for the following expression: $a=5=8\&\& 6!=5$

```
#include<stdio.h>

#include<conio.h>

void main ( )

{

    int a;

    a = 5<=&& 6!=5;

    printf"%d",a);

    getch ();

}
```

Output

1

Fig. 3.6 Program for the expression: $a = 5 < = 8\&\& 6! = 5$

3.13 SOME COMPUTATIONAL PROGRAMS

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:

```
a = 1.0/3.0
```

```
b = a*3.0;
```

we know that $(1.0/3.0)3.0$ is equal to 1. But there is no guarantee that the value of b computed in a program will equal 1.

Another problem is division by zero. On most computers, any attempt to divide a number zero will result in abnormal termination of the program. In one cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.

The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that operands are of the correct type and range, and the result may not produce any overflow of underflow.

Program 3.6 Output of the program in fig.3.7 shows round-off errors that can occur in computation of floating point numbers.

Program

```
/*.....Sum of n terms of 1/n.....*/
```

```
main()
```

```
{
```

```
float sum, n, term;
```

```
int count = 1;
```

```
sum = 0;
```

```

print("enter the value of n\n");

scanf("%f", &n);

term = 1.0/n ;

while(count <= n)

{

cum = sum + term ;

count++ ;

}

printf("sum = %f \n", sum) ;

}

```

output

enter vaule of n

99

sum =1.000011

enter vaule of n

143

sum = 0.9999999

Fig 3.7 round-off errors in floating point computations

We know that the sum of n terms of $1/n$ is 1. However, due to errors in floating point representation, the result is not always 1.

3.14 TYPE CONVERSIONS IN EXPRESSIONS

Implicit type conversions

C permits mixing of constants and variable of different types in an expression. C automatically converts any intermediate value to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as implicit type conversion.

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the ‘lower’ type s is automatically converted to the ‘higher’ type before the operation proceeds. The result is of the higher types.

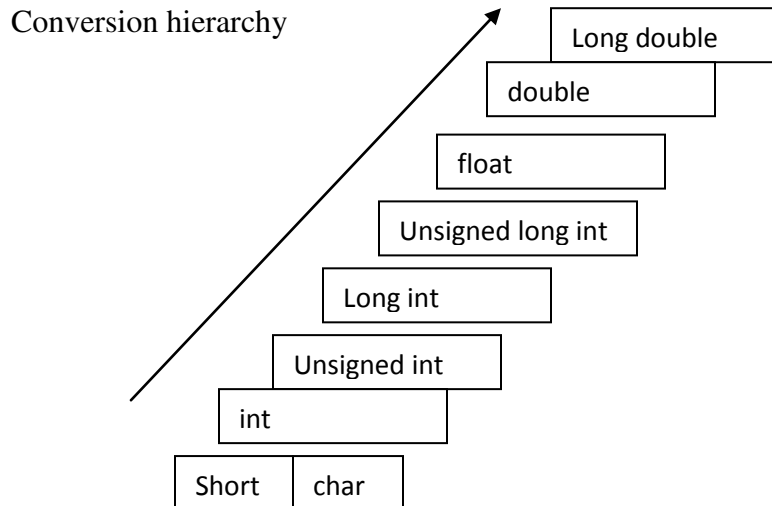
Given below is the sequence of rules that are applied while evaluating expressions.

All short and char are automatically converted to **int**; then

1. If one of the operands is long double, the other will be converted to long double and the result will be long double;
2. Else, if one of the operands is double, the other will be converted to double and the result will be double;
3. Else, if one of the operands is float, the other will be converted to float and the result will be float;
4. else, if one of the operands is unsigned **long int**, the other will be converted to unsigned **long int** and the result will be unsigned **long int**;
5. Else, if one of the operands is long **int** and other is unsigned **int**, then
 - a) If unsigned int can be converted to long **int**, the unsigned **int** operand will be converted as such and the result will be **long int**;
 - b) Else, both operands will be converted to unsigned **long int** and the result will be unsigned long int;
6. Else, if one of the operands is long int, the other will be converted to long int and the result will be long int;
7. Else, if one of the operands is unsigned int, the other will be converted to unsigned int and the result will be unsigned int;

Conversion Hierarchy

Note that C uses the rule that, in all expressions except assignment, any implicit type conversions are made from a lower size type to a higher size type as shown below:



Note that some versions of C automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of the variable of the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

Explicit Conversion

We have just discussed how C performs type conversion automatically. However, there are instances when we want to force a type conversion in way that is different from the automatic conversion.

Consider , for example, the calculation of ratio of females to males in a town.

$$\text{Ratio} = \text{female_number}/\text{male_number}$$

Since `female_number` and `male_number` are declared as integers in the program, the decimal part of the result of the division would be lost and `ratio` would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

Ratio = (float)female_number/male_number

The operator (float) converts the female_number to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (float) affect the value of the variable female number. And also, the type of female number remains as int in the other parts of the program.

The process of such a local conversion is known as explicit conversion or casting a value. The general form of a cast is:

(type-name) expression

Where type-name is one of the standard C data types. The expression may be a constant, variable or an expression) some examples of casts and their actions are shown in Table 3.7

Table 3.7 Use of Casts

Example	Action
x = (int) 7.5	7.5 is converted to integer by truncation
a = (int) 21.3/(int)4.5	Evaluated as 21/4 and the result would be 5
b = (double)sum/n	Division is done in floating point mode
y = (int)(a+b)	The result of a+b is converted to integer
z = (int)a+b'	a is converted to integer and then added to b.
p = cos((double)x)	Converts x to double before using it

Casting can be used to round-off a given value. Consider the following statement:

x = (int)(y+0.5);

If y is 27.6, y+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to x. Of course, the expression, being cast is not changed.

Program 3.7

Figure 3.9 shows a program using a cast to evaluate the equation $\sum_{i=1}^n \left(\frac{1}{i}\right)$

Program

```
main ()
{
float sum :
int    n ;
sum = 0 ;
for (n =1 ; n<=10 ; ++n)
    {
        sum = sum + 1/(float) n;
        printf ("2d %6.4f\n", n, sum);
    }
}
```

Output

```
1. 1.0000
2. 1.5000
3. 1.8333
4. 2.0833
5. 2.2833
6. 2.4500
```

7. 2.5929
8. 2.7179
9. 2.8290
10. 2.9290

Fig. 3.9 Use of a cast

3.15 OPERATOR PRECEDENCE AND ASSOCIATIVITY

As mentioned earlier each operator, in /c has precedence associated with it this precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from 'left to right' or from 'right to left', depending on the level this is in own as the associativity property of an operator table 3.8 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence. Rank 1 indicated the highest precedence level and 15 the lowest. The list also includes those operators, which we have not yet been discussed.

It is very important to not carefully. The order of precedence and associativity of operators, consider the following conditional statement:

$$\text{If}(x == 10 + 15 \ \&\& y < 10)$$

The precedence rules say that the addition operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

$$\text{If}(x == 25 \ \&\& y < 10)$$

The next step is to determine whether x is equal to 25 and y is less than 10. If we assume a value of 20 for x and 5 for y, then

$$x == 25 \text{ is FALSE (0)}$$

$$y < 10 \text{ is TRUE (1)}$$

Note that since the operator < enjoys a higher priority compared to ==, y<10 is tested first and then x ==25 is tested.

Finally we get:

if (FALSE && TRUE)

Because one of the conditions is FALSE, the complex condition is FALSE.

In the case of &&, it is guaranteed that the second operand will not be evaluated if the first is zero and in the case of ||, the second operand will not be evaluated if the first is non-zero.

Table 3.8 Summary of C Operators

Operator	Description	Associativity	Rank
()	Function call	Left to right	1
[]	Array element reference		
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference(indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+	Addition	Left to right	4

-	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		

Rules of Precedence and Associativity

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied

3.16 MATHEMATICAL FUNCTIONS

Mathematical functions such as cos, sqrt, log, etc, are frequently used in analysis of real-life problems. Most of the C compilers support these basic math functions. However, there are systems that have a more comprehensive math library and one should consult the reference manual to find out which functions are available. Table 3.9 lists some standard math functions.

Table 3.9 Math functions

Function	Meaning
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x

Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power= (e ^x)
fabs(x)	Absolute value of x
floor(x)	x rounded down to the nearest integer
fmod(x,y)	remainder of x/y
log(x)	natural log of x,x>0
log10(x)	Base 10 log x,x>0
pow(x,y)	x to the power y (x ^y)
sqrt(x)	Square root of x,x>=0

Note

1. x and y should be declared as **double**
2. In trigonometric and hyperbolic functions, x and y are in radians.
3. All the functions return a double
4. C99 has added float and long double versions of these functions
5. See the appendix 'C99 Features' for details.

As pointed out earlier in Chapter 1, to use any of these functions in a program, we should include the line:

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

in the beginning of the program.

Just Remember

- Use decrement and increment operators carefully, understand the difference between postfix and prefix operations before using them.
- Add parentheses wherever you feel they would help to make the evaluation order clear.
- Be aware of side effect produced by some expressions.
- Avoid any attempt to divide by zero. It is normally undefined. It will either result in a fatal error or in incorrect results.
- Do not forget a semicolon at the end of an expression
- Understand clearly the precedence of operators in an expression, use parentheses, if necessary.
- Associativity is applied when more than one operator of the same precedence are used in an expression. Understand which operators associate from right to left and which associate from left to right
- Do not use increment or decrement operators with any expression other than a variable identifier.
- It is illegal to apply modulus operator `%` with anything other than integers.
- Do not use a variable in an expression before it has been assigned a value
- Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary.
- The result of an expression is converted to the type of the variable on the left of the assigning the value to it. Be careful about the loss of information during the conversion.
- All mathematical functions implement double type parameters and return double type values.
- It is an error if any space appears between the two symbols of the operators `==`, `!=`, `<=` and `>=`.
- It is an error if the two symbols of the operators `!=`, `<=` and `>=` are reversed.
- Use space on either side of binary operator to improve the readability of the code.
- Do not use increment and decrement operators to floating point variables

- Do not confuse the equality operator == with the assignment operator =.

Case Studies

1. Salesman's Salary

A computer manufacturing company has the following monthly compensation policy to their sales-persons:

Minimum base salary	:	1500.00
Bonus for every computer sold	:	200.00
Commission on the total monthly sales	:	2 per cent

Since the prices of computers are changing, the sales price of each computer is fixed at the beginning of every month. A program to compute a sales-person's gross salary is given in Fig.3.10.

Program

```
#define BASE_SALARY 1500.00
#define BONUS_RATE 200.00
#define COMMISSION 0.02
main ()
{
    int quantity ;
    float gross_salary, price;
    float bonus, commission;
    printf("input number sold and price\n");
    scanf("%d %f", &quantity, &price);
    bonus          =    BONUS_RATE* quantity;
    commission     =    COMMISSION * quantity * price;
    gross salary   =    BASE_SALARY +bonus + commission;
    printf("/n");
    printf("Bonus      =    %6.2f/n", bonus);
    printf("commission =    %6.2f/n", commission);
    printf("Gross salary=    %6.2f/n", gross_salary);
```

```

    }
Output
    Input number sold and price
5      20450.00
    Bonus      =      1000.00
    Commission=      2045.00
    Gross salary=      4545.00

```

Fig .3.10 Program of salesman's salary

Given the base salary, bonus, and commission rate, the inputs necessary to calculate the gross salary are, the price of each computer and the number sold during the month.

The gross salary is given by the equation:

$$\text{Gross salary} = \text{base salary} + (\text{quantity} * \text{bonus rate}) + (\text{quantity} * \text{price}) * \text{commission rate}$$

2. Solution of the quadratic equation

An equation of the form

$$ax^2 + bx + c = 0$$

is known as the quadratic equation. The values of x that satisfy the equation are known as the roots of the equation. A quadratic equation has two roots which are given by the following two formulae:

$$\text{root1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$\text{root2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

A program to evaluate these roots is given in Fig. 3.11. The program the user to input the values of a,b and c and outputs root1 and root2.

Program

```
#include<math.h>

main ()
{
    float a, b ,c discriminant,
    root1, root2

    printf("input values of a, b and c\n");

    scanf("%f %f, &a, &b, &c);

    discriminant = b*b-4*a*c;

    if(discriminant < 0)

    printf("\n\nROOTS ARE IMAGINARY\n");

    else

    {

    root1 = (-b + sqrt(discriminant))/(2.0*a);

    root2 = (-b - sqrt(discriminant))/(2.0*a);

    printf("\n\nRoot1 = %5.2f/n/nRoot2 = %5.2f\n",
    root1,root2);

    }

}
```

Output

Input values of a,b and c

2 4 - 16

Root 1 = 2.00

Root 2 = - 4.00

Input values of a,b, and c

1 2 3 ROOT ARE IMAGINARY

Fig .3.11 Solution of a quadratic equation

The term (b^2-4ac) is called the discriminant. If the discriminant is less than zero, its square roots cannot be evaluated. In such cases, the roots are said to be imaginary numbers and the program outputs an appropriate message.

4 MANAGING INPUT AND OUTPUT OPERATIONS

Key Terms

Formatted input | control string | formatted output

4.1 INTRODUCTION

Reading, processing, and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as information or results, on a suitable medium. So far we have seen two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements such as $x =5$; $a=0$; and so on. Another method is to use the input function **scanf** which can read data from a keyboard. We have used both the methods in most of our earlier example programs. For outputting results we have used extensively the function **printf** which sends results out to a terminal.

Unlike other high-level languages, C does not have any built-in input/output statements as part of its syntax. All input/output operation are carried out through function calls such as **printf** and **scanf**. There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the standard I/O library. In

this chapter we shall discuss some common I/O functions that can be used on many machines without any change. However, one should consult the system reference manual for exact details of these functions and also to see what other functions are available.

It may be recalled that we have included a statement

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

In the sample program 5 in chapter 1, where a math library function `cos(x)` has been used. This is to instruct the compiler to fetch the function `cos(x)` from the math library, and that it is not a part of C language. Similarly, each program that uses a standard input/output function must contain the statement

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

at the beginning. However, there might be exceptions. for example, this is not necessary for the functions `printf` and `scanf` which have been defined as a part of the C language.

The file name `stdio.h` is an abbreviation for standard input-output header file. The instruction `#include <stdio.h>` tells the compiler to search for a file named `stdio.h` and place its contents at this point in the program. The contents of the header file become part of the source code when it is compiled.

4.2 READING A CHARACTER

The simplest of all input/output operations is reading a character from the ‘standard input’ unit (usually the keyboard) and writing it to the ‘standard output’ unit (usually the screen). Reading a single character can be done by using the function `getchar`. (this can also be done with the help of the `scanf` function which is discussed in section 4.4). The `getchar` takes the following form:

```
variable_name = getchar ();
```

`variable_name` is a valid C name that has been declared as `char` type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to `getchar` function. Since `getchar` is used on the right-hand side of an assignment statement, the character value of `getchar` is in turn assigned to the variable name on the left. For example

```
char name;  
  
name = getchar ();
```

will assign the character 'H' to the variable name when we press the key H on the keyboard. Since getchar is a function, it requires a set of parentheses as shown.

Program 4.1

The program in fig. 4.1 shows the use of **getchar** function in an interactive environment.

The program displays a question of YES/NO type to the user and reads the user's response in a single character (Y or N), if the response is Y or y, it outputs the message

```
My name is BUSY BEE
```

Otherwise, outputs

```
You are good for nothing
```

Note There is one line space between the input text and output message.

Program

```
#include <stdio.h>  
  
main ()  
{  
  
char answer;  
  
printf("would you like to know my name?\n");  
  
printf("Type Y for YES and N for NO;");  
  
answer = getchar();/*.... reading a character...*/  
  
if (answer == 'y' || answer == 'Y')  
  
printf("\n\nMy name is BUSY BEE\n");
```

```
else  
  
printf(“\n\nYou are good for nothing\n”);  
  
}
```

Output

Would you like to know my name?

Type Y for YES and N for NO; Y

My name is BUSY BEE

Would you like to know my name?

Type Y for YES and N for NO: n

You are good for nothing

Fig. 4.1 use of getchar function to read a character from keyboard

The getchar function may be called successively to read the characters contained in a line of text. For example, the following program segment reads characters from keyboard one after another until the ‘Return’ key is pressed.

```
-----  
-----  
  
char character;  
  
character = ‘ ‘;  
  
while(character != ‘\n’)  
{  
  
    character = getchar ( );  
  
}
```


Warning

The **getchar()** function accepts any character keyed in. This includes RETURN and TAB. This means when we enter single character input, the newline character is waiting in the input queue after **getchar()** returns. This could create problems when we use **getchar(0)** in a loop interactively. A dummy **getchar()** may be used to ‘eat’ the unwanted newline character. We can also use the **fflush** function flush out the unwanted characters.

Note

We shall be using decision statements like if, if..... else and while extensively in this chapter. They are discussed in detail in chapters 5 and 6.

Program 4.2

The program of fig .4.2 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

This program receives a character form the keyboard and tests whether it is a letter or digit and prints out a message accordingly. These tests are done with the help of the following functions.

isalpha(character)

isdigit(character)

for example, **isalpha** assumes a value non-zero (TRUE) if the argument character contains alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function is digit.

Program

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

```
#include <ctype.h>
```

```
main ()
{
    char character;

    printf("press any key\n");

    character = getchar ();

    if (isalpha(character) > 0)/* test for letter */
        printf(" The character is a letter");
    else
        if (isdigit (character) > 0)/* test for digit */
            printf(" The character is a digit");
        else
            printf(" The character is not alphanumeric");
    }
}
```

Output

Press any key

h

The character is a letter.

Press any key

5

The character is a digit

Press any key

*

The character is not alphanumeric.

Fig .4.2 Program to test the character type

C supports many other similar functions, which are given in Table 4.1. These character functions are contained in the file ctype.h and therefore the statement

```
#include <ctype.h>
```

Must be included in the program

Table 4.1 Character Test functions

Function	Test
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case letter?

4.3 WRITING A CHARACTER

Like **getchar**, there is an analogous function **putchar** for writing characters one at a time to the terminal. It takes the form as shown below:

```
putchar (variable_name);
```

where `variable_name` is a type `char` variable containing a character. This statement displays the character contained in the `variable_name` at the terminal. For example, the statements

```
answer = 'Y'  
putchar (answer);
```

will display the character Y on the screen. The statement

```
putchar ('\n');
```

would cause the cursor on the screen to move to the beginning of the next line.

Program 4.3

A program that reads a character from keyboard and then prints it in reverse case is given in fig 4.3. That is, if the input is upper case, the output will be lower case and vice versa.

The program uses three new functions: **islower**, **toupper**, and **tolower**. This function **islower** is a conditional function and takes the value `TRUE` if the argument is a lowercase alphabet; otherwise takes the value `FALSE`. The function **toupper** converts the lowercase argument into an uppercase alphabet while the function **tolower** does the reverse.

Program

```
#include <stdio.h.>  
  
#include <ctype.h>  
  
main ()  
{  
  
    char alphabet;  
  
    printf('Enter an a alphabet');  
  
    putchar('\n'); /* move to next line*/  
  
    alphabet = getchar ();
```

```
if (islower(alphabet));  
  
putchar(toupper(alphabet));/* Reverse and display*/  
  
else  
  
putchar(tolower(alphabet));/* Reverse and display*/  
  
}
```

Output

Enter an alphabet

a

A

Enter a alphabet

Q

Q

Enter an alphabet

z

Z

Fig .4.3 Reading and writing of alphabets in reverse cast

4.4 FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data:

15.75 1233 John

This line contains three pieces of data, arranged in particular form. Such data has to be read conforming to the format of its appearance. For example, the first part of the data should be

read into a variable float the second in tint, and the third part int char. This is possible in 'c using the scanf function. (scanf means scan formatted)

We have already used this input function in a number of examples. Here, we shall explore all of the options that are available for reading the formatted data with scanf function. The general form of scanf is

```
scanf(“ control string” , arg1,arg2,.....argn);
```

The control string specifies the field format in which the data is to be entered and the arguments arg1, arg2, ,,,.....argn specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string (also known as format string) contains field specifications, which direct the interpretation of input data. int may include:

- Field (or format) specifications, consisting of the conversion character %, data type character (or type specifier), and an optional number, specifying the field width.
- Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional. The discussions that follow will clarify these concepts.

Inputting integer numbers

The field specification for reading an integer number is

```
% w sd
```

The percentage sign(%) indicates that a conversion specification follows. W is an integer number that specifies the field width of the number to be read and as, known as data type character, indicates that the number to read is in integer mode, consider the following example:

```
scanf (%d %d, &num1, &num2);
```

Data line:

50 31426

The value 50 is assigned to num1 and 31426 50 to num2. Suppose the input data is as follows:

31426 50

The variable num1 will be assigned 31 (because of %2d) and num2 will be assigned 426 (unread part of 31426). The value 50 that is unread will be assigned to the first variable in the next scanf call. This kind of errors may be eliminated if we use the field specifications without the field width specifications. That is, the statement

```
scanf("%d %d", &num1, &num2);
```

Will read the data

31426 50

Correctly and assign 31426 to num1 and 50 to num2.

Input data items must be separated by spaces, tabs or newlines. Punctuation marks do not count as separators, when the scanf function searches the input data line for a value to be read, it will always bypass any white space characters.

What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away! Also, scanf may skip reading further input.

When the scanf reads a particular value, reading of the value will be terminated as soon as the number of characters specified by the field width is reached (if specified) or until a character that is not valid for the value being read is encountered. In the case of integers, valid characters are an optionally signed sequence of digits.

An input field may be skipped by specifying * in the place of field width. For example, the statement

```
scanf("%d %d", &a &b)
```

Will assign the data

123 456 789

as follows

123 to a

456 skipped (because of *)

789 to b

The data type character d may be preceded by 'l' (letter ell) to read long integers and h to read short integers.

Note

We have provided white space between the field specifications these spaces are not necessary with the numeric input it is a good practice to include the.

Program 4.4

Various input formatting options for reading integers are experimented in the program shown in fig.4.4

Program

```
main ( )  
  
[  
  
int a,b,c x.y.z;  
  
int p,q,r;  
  
printf("Enter three integer numbers\n");  
  
scanf("%d %d %d", &a,&b,&c);  
  
printf("%d %d %d \n\n", a,b,c);
```



```

printf("Enter two 4-digit numbers\n");

scanf("%d %4d", &x,&y);

printf("%d, %d\n\n", x,y);

print("Enter two integers\n");

scanf("%d %d", &a &x);

printf("%d %d /n/n",a,x);

printf("Enter a nine digit number \n");

scanf("%3d %4d %, &q, &r);

printf("%d %d %d /n/n",p,q,r);

printf("Enter two digit number\n")

scanf("%d %d, &x,&y);

printf("%d %d^d, x,y\n");

```

Output

Enter three digit number

1 2 3

1 3 - 3577

Enter tow 4-digit number

6789 4321

67 89

Enter three integer number

1 3 6

```
1 3 -3577
```

Enter two 4 digit numbers

```
6789 4321
```

```
67 89
```

Enter two integers

```
44 66
```

```
4321 44
```

Enter a nine-digit number

```
1234566789
```

```
66n#      1234      567
```

Enter two three – digit number

```
1233      456
```

```
89      123
```

Fig. 4.4 Reading integers using scanf

The first scanf requests input data for three integer values a,b, and c, and accordingly three values 1,2, and 3 are keyed in. Because of the specification `%*d` the value 2 has been skipped and 3 jis assigned to the variable b. Notice that since no data is available for c, it contains garbage.

The second scanf specifies the format `%2d` and `%4d` for the variables x and y respectively. Whenever we specify field width for reading integer number, the input numbers should not contain more digits that the specified size. Otherwise, the extra digits on the right-

hand side will be truncated and assigned to the next variable in the list. Thus, the second scanf has truncated the for digit number 6789 and assigned 67 to x and 89 to y. The value 4321 has been assigned to the first variable in the immediately following scanf statement.

Note: it is legal to use a non-whitespace character between field specifications; however, the scanf expects a matching character in the given location, for example

```
scanf(“%d-%d, &a, &b);
```

accepts input like

```
123-456
```

to assign 123 to a and 456 to b.

Inputting Real Number

Unlike integer numbers, the field width or real numbers in s=not to be specified and therefore scanf reads real numbers using the simple specification %f fro both the notations, namely, decimal point notation and exponential notation. For example, the statement

```
scanf(“%f %f %f”, &x, &y, &z)
```

with the input data

```
475.89 43.21E-1 678
```

will assign the value 475.89 to x, 4.321 to y, and 678.0 to z. The input field specifications may be separated by any arbitrary blank spaces.

If the number to be read is of double type, then the specification should be %lf instead of simple %f. A number may be skipped using %*f specification.

Program 4.5.

Reading of real numbers (in both decimal point and exponential notation) is illustrated in fig4.5

Program

```
main ( )  
  
{  
  
float x,y;  
  
double p,q;  
  
printf("values of x and y:");  
  
scanf("%f %d", &x, &y);  
  
printf("\n");  
  
printf("x = %f/ny = $f\n\n", x,y);  
  
printf("values of p and q:");  
  
scanf("%lf", &p, &q);  
  
printf("/n/np = %.12f/np = %.12e", p, q);  
  
}
```

Output

Values of x and y: 12.3456 17.5e-2

x = 12.345600

y = 0.17500

values of p and q: 4.142857142857 18.5678901234567890

p = 4.142857142857

q = 1.856789012346e+001

Fig 4.5 Reading of real numbers

Inputting Character Strings

We have already seen how a single character can be read from the terminal using the `getchar` function. The same can be achieved using the `scanf` function also. In addition, a `scanf` function can input strings containing more than one character. Following are the specifications for reading character strings:

`%ws` or `%wc`

The corresponding argument should be a pointer to a character array. However, `%c` may be used to read a single character when the argument is a pointer to a `char` variable.

Program 4.6

Reading of strings using `%wc` and `%ws` is illustrated in fig. 4.6

The program in fig. 4.6 illustrates the use of various field specifications for reading strings. When we use `%wc` for reading a string the system will wait until the w^{th} character is keyed in.

Note that the specification `%s` terminates reading at the encounter of a blank space. Therefore, `name2` has read only the first part of 'New York' and the second part is automatically assigned to `name3`. However, during the second run, the string 'New-York' is correctly assigned to `name2`.

Program

```
main ()
{
    int no;

    char name1[15], name2[15], name3[15];

    printf(' Enter serial number and nbase on\n');

    scanf('%d %15c, &no, name1);

    printf('%d %15s\n\n', no, name1);
```

```
printf("Enter serial number and name two\n");  
  
scanf("%d %s", &no, name2);  
  
printf("%d %15s/n/n", no, name2);  
  
printf("Enter serial number and name three\n");  
  
scanf("%d %15s", &no, name3);  
  
printf("%d %15s\n\n", no, name3);  
  
}
```

Output 1

Enter serial number and name one

1 123456789012345

1 123456789012345r

Enter serial number and name two

2 New York

3 New

Enter serial number and name three

2 York

Output 2

Enter serial number and name one

1 123456789012

1 123456789012r

Enter serial number and name two

2 New-York

2 New-York

Enter serial number and name three

3 London

4 London

Fig 4.6 Reading of strings

Some versions of scanf support the following conversion specifications ofr strings:

`%[characters]`

`%[^characters]`

The specification `%[characters]` means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character the specification `%[^characters]` does exactly the reverse.. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

Program 4.7

The program in fig. 4.7 illustrates the function of `%[]` specification.

Program-A

```
main ( )
{
char address[80];

printf('Enter address\n');

scanf('%[a-z]', address);

printf('%-80s\n\n', address);
```

```
}
```

Output

Enter address

New delhi 110002

New delhi

Program-B

```
main ()  
{  
    char address[80];  
    printf("Enter address\n");  
    scanf("%[^/n]", address);  
    printf("%-80s", address);  
}
```

Output

Enter address

New delhi 110 002

New delhi 110 002

Fig 4.7 Illustration of conversion specification %[] for strings

Reading blank spaces

We have earlier seen that `%s` specifier cannot be used to read string with blank spaces. But, this can be done with the help of `%[]` specification. Blank spaces may be included within the brackets, thus enabling the `scanf` to tread strings with spaces. Remember that the lowercase and uppercase letters are distinct. See fig. 4.7

Reading Mixed Data Types

It is possible to use one `scanf` statement to input a data line containing mixed mode data. In such case, care should be exercised to ensure that the input data items match the control specification in order and type. When an attempt is made to read an item that does not match the type expected, the `scanf` function does not read any further and immediately returns the values read. The statement

```
scanf("%d %c %s", &count, &ratio, name);
```

Will read the data

```
15 p 1.575 coffee
```

Correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.

Note: A space before the `%c` specification in the format string is necessary to skip the white space before `p`.

Detection of Errors in Input

When a `scanf` function completes reading its list, it returns the value of number of items that are successively read. This value can be used to test whether any errors occurred in reading the input. For example, the statement

```
scanf("%d %f %s", &a, &b, name);
```

Will return the value 3 if the following data is typed in

```
20 150.25 motor
```

and will return the value 1 if the following line is entered

```
20 motor 150.25
```

This is because the function would encounter a string when it was expecting a floating-point value, and would therefore terminate its scan after reading the first value.

Program 4.8

The program presented in fig. 4.8 illustrates the testing for correctness of reading of data by scanf function.

The function scanf is expected to read three items of data and therefore, when the values for all the three variables are read correctly, the program prints out their values. During the third run, the second item does not match with the type of variable and therefore the reading is terminated and the error message is printed. Same is the case with the fourth run.

In the last run, although data items do not match the variables, no error message has been printed. When we attempt to read a real number for an int variable. The integer part is assigned to the variable, and the truncated decimal part is assigned to the next variable.

Note The character '2' is assigned to the character variable c.

Program

```
main ()
{
int a;
float b;
char c;

printf("Enter values of a, b and c\n");

if (scanf("%d %f %c", &a, &b, &c) == 3)
```

```

printf('a = %d b= %f c=%c\n', a, b, c);

else

printf('Error in input.\n');

}

```

Output

```

Enter values of a,b and c

12 3.45 A

a = 12  b = 3.450000  c = A

Enter values of a, b and c

23 78 9

a = 23  b = 78.0000  c = 9

Enter values of a, b and c

8 A 5.25

Error in input

Enter values of a, b and c

Y 12 67

Error in input.

Enter values of a , b and c

15.75 23 X

a = 15  b = 0.750000 = 2

```

Fig.4.8 Detection of errors in scanf input

Commonly used scanf format codes are given in Table 4.2

Table 4.2 Commonly used scanf Format codes

Code	Meaning
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer'
%i	read a decimal, hexadecimal or
%o	octal integer
%s	read an octal integer
%u	read a string
%x	read an unsigned decimal integer
%[.].]	read hexadecimal integer
	read a string of word(s)

The following letters may be used as prefix for certain conversion characters

- h for short integers
- i for long integers or double
- L for long double

Note: C99 adds some more format codes. See the Appendix 'C99' Features''

Point to remember while using scanf

If we do not plan carefully, some 'crazy' things can happen with scanf. Since the I/O routines are not a part of C languages, they are made available either as a separate module of the C library or as a part of the operating system (like UNIX). New features are added to these routines from time to time as new versions of systems are released. We should consult the

system reference manual before using these routines. Given below are some of the general points to keep in mind while writing a scanf statement.

1. All function arguments, except the control string, must be pointers to variables.
2. Format specifications contained in the control string should match the arguments in order.
3. Input data items must be separated by spaces and must match the variables receiving the input in the same order.
4. The reading will be terminated, when scanf encounters a 'mismatch' of data or a character that is not valid for the value being read.
5. When searching for a value, scanf ignores line boundaries and simply looks for the next appropriate character.
6. Any unread data items in a line will be considered as part of the data input line to the next scanf call.
7. When the field width specifier w is used, it should be large enough to contain the input data size.

Rules for scanf

- Each variable to be read must have a field specification
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a matching character in the user input.
- Never end the format string with whitespace. It is a fatal error!
- The scanf reads until:
 - A whitespace character is found in a numeric specification, or
 - The maximum number of characters have been read or
 - An error is detected, or
 - The end of file is reached

4.5 FORMATTED OUTPUT

We have seen the use of **printf** function for printing captions and numerical results. It is highly desirable that the outputs are produced in such a way that they are understandable and are in an easy-to-use form. It is therefore necessary for the programmer to give careful consideration to the appearance and clarity of the output produced by his program.

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminal. The general form of **printf** statement is:

```
printf('control string', arg1, arg2,.....argn);
```

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. Escape sequence characters such as /n, /t, and /b.

The control string indicates how many arguments follow and what their types are. The arguments arg1, arg2,.....argn are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

A simple format specification has the following form.

```
% w.p type – specifier
```

Where w is an integer number that specifies the total number of columns for the output value and p is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both w and p are optional. Some examples of formatted printf statement are:

```
printf('Programming in C');
```

```
printf(' ');
```

```
printf ('\n');
```

```

printf(“”%d” ,x);

printf(“a = %f/n b = %f”, a, b);

printf(“sum = %d”, 1234);

printf(“\n\n”);

```

printf never supplies a newline automatically and therefore multiple printf statements may be used to build one line of output. A newline can be introduced by the help of a newline character ‘\n’ as shown in some of the examples above.

Output Of Integer Numbers

The format specification for printing an integer number is

% w d

where w specifies the minimum field width for the output. However, if a number is greater than the specified field width it will be printed in full, overriding the minimum specification. d specifies that the value to be printed is an integer. The number is written right-justified in the given field width. Leading blanks will appear as necessary. The following examples illustrate the output of the number 9876 under different formats:

Format	Output
printf(“”%d”, 9876)	9 8 7 6
printf(“”%6d”, 9876)	9 8 7 6
printf(“”%2d”, 9876)	9 8 7 6
printf(“”%-6d”9876)	9 8 7 6
printf(“”%06d”9876)	0 0 9 8 7 6

It is possible to force the printing to be left-justified by placing a minus sign directly after the % character, as shown in the fourth example above. It is also possible to pad with zeros the leading blanks by placing a 0(zero) before the field width specifier as shown in the last item above. The minus(-) and zero (0) are known as flags.

Long integers may be printed by specifying `ld` in the place of `d` in the format specification. Similarly, we may use `hd` for printing short integers.

Program 4.9

The program in fig. 4.9 illustrates the output of integer numbers under various formats.

Program

```
main ( )
{
int m = 12345;
long n= 987654;
printf(“%d\n”,m);
printf(“`%10d\n”,m);
printf(“`%10d\n”,m);
printf(“`%-10d\n”,m);
printf(“`%10ld\n”,n);
printf(“`%10ld\n”,-n);
}
```

Output

```
12345
```



```
12345
0000012345
12345
987654
- 987654
```

Fig. 4.9 Formatted output of integers

Output of Real Numbers

The output of a real number may be displayed in decimal notation using the following specification:

% w p f

The integer *w* indicates the minimum number of positions that are to be used for the display of the value and the integer *p* indicates the number of digits to be displayed after the decimal point (precision). The value, when displayed, is rounded to *p* decimal places and printed right-justified in the field of *w* columns. Leading blanks and trailing zeros will appear as necessary. the default precision is 6 decimal places. The negative numbers will be printed with the minus sign. The number will be displayed in the form [-] mmm-nnn.

We can also display a real number in exponential notation by using the specification:

% w. p. e

The display takes the form

[-] m.nnnnne [-] xx

Where the length of the string of *n*' is specified by the precision. The default precision is 6. The field width *w* should satisfy the condition.

w. p+7

The value will be rounded off and printed right justified in the field of *w* columns.

Padding the leading blanks with zeros and printing with left-justification are also possible by using flags 0 or - before the field width specifier w.

The following examples illustrate the output of the number $y = 98.7654$ under different format specification:

Format	Output
<code>printf("'%7.4f'",y)</code>	9 8 - 7 6 5 4
<code>printf("'%7.2f'",y)</code>	9 8 - 7 7
<code>printf("'%7.2f'",y)</code>	9 8 - 7 7
<code>printf('%f',y)</code>	9 8 - 7 6 5 4
<code>printf("%10.2e",y)</code>	- - 9 - 8 8 E + 0 1
<code>printf("%11.4e",-y)</code>	- 9 - 8 7 6 5 e + 0 1
<code>printf("'%-10.2e'",y)</code>	9 - 8 8 e + 0 1
<code>printf'%e'.y)</code>	9 - 8 7 6 5 4 0 e + 0 1

Some systems also support a special field specification character that lets the user define the field size at run time. This takes the following form:

```
printf('%', 'f', width, precision, number);
```

In the case, both the field width and the precision are given as arguments which will supply the values for w and p. for example,

```
printf('%'. 'f', 7.2, number);
```

Is equivalent to

```
printf(“%7.2f”, number);
```

The advantage of this format is that the values for width and precision may be supplied at run time, thus making the format a dynamic one. For example, the above statement can be used as follows:

```
int width = 7;  
  
int precision = 2;  
  
.....  
  
.....  
  
printf(“%f”, width, precision, number);
```

Program 4.10

All the options of printing a real number are illustrated in fig 4.10

Program

```
main (  
  
{  
  
float y = 98.7654  
  
printf(“%7.4f\n”,y);  
  
printf(“%f\n”,y);  
  
printf(“%7.2f\n”,y);  
  
printf(“%-7.2f\n”,y);  
  
printf(“%07.2f\n”,y);  
  
printf(“%f”,7,2,y);  
  
printf(“\n”);
```

```
printf(“%10.2e\n”,y);  
printf(“%12.4e\n”,y);  
printf(“%-10.2e\n”,y);  
printf(“%e\n”,y);  
}
```

Output

```
98.7654  
98.765404  
98.77  
98.77  
0098.77  
98.77  
9.8e+001  
-9.8765e+001  
9.8e+001  
9.87654e+001
```

Fig 4.10 Formatted output of real numbers

Printing of a Single Character

A single character can be displayed in a desired position using the format

`%wc`

The character will be displayed right-justified in the field of *w* columns. We can make the display left-justified by placing a minus sign before the integer *w*. The default value for *w* is 1.

Printing of String

The format specification for outputting strings is similar to that of real numbers. It is of the form where *w* specifies the field width for display and *p* instructs that only the first *p* characters of the string are to be displayed. The display is right-justified.

The following examples show the effect of variety of specifications in printing a string ‘NEW DELHI 110001’, containing 16 characters (including blanks).

Specification	Output
12345678901234567890	
	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
%s	N E W D E L H I 1 1 0 0 0 1
%20S	N E W D E L H I 1 1 0 0 0 1
%20.10s	N E W D E L H I
%.5s	N E W D
%-20.10s	N E W D E L H I
%5s	N E W D E L H I 1 1 0 0 0 1

Program

```
main ()
{
char x = 'A';

char name[20] = 'ANIL KUMAR GUPTA';
```

```

printf("COURT OF CHARACTERS\n\n");

printf("%c/n%3c/n%5c/n",x,x,x);

printf("%3c/n%c\n",x,x);

printf("\n");

printf("OUTPUT OF STRINGS/n/n");

printf("%s\n",name);

printf("%20s\n",name);

printf("%20.10s\n",name);

printf("%.5s\n",name);

printf("%-20.10s\n",name);

printf("%5c\n",name);

}

```

Output

OUT OF CHARACTERS

A

A

A

A

A

OUTPUT OF STRINGS

ANIL KUMAR GUPTA

ANIL KUMAR GUPTA

ANIL KUMAR

ANIL

ANIL KUMAR

ANIL KUMAR GUPTA

Fig 4.11 Printing of characters and strings

Mixed Data Output

It is permitted to mix data types in one printf statement. For example, the statement of the type

```
printf(“%d %f %s %c”, a,b,c,d);
```

is valid. As pointed out earlier, printf uses its control string to decide how many variables to be printed and what their are. Therefore, the format specifications should match the variables in number, order, and type. If there are not enough variables or if they are of the wrong type, the output result will be correct.

Table 4.3 commonly used printf Format Codes

Code	Meaning
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on

<code>%i</code>	print a signed decimal integer
<code>%o</code>	print an octal integer, without leading zero
<code>%s</code>	print a string
<code>%u</code>	print an unsigned decimal integer
<code>%x</code>	print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion characters.

h for short integers

l for long integers or double

L for long double.

Table 4.4 commonly used Output Format Flags

Flag	Meaning
-	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	Causes leading zeros to appear.
#(with 0 or x)	Causes octal and hex items to be preceded by 0 and Ox, respectively.
#(with e,f or g)	Causes a decimal point to be present in all floating point numbers, even if it is whole number. Also

Note C99 adds some more format codes. See the Appendix ‘C99 Features’.

Enhancing the Readability of Output

Computer outputs are used as information for analysing certain relationships between variables and for making decisions. Therefore the correctness and clarity of outputs are of utmost importance. While the correctness depends on the solution procedure, the clarity depends on the way the output is

presented. Following are some of the steps we can take to improve the clarity and hence the readability and understand ability of outputs.

1. Provide enough blank space between two numbers
2. Introduce appropriate headings and variable names in the output.’
3. Print special messages whenever a peculiar condition occurs in the output.
4. Introduce blank lines between the important sections of the output.

The system usually provides two blank spaces between the numbers. However, this can be increased by selecting a suitable field width for the numbers or by introducing a ‘tab’ character between the specifications. For example, the statement

```
printf(‘a = %d/t b = %d’, a,b);
```

will provide four blank spaces between the two fields. We can also print from them on two separate lines by using the statement

```
printf(‘ a = %d/n b = %d’, a, b);
```

messages and headings can be printed by using the character strings directly in the printf statement.

Examples:

```
printf(‘/n OUTPUT RESULTS /n’);
```

```
printf(‘Code/t Name/t Age/n’);
```

```
printf(‘Error in input data/n’);
```

```
printf(‘Enter your name/n’);
```

Just Remember

- While using getchar function, care should be exercised to clear any unwanted characters in the input stream.
- Do not forget to include <stdio.h> headerfiles when using functions from standard input/output library.

- Do not forget to include <styp.e.h> header file when using functions from character handing library.
- Provide proper field specifications for every variable to be read or printed.’
- Enclose format control strings in double quotes.
- Do not forget to use address operator & for basic type variables in the input list of scanf
- Use double quotes for character string constants.
- Use single quotes for single character constants.
- Provide sufficient field with to handle a value to be printed.
- Be aware of the situations where output may be imprecise due to formatting.
- Do not specify any precision in input field specifications.
- Do not provide any white-space at the end of format string of a scanf statement.
- Do not forget to close the format string in the scan or prinlf statement with double quotes.
- Using an incorrect conversion code for data type being read or written will result in runtime error.
- Do not forget the comma after the format sting in scanf and printf statements.
- Not separating read and writes an argument is an error.
- Do not sue commas in the format string of a scanf statement.
- Using an address operator & with a variables in the printf statement will result in runtime error.

Case Studies

1. Inventory Report

Problem: The ABC Electric Company manufactures four consumer products. Their inventory position on a particular day is given below.

Code	Quantity	Rate(Rs)
F105	275	575.00
H220	107	99.95

I019	321	215.50
M315	89	725.00

It is required to prepare the inventory report table in the following format:

INVENTORY REPORT

Code	Quantity	Rate	Value
-----	-----	-----	-----
-----	-----	-----	-----
-----	-----	-----	-----
-----	-----	-----	-----

Total value:

The value of each item is given by the product of quantity and rate.

Program: The program given in fig.4.12 reads the data from the terminal and generates the required output. The program uses subscripted variables which are discussed in Chapter 7.

Program

```
#define ITEMS 4

main ()

/* BEGIN*/

int i, quantity[5];

float rate[5], value, total_value;

char code[5][5];

/* READING VALUES*/

i = 1;
```

```

while (i <= ITEMS)

{

printf("Enter code, quantity, and rate:");

scanf("%s %f", code[i], &quantity[i], &rate[i];

i++;

}

/* .....printing of Table and Column Heading....*/

printf("\n\n")

printf(".....INVENTORY REPORT   \n");

printf(".....\n");

printf("Code Quantity Rate value  \n");

printf(".....\n");

/* .....Preparation of Inventory Position.....*/

    total value = 0;

    i = 1;

    while (i <= ITEMS)

    {

        Value = quantity[i] * rate[i];

printf("%5s %10d %10.2f %e/n", code[i], quantity[i],

        rate[i], value);

total_value += values;

```

```

i++

}

/*.....Printing of End of Table.....*/

printf(' ..... \n');

printf(' ..... Total Value = %e\n', total_value);

printf(' ..... \n');

}/*END */.

```

Output

```

Enter code, quantity, and rate:F105 275 575.00

Enter code, quantity, and rate:H220 107 99.95

Enter code, quantity, and rate:I019 321 215.50

Enter code, quantity, and rate:M315 89 725.00

```

INVENTORY REPORT

Code	Quantity	Rate	Value
F105	275	575.00	1.581250e+005
H220	107	99.95	1.069465e+004
I019	321	215.50	6.917550e+004
M315	89	725.00	6.452500e+004
Total value			= 3.025202e+005

Fig 4.12 Program for inventory report

2. Reliability Graph

Program : The reliability of an electronic component is given by

$$\text{Reliability (r)} = e^{-\lambda t}$$

where λ is the component failure rate per hour and t is the time of operation in hours. A graph is required to determine the reliability at various operating times, from 0 to 3000 hours. The failure rate λ (lambda) is 0.001.

Problem

```
#include <math.h>

#define LAMBDA 0.001

main ( )

{

double t;

float r;

int i, R;

for (i=1; i<=27;i++)

{

printf(''.....'');

}

printf('\n');

for (t = 0; t<=3000; t+=150)

{

r = exp(- LAMBDA*t);
```

```

R = (int)(50*r+0.5);

printf(' \n');

for (i = 1; i<=R; i++)

{

    printf('*');

}

printf('#\n');

}

for (i=1; i<3;i++)

{

    printf(' |n');

}

}

```

Output

```

-----
I*****#
I*****#
I*****#
I*****#
I*****#
I*****#

```


5 DECISION MAKING AND BRANCHING

Key Terms

Decision-making statements | switch statement | conditional operator | goto statement | infinite loop.

5.1 INTRODUCTION

We have seen that a C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary. However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular conditions has occurred or not and then direct the computer to execute certain statements accordingly.

C language possesses such decision-making capabilities by supporting the following statements:

1. If statements
2. Switch statement
3. Conditional operator statement
4. **goto** statement

These statements are popularly known as decision-making statements. Since these statements ‘control’ the flow of execution, they are also known as control statements.

We have already used some of these statements in the earlier examples. Here, we shall discuss their features, capabilities and applications in more detail.

5.2 DECISION MAKING WITH IF STATEMENT

The if statement is a powerful decision-making statement and is used to control the flow of execution of statements. It is basically a two-way decision statements and us used in conjunction with an expression. It takes the following form

```
if(test expression)
```

It allows the computer to evaluate the expression first and then depending on whether the value of the expression (relation or condition) is 'true'(or non-zero) or 'false' (zero), it transfers the control to a particular statement. This point of program has two paths to follow condition as shown in fig 5.1.

Some examples of decision making, using if statements are:

1. if (bank balance is zero)
borrow money
2. if (room is dark)
put on lights
3. if (code is 1)
person is male
4. If (age is more than 55)
Person is retired

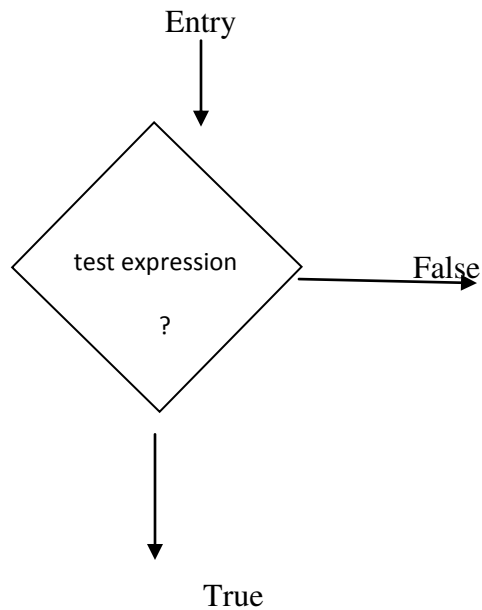


Fig 5.1 Two-way branching

The if statement may be implement in different forms depending on the complexity of conditions to be tested. The different forms are:

1. Simple if statement
2. If.....else statement
3. Nested ifelse statement
4. else if ladder.

We shall discuss each one of them in the next few section.

5.3 SIMPLE IF STATEMENT

The general form of a simple if statement is

If (test expression)

```

{
    Statement-block;
}
Statement -x;

```

The 'statement-block' may be a single statement or a group of statements. If the test expression is true. The statement-block will be executed;

otherwise the statement-block will be skipped and the execution will jump to the statement-x. Remember, when the condition is true both the statement-block and the statement are executed in sequence. This is illustrated in fig 5.2.

consider the following segment of a program that is written for processing of marks obtained in an entrance examination.

.....
.....

```

if(category == SPORTS)
{
    mark = marks + bonus_marks;
}

```

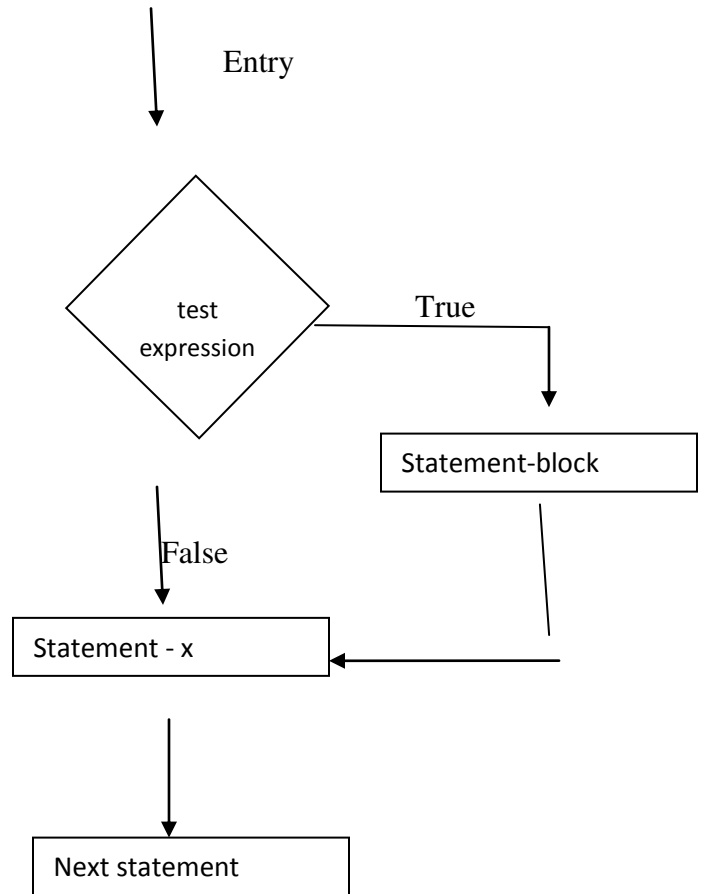


Fig.5.2 Flowchart of simple if control

```
printf(“%f”, marks);
```

.....

.....

The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bonus_marks are added to his marks before they are printed. For others, bonus_marks are not added.

Program 5.1

The program in fig.5.3 reads four values a,b,c and d from the terminal and evaluates the ratio of (a+b) to (c-d) and prints the results. If c-d is not equal to zero.

The program given in fig .5.3 has been run for two sets of data to see that the program function properly. The result of the first run is printed as,

Ratio = -3.181818.

Program

```
main ()
{
    int a, b, c,d
    float ratio

    printf(“Enter four integer values\n”);

    scanf(“%d %d %d, &a, &b, &c, &d);

    if (c-d !=0) / * Execute statement block */.
    {
        ratio = (float)(a+b)/(float)(c-d);
```

```

        printf('Ratio = %f\n', Ratio);
    }
}

```

Output

Enter four integer values

12 23 34 45

Ratio = -3.181818

Enter four integer values

12 23 34 34

Fig 5.3 Illustration of simple if statement

The second run has neither produced any results nor any message. During the second run, the value of (c-d) is equal to zero and therefore, the statements contained in the statement-block are skipped. Since no other statement follows the statement-block, program stops without producing any output.

Note the use of float conversion in the statement evaluating the ratio. This is necessary to avoid truncation due to integer division. Remember, the output of the first run-3.181818 is printed correct to six decimal places. The answer contains a round off error. If we wish to have higher accuracy, we must use double or long double data type.

The simple if is often used for counting purposes. The program 5.2 illustrates this.

Program 5.2

The program has to test two conditions, one for weight and another for height. This is done using the compound relation

```
if (weight < 50 && height > 170)
```

This would have been equivalently done using two if statements as follows:

```
if (weight < 50)

if (height >170)

count = count +1;
```

if the value of weight is less than 50, then the following statement is executed, which in turn is another if statement. This if statement tests height and if the height is greater than 170, then the count is incremented by 1.

Program

```
main ()

{

int count, i;

float weight, height;

count = 0;

printf("Enter weight and height for 10 boys\n");

for (i=1, i<= 10; i++);

{

scanf("%f %f", &weight, &height);

if (weight < 50 && height > 170)

count = count + 1;

}
```

```
printf("Number of boys with weight < 50 kg\n");  
printf("and height > 170 cm = %d\n", count);  
}
```

Output

Enter weight and height for 10 boys

45	176.5
55	174.2
47	168.0
49	170.7
54	169.0
53	170.5
49	167.0
48	175.0
47	167
51	170

Number of boys with weight < 50 kg

and height > 170 cm = 3

Fig .5.4 Use of if for counting

Applying De Morgan's Rule

While designing decision statements, we often come across a situation where the logical NOT operator is applied to a compound logical expression, like $!(x \& \& y \mid z)$. However, a positive

logic is always easy to read and comprehend than a negative logic. In such cases, we may apply what is known as De Morgan's rule to make the total expression positive. This rule is as follows:

'Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators''

That is,

x becomes !x

!x becomes x

&& becomes ||

|| becomes &&

Examples:

!(x && Y ||z) becomes !x || !y && z

!(x <=0 || ! condition) becomes x >0 && condition

5.4 THE IF.....ELSE STATEMENT

The if.... else statement is an extension of the simple if statement. The general form is

```
if (test expression)
{
True-block statement(s)
}
else
{
false-block statement(s)
}
statement-x
```

if the test expression is true, then the true-block statement(s), immediately following the if statements are executed; otherwise, the false-block statement(s) are executed. In either case,

either true-block or false-block will be executed, not both. This is illustrated in fig. 5.5. in both the cases, the control is transferred subsequently to the statement-x

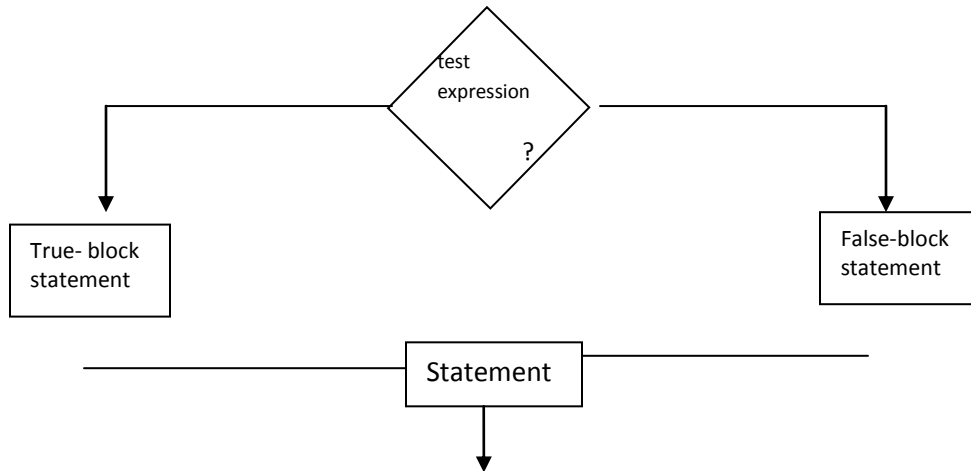


Fig 5.5 Flowchart of ifelse control

Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The program statement to do this may be written as follows:

```

.....
.....
if ( code ==1)
    boy = boy + 1;
if (code ==2 )
    girl = girl + 1;
.....
.....
  
```

The first test determines whether or not the student is a boy. If yes, the number of boys is increased by a and the program continues to the second test. The second test again determines whether the student is a girl. This is unnecessary. Once a student is identified as a boy, there is

no need to test again for a girl. A student can be either a boy or a girl, not both. The above program segment can be modified using the else clause as follows:

```
.....  
.....  
if(code ==1)  
  
    boy = boy +1;  
  
else  
  
    girl = girl + 1;  
  
xxxxxxx  
.....
```

Here, if the code is equal to 1, the statement `boy = boy + 1 ;` is executed and the control is transferred to the statement `xxxxx`, after skipping the else part. If the code is not equal to 1, the statement `boy = boy+1` is skipped and the statement in the else part `girl = girl +1;` is executed before the control reaches the statement `xxxxxxx`.

Consider the program given in fig. .5.3. When the value $(c-d)$ is zero, the ratio is not calculated and the program stops without any message. In such cases we may not know whether the program stopped due to a zero value or some other error. This program can be improved by adding the else clause as follows:

```
.....  
.....  
if(c-d !=0)  
  
    {  
  
        ratio = (float)(a+b)/(float)(c-d);
```

```

        printf("Ratio = %f\n",ratio);
    }

else

printf("c-d is zero\n");

.....

.....

```

Program 5.3

A program to evaluate the power series.

$$e^x = 1 + \frac{x^2}{2!} + \frac{x^2}{3!} + \dots + \frac{x^2}{n!}, 0 < x < 1$$

Is given in fig 5.6 it uses ifelse to test the accuracy.

The power series contains the recurrence relationship of the type

$$T_n = T_{n-1} \left(\frac{x}{n} \right) \text{ for } n > 1$$

$$T_1 = x \text{ for } n=1$$

$$T_0 = 1$$

If T_{n-1} (usually known as previous term) is known, then T_n (known as present term) can be easily found by multiplying the previous term by x/n . Then

$$e^x = T_0 + T_1 + T_2 + \dots + T_n$$

Program

```

#define ACCURACY 0.00001

main ()
{

```

```

int n, count;

float x, term, sum;

printf("Enter value of x;");

scanf("%f", &x);

n = term = sum = count =1;

while (n <+100(

{

term = term * x/n;

sum = sum + term

count = count + 1;

if (term < ACCURACY)

n = 999;

else

n = n+1;

}

Printf("Terms = %d sum = %f\n", count, sum);

}

```

Output

Enter values of f;0

Terms = 2 sum= 1.00000

Enter values of x;0.1

Terms = 5 sum = 1.105171

Enter value of x;0.5

Terms = 7 sum = 1.648720

Enter values of f; 0.75

Terms = 8 sum = 2.116997

Enter value of x;0.99

Terms = 9 sum = 2.691232

Enter value of x;1

Terms = 9 sum = 2.718279

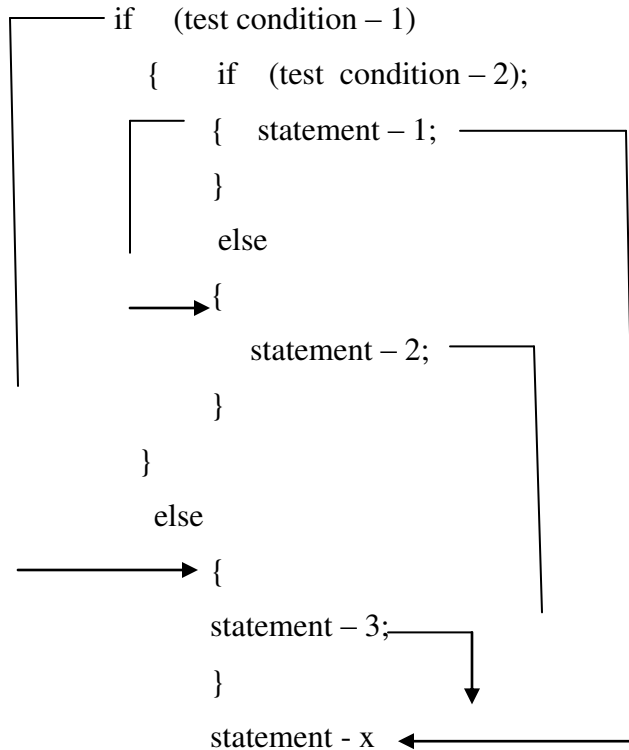
Fig 5.6 Illustration of ifelse statement

The program uses count to count the number of terms added. The program stops when the value of the term is less than 0.0001(ACCURACY). Note that when a term is less than ACCURACY, the value of n is set equal to 999 (a number higher than 100) and therefore the while loop terminates. The results are printed outside the while loop.

5.5 NESTING OF IF ...ELSE STATEMENTS

When a series of decisions are involved, we may have to use more than one if.....else statement in nested form as shown below:

The logic of execution is illustrated in fig. 5.7 if the condition-1 is false, the statement -3 will be executed; otherwise it continues to perform the second test. If the condition-2 is true.



The statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows. A bonus of 2 per cent of the balance held on 31st December is given to everyone, irrespective of their balance, and 5 per cent is given to female account holder's if their balance is more than Rs.5000. this logic can be coded as follows:

```

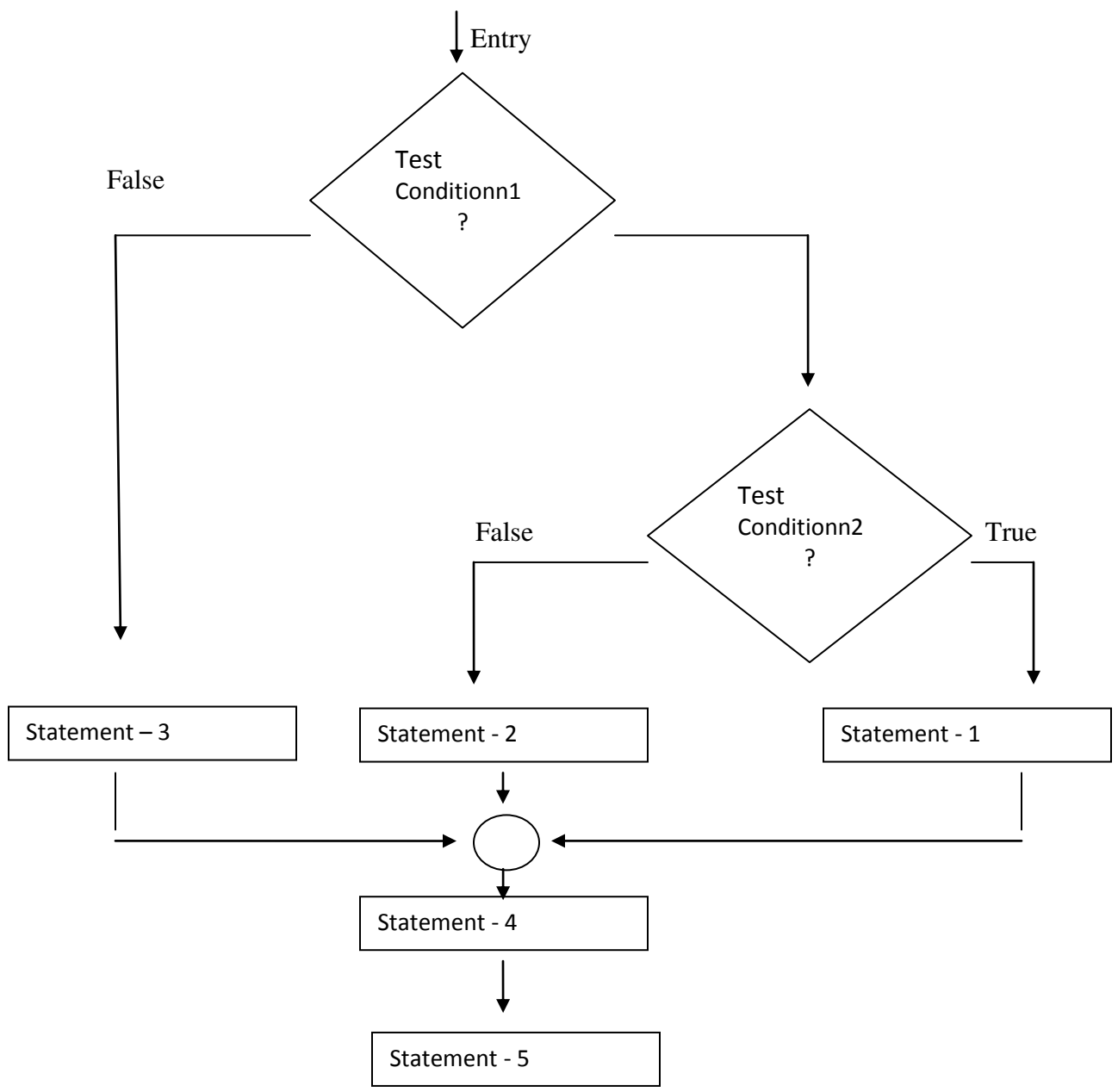
.....
if( sex is female)
{
  if ( balance > 5000)
    Bonus = 0.05 * balance;
  else
    bonus = 0.02 * balance;
}

```

```
else
{
bonus = 0.02 * balance;
}

balance = balance + bonus;

.....
.....
```



When nesting, care should be exercised to match every if with an else. consider the following alternative to the above program⁹which looks right at the first sight);

```
if( sex is female)

if ( balance > 5000)

    bonus = 0.05 * balance;

else

    bonus = 0.02 * balance;

balance = balance +bonus;
```

There is an ambiguity as to over which if the else belongs to. In C, an else is linked to the closest non-terminated. If. Therefore, the else is associated with the inner if and there is no else option for the outer if. This means that the computer is trying to execute the statement

```
balance = balance + bounus
```

without really calculating the bonus for the male account holders.

Consider another alternative, which also looks correct;

```
if(sex is female)

{

if (balance > 5000)

bounus = 0.05 * balance;

}

else

bonus = 0.02 * balance;

balance = balance+bonus;
```


In this case, else is associated with the outer. If and therefore bonus is calculated for the male account holders. However bonus for the female account holders, whose balance is equal to or less than 50000 is not calculated because of the missing else option of the inner **if**

Program 5.4

The program in fig. 5.8 selects and prints the largest of the three numbers using nested if....else statements.

Program

```
main ()
{
float A, B,C;
printf("Enter three values \n");
scanf("%f %f %f", &A, &B, &C);
printf("\nLargest value is ");
if (A>B)
{
If (A>C)
printf("%f\n",A);
else
printf("%f\n",C);
}
else
{
```

```

    if(C>B)

printf(“’%f\n”,C);

else

printf(“’%f\n\n”,B);

    }

}

```

Output

```

Enter three values

23445 67379 88843

Largest value is 88843.000000

```

Fig 5.8 Selecting the largest of three numbers

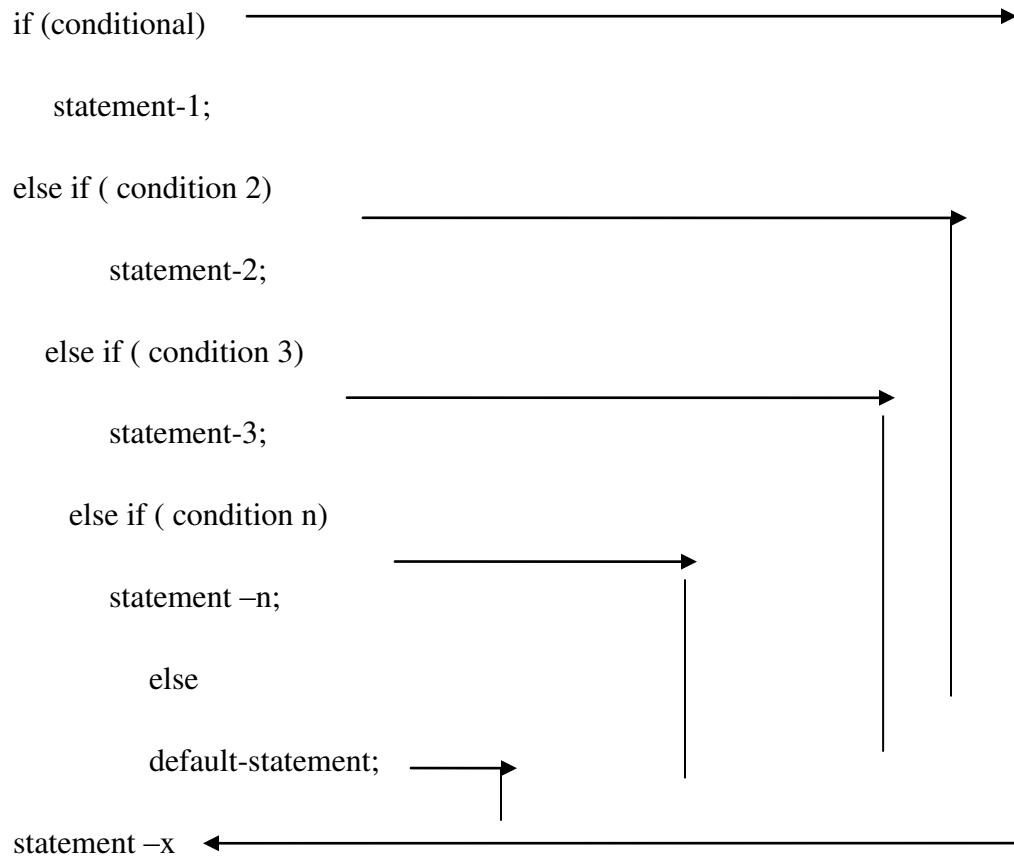
Dangling Else Problem

One of the classic problems encountered when we start using nested if...else statements is the dangling else. This occurs when a matching else is not available for an if. The answer to this problem is very simple. Always match an else to the most recent unmatched if in the current block. In some cases, it is possible that the false condition is not required. In such situations, else statement may be omitted.

‘else is always paired with the most recent unpaired if’

5.6 THE ELSE IF LADDER

There is another way of putting it together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if, it takes the following general form



This construct is known as the else if ladder. The conditions are evaluated from the top (of the ladder), downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder). When all the n conditions become false, then the final else containing the default-statement will be executed. Fig.5.9 shows the logic of execution of else if ladder statements.

Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules

Average marks	grade
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division

40 to 49

Third Division

0 to 39

Fail

This grading can be done using the else if ladder as follows:

```
if(marks >79)
    grade = 'Honour';
else if ( marks >59)
    grade = 'First Division';
else if (marks >49)
    grade = 'Second Division';
else if ( marks >39)
    grade = 'Third Division';
else
    grade = 'Fail';
printf("%ss\n",grade);
```

consider another example given below:

```
-----
-----
if (code ==1)
    colour = 'RED';
else if (code ==2)
    colour = 'GREEN';
```

```
else if ( code ==3)
    colour = 'WHITE';
```

```
-----
-----
```

Code numbers other than 1,2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested if.....else statements.

```
if (code !=1)
    if( code!=2)
        if(code !=3)
            colour = 'YELLOW';
        else
            colour = 'WHITH';
    else
        colour = 'GREEN';
else
    colour = 'RED';
```

In such situations, the choice is left to the programmer. However, in order to choose an if structure that is both effective and efficient, it is important that the programme is fully aware of the various forms of an if statement and the rules governing their nesting.

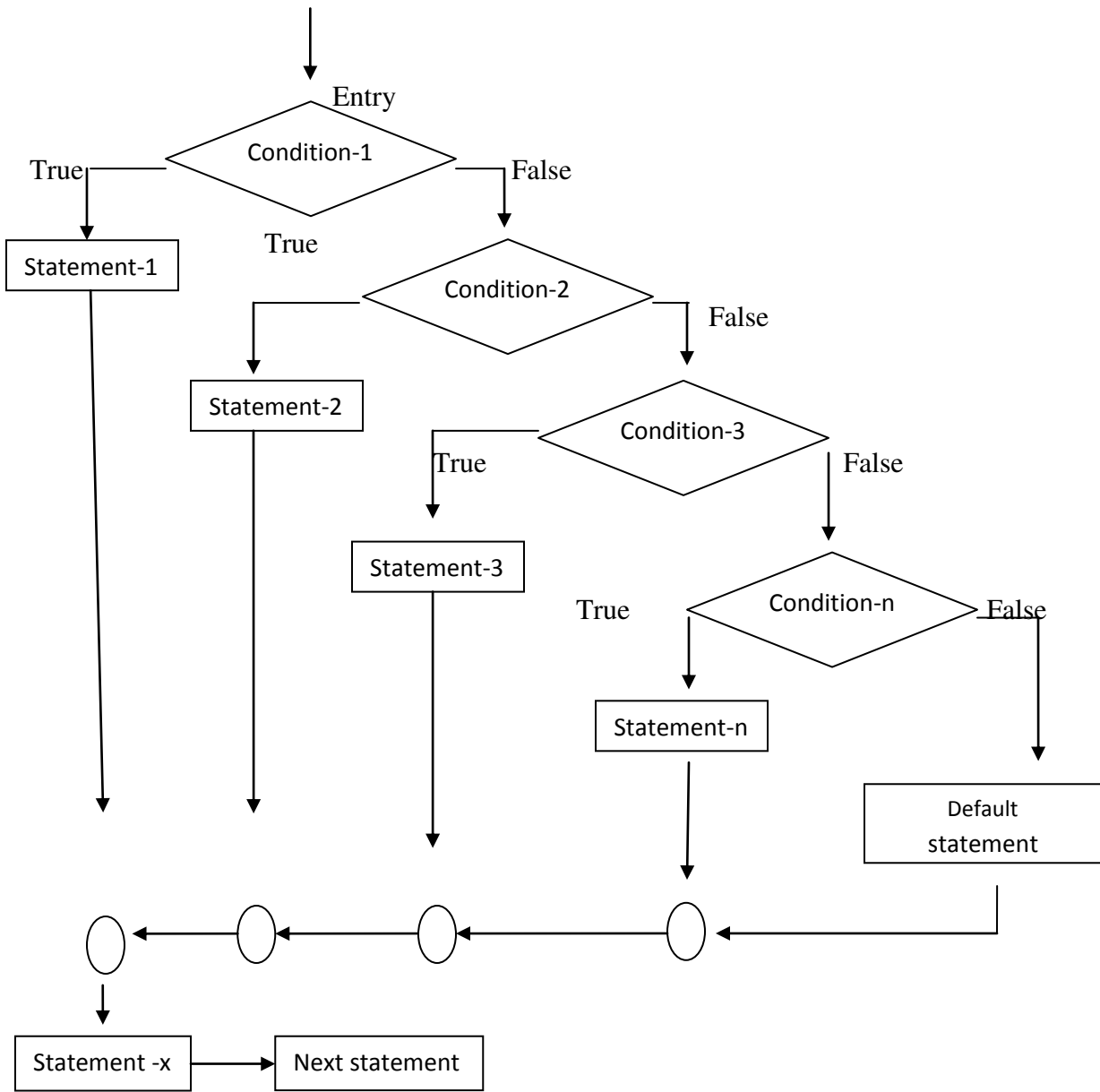


Fig.5.9 Flow chart of else..... if ladder

Program 5.5

An electric power distribution company charges its domestic consumers as follows:

Consumption units	Rate of Charge
0 – 200	Rs. 0.50 per unit

201 -400	Rs. 100 plus Rs. 0.65 per unit excess of 200
401 – 600	Rs. 230 plus Rs. 0.80 per unit excess of 400
601 and above	Rs. 390 plus Rs. 1.00 per unit excess of 600

The program in fig.5.10 reads the customer number and power consumed and print the amount to be paid by the customer.

Program

```

main( )
{

    int units, custnum;

    float charges;

    printf("Enter CUSTOMER NO. and UNITS  consumed\n")
    scanf("%d %d", &custnum, &units);

    if(units <=200)

        charges = 0.5 * units;

    else if (units <=400)

        charges = 100 + 0.65 * (units – 200)

    else if (units <=600)

        charges = 230 + 0.8 * (units – 400);

    else

        charges = 390 + (units – 600);

    printf("\n\n CUSTOMER No: %d: charges = %.2f\n",
        custnum, charges);
}

```

}

Output

Enter CUSTOMER No.UNITS consumed 101 150

Customer No:101 Charges = 75.00

Enter CUSTOMER No. and UNITS consumed 202 225

Customer No: 202 Charges = 116.25

Enter CUSTOMER NO. and UNITS consumed 303 375

Customer No. 303 Charges = 213.75

Enter CUSTOMER NO. and UNITSS consumed 404 520

Customer No. 404 Charges = 326.00

Enter CUSTOMER NO.and UNITS consumed 505 625

Customer no. 505 Charges 415.00

Fig 5.10 Illustration of else.....if ladder

Rules of Indentation

When using control structures, a statement often controls many other statements that follow it. In such situations it is a good practice to use *indentation* to show that the indented statements are dependent on the preceding controlling statement. Some guidelines that could be followed while using indentation are listed below:

- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
- Align vertically else clause with their matching if clause.
- Use braces on separate lines to identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of the braces.
- Align the opening and closing braces.

- Use appropriate comments to signify the beginning and end of blocks.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line

5.7 THE SWITCH STATEMENT

We have seen that when one of the many alternatives is to be selected, we can use an **if** statement to control the selection. However, the complexity of such program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below:

```

switch ( expression)
{
    case value-1;
        block-1
        break;
    case value-2
        block-2
        break;
    .....
    .....
    default;
        default-block

```

```
        break;
    }
    statement=x;
```

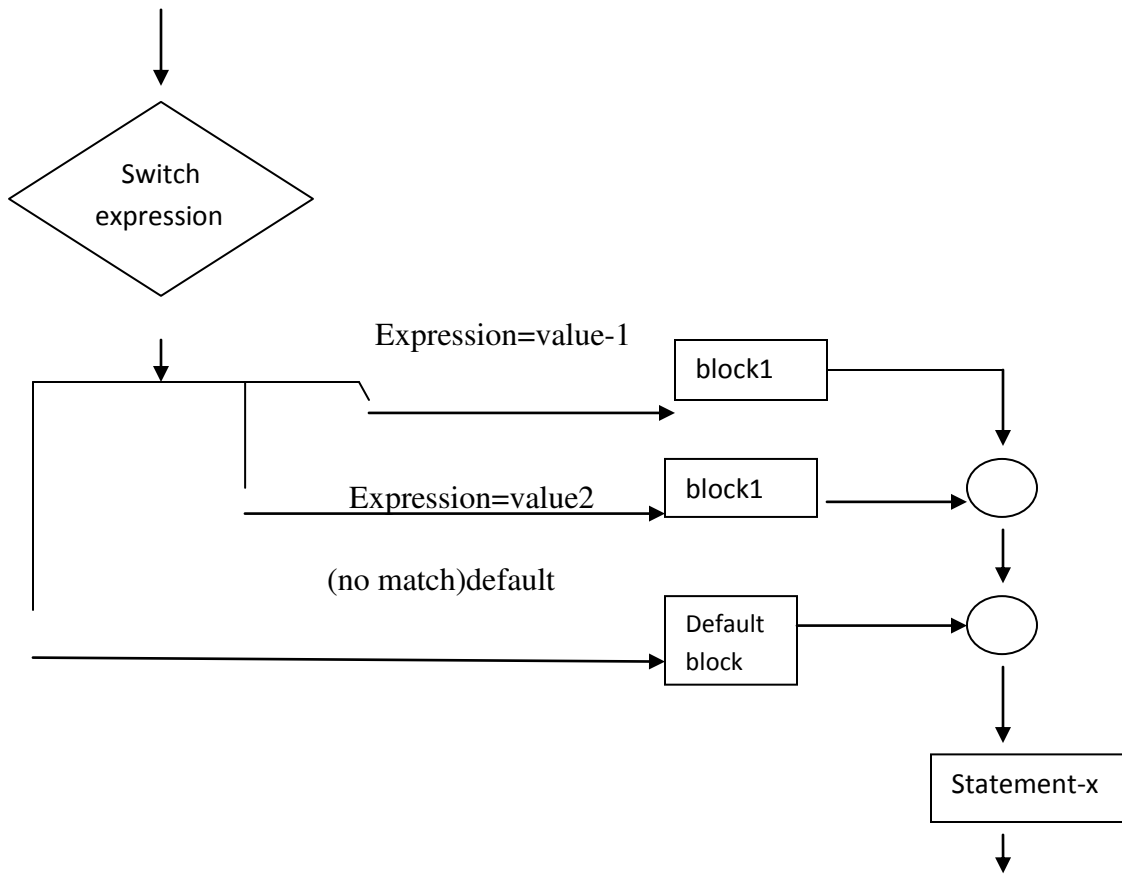
The expression is an integer expression or characters. Value-1, value,2are constants or constant expressions (evaluable to an integral constant) and are known as case labels. Each of these values should be unique within a switch statement. Block-1, block-2....are statement lists and may contain zero or more statement. There is no need to put braces around these blocks. Note that case labels en with a colon(:).

When the **switch** is executed, the value of the expression is successfully compared against the values value-1, value-2.... if a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The break statement at the end of each block signal the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch.

The default is an optional case. When present it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the statement-x (ANSI C permits the use of as many as 257 case labels);

The selection process of switch statement is illustrated in the flow chart shown in fig 5.11



The switch statement can be used to grade the students as discussed in the last section. This is illustrated below:

```

-----
-----
index = marks/10
switch ( index)
{
    case 10;
    case 9;
    case 8;

```

```

        grade = 'Honours';

        break;

case 7:

case 6:

        grade = "First Division";

        break;

case 5;

        grade = 'Second Division';

        break;

case 4;

        grade = 'Third Division';

        break;

default:

        grade = 'Fail';

        break;

}

printf('%s/n', grade);

-----

-----

```

Note that we have used a conversion statement

```
Index = marks / 10;
```

Where, index is defined as an integer. The variable index takes the following integer values.

Marks	Index
100	10
90-99	9
80-89	8
70-79	7
60-69	6
50-59	5
40-49	4
.....
0	0

This segment of the program illustrates two important features. First, it uses empty cases.

The first three cases will execute the same statements

```
grade = 'Honours';  
  
break;
```

Same is the case with 7 and case6. Second, default condition is used for all other cases where a mark is less than 40.

The switch statement is often used for menu selection. For example:

```
-----  
-----  
  
printf("TRAVEL GUIDE\n\n");  
  
printf(" A Air Timings\n");
```

```

printf("T Train Timings\n");

printf("B Bus Service\n");

printf("X To Skip\n");

printf("\n Enter your choice\n");

character = getchar();

switch (character)

{

case 'A' :

    air – display ();

    break;

case 'B' :

    bus-display()

    break;

case 'T' :

    train-dispaly ();

    break;

default :

    printf(" No choice \n");

}

-----

-----

```

It is possible to nest the switch statements. That is, a switch may be part of a case statement. ANSI C permits 15 levels of nesting.

Rules for switch statement

- The switch expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with colon.
- The break statement transfers the control out of the switch statement.
- The break statement is optional. That is, two or more case labels may belong to the same statements.
- The default label is optional. If present, it will be executed when the expression does not find matching case label.
- There can be at most one default label.
- The default may be placed anywhere but usually placed at the end.
- It is permitted to nest switch statements.

Program 5.6

Write a complete C program that reads a value in the range of 1 to 12 and print the name of that month and the next month. Print error for any other input value.

Program

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void main ()

{
```

```

        Char month[12][20] = {"January", "february", "March", April", May",
        "June", "July", "August", September", "October", "November", "December"};

        int i;

        printf("Enter the month value:");

scanf("%d", &i);

if(i<1 || i>12)

{

printf("Incorrect value!!/n Press any key to terminate the program....");

getch();

exit(0);

}

if(i!=12)

printf("%s followed ny %s, month[i-1], month[i]);

else

printf("%s followed by %s", month[i-1], month[0]);

getch();

}

```

Output

Enter the month value: 6

June followed by July

Fig 5.12 Program to read and print name of months in the range of 1 and 12

5.8 THE ? : OPERATOR

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and :, takes three operands. This operator is popularly known as the conditional operator. The general form of use of the conditional operator is as follows:

Conditional expression ? expression 1 : expression 2

The conditional expression is evaluated first. If the result is non-zero, expression1 is evaluated and is returned as the value of the conditional expression. Otherwise, expression2 is evaluated and its value is returned. For example, the segment

```
if ( x <0)
    flag = 0;
else
    flag = 1;
```

can be written as

```
flag = (x <0) ? 0:1
```

Consider the evaluation of the following function:

$$y = 1.5x + 3 \text{ for } x \leq 2$$
$$y = 2x + 5 \text{ for } x > 2$$

This can be evaluated using the conditional operator as follows:

$$Y = (x > 2) ? (2 * x + 5) : (1.5 * x + 3);$$

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

$$\text{Salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written as

$$\text{Salary} = (x \neq 4) ? ((x < 40) ? (4 * x + 100) : (4.5 * x + 150)) : 300;$$

The same can be evaluated using ifelse statements as follows:

```
if ( x <= 40)
    if (x < 40)
        salary = 4 * x + 100;
    else
        salary = 300;
else
    salary = 4.5 * x + 150;
```

When the conditional operator is used, the code becomes more concise and perhaps, more efficient. However, the readability is poor. It is better to use if statements when more than a single nesting of conditional operator is required.

Program 5.7

An employee can apply for a loan at the beginning of every six months, but he will be sanctioned the amount according to the following company rules:

Rule 1: An employee cannot enjoy more than two loans at any point of time.

Rule 2: Maximum permissible total loan is limited and depends upon the category of the employee.

A program to process loan applications and to sanction loans is given in fig 5.13.

Program

```
#define MAXLOAN 50000
```

```

main ()

{

long int loan1, loan2, loan3, sancloan, sum23;

printf("Enter the values of previous two loans :\n");

scanf("%1d %1d, &loan1, &loan2);

printf("\n Enter the value of new loan:\n");

scanf("%1d, &loan3);

sum23 = loan2 + loan 3;

sancloan = (loan1>0) ? 0 : ((sum23>MAXLOAN)?

MAXLOAN – loan2 : loan3);

printf("\n\n");

printf("Previous loans pending:\n%1d\n", loan1,loan2);

printf("Loan requested = %1d\n", loan3);

printf("Loan sanctioned = %1d\n", sancloan);

}

```

Output

Enter the values of previous two loans:

0 20000

Enter the value of new loan:

450000

Previous loans pending:

0 20000

Loan requested = 45000

Loan sanctioned = 30000

Enter the values of previous two loans:

1000 15000

Enter the value of new loan:

25000

Previous loans pending:

1000 15000

Loan requested = 25000

Loan sanctioned = 0

Fig 5.13 Illustration of the conditional operator

The program uses the following variables:

loan3	-	present loan amount requested
loan2	-	previous loan amount pending
loan1	-	previous to previous loan pending
sum23	-	sum of loan2 and loan3
sancloan	-	loan sanctioned

The rules for sanctioning new loan are:

1. loan1 should be zero
2. loan2 + loan 3 should not be more than MAXLOAN.

Note the use of **long int** type to declare variables.

Program 5.8

Write a program to determine the Greatest Common Divisor (GCD) of two numbers.

Algorithm

- Step1 - start
- Step2 - Accept the two numbers whose GCD is to be found(num1, num2)
- Step3 - Call function GCD(num1, num2)
- Step4 - Display the value returned by the function call GCD(num1, num2)
- Step5 - Stop

GCD(a,b)

- Step1 - Start
- Step2 - If $b > a$ goto step 3 else goto step 4
- Step3 - Return the result of the function call GCD(b,a) to the calling function
- Step4 - If $b = 0$ goto Step 5 else goto step6
- Step5 - Return the value a to the calling function
- Step6 - Return the result of the function call GCD(b,a ,mod b) to the calling function

Program

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

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

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

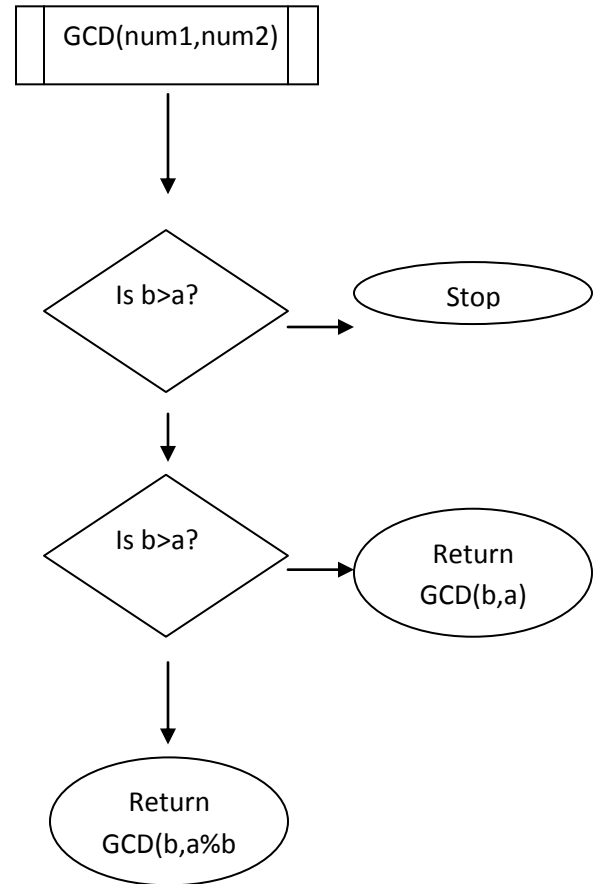
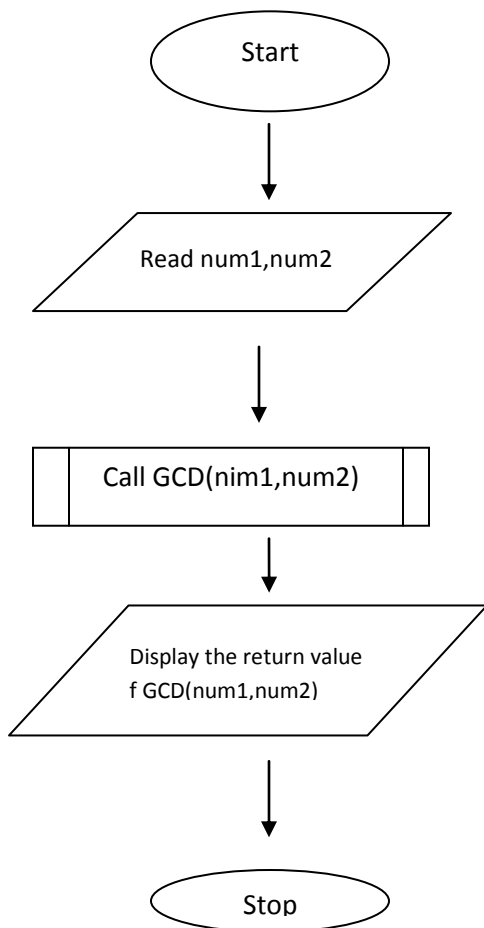
```
int GCD (int m, int n);
```

```
void main ()
```

```
{
```

```
    int num1, num2;
```

```
    clrscr( );
```



```
printf("Enter the two numbers whose GCD is to be found:");
```

```
scanf("%d %d", &num1, &num2);
```

```
printf("\nGCD of %d and %d is %d\n", num1, num2, GCD( num1, num2));
```

```
getch();
```

```

    }

int GCD(int a , int b)

{

    If(b>a)

        return GCD(b,a);

    if (b==0)

        return a;

    else

        return GCD(b,a%b);

}

```

Output

Enter the two numbers whose GCD is to be found: 18 12

GCD of 18 and 12 is 6

Fig 5.14 Program to determine GCD of two numbers

Some Guidelines for Writing Multiway Selection Statements

Complex multiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are some guidelines that would help improve readability and facilitate maintenance.

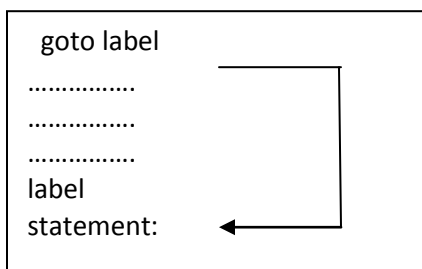
- Avoid compound negative statements. Use positive statements wherever possible.
- Keep logical expressions simple. We can achieve this using nested if statements, if necessary (KISS – Keep It Simple and Short).
- Try to code the normal/anticipated condition first.

- Use the most probable condition first. This will eliminate unnecessary tests, thus improving the efficiency of the program.
- The choice between the nested if and switch statements is a matter of individual's preference. A good rule of thumb is to use the switch when alter-native paths are three to ten.
- Use proper indentations (See Rules for Indentation).
- Have the habit of using default clause in switch statements.
- Group the case labels that have similar actions.

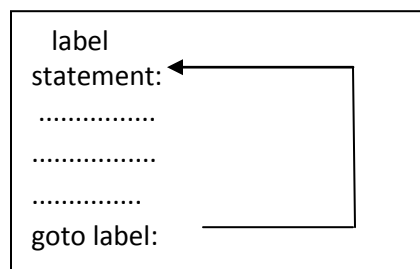
5.9 THE GOTO STATEMENT

So far we have discussed ways of controlling the flow of execution based on certain specified conditions. Like many other languages, C supports the goto statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of **goto** might be desirable.

The **goto** requires a label in order to identity the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and label statements are shown below:



En Forward jump



Backward jump

The label can be anywhere in the program either before of after the goto label: statement.

During running of a program when a statement like


```
goto begin;
```

is met, the flow of control will jump to the statement immediately following the label begin. This happens unconditionally.

Note that a goto breaks the normal sequential execution of the program. If the label is before the statement goto label; a loop will be formed and some statement will be executed repeatedly, such a jump is known as backward jump. On the other hand, if the label; is placed after the goto label; some statements will be skipped and the jump is known as forward jump.

A goto is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example;

```
main ( )  
  
    {  
  
    double x,y  
  
    read;  
  
    scanf("%f", &x);  
  
    if (x<0) goto read;  
  
    y = sqrt(x)  
  
    print("%f %f\n", x, y);  
  
    goto read;  
  
    }
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two goto statements. One at the end, after printing the results to transfer the control back to the input statement and the other to skip any other computation when the number is negative.

Due to the unconditional goto statement at the end the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an infinite loop. The computer goes round and round until we take some special steps to terminate the loop. Such infinite loops should be avoided program 5.9 illustrates how such infinite loops can be eliminated.

Program 5.9

Program presented in fig.5.15 illustrates the use of the goto statement. The program evaluates the square root for five numbers; the variable count keeps the count of numbers read. When count is less than or equal to 5, goto read; directs the control to the label read; otherwise, the program prints a message and stop.

Program

```
#include<math.h>

main ( )
{
    double x, y;

    int count;

    count = 1;

    printf("Enter FIVE real values in a LINE \n");

    read:

        scanf("%lf", &x);

        printf("\n");

        if( x<0)

            printf("Value - %d is negative\n", count);
```

```

else
{
y = sqrt(x);
printf(“%1ft %1f\n”, x, y);
}
count = count +1;
if (count <= 5)
goto read;
    printf(“\nEnd of computation”);
}

```

Output

Enter FIVE real values in a LINE

50.70 40 -36 75 11.25

50.750000 7,123903

40.00000 6.324555

Value -3 is negative

75.00000 8.6660254

11.25000 3.354102

Fig 5.15 Use of the goto statement

Another use of the goto statement is to transfer the control out of a loop (or nested loops) when certain peculiar conditions are encountered. Example:

```

-----
-----
while (-----)
{
    for(-----)
    {
        -----
        -----
        if(-----)goto end_of_program:
        -----
    }
    -----
    -----
}

end_of_process:

```

we should try to avoid using goto as far as possible. But there is nothing wrong, if we use it to enhance the readability of the program or to improve the execution speed.

Just Remember

- Be aware of dangling else statements.
- Be aware of any side effect in the control expression such as if(x++)
- Use braces to encapsulate the statements in if and else clauses of an if.....else statement.
- Check the use of = operator in place of the equal operator ==.

- Do not give any spaces between the two symbols of relational operators =, >=, <=, >= and <=.
- Writing !=, >= and <= operators like =!, >= and =< is an error.
- Remember to use two ampersands (&&) and two bars(!!) for logical operators. Use of single operators will result in logical errors.
- Do not forget to place parentheses for the if expression
- It is an error to place a semicolon after the if expression
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal.
- Do not forget to use a break statement when the cases in a switch statement are exclusive.
- Although it is optional, it is a good programming practice to use the default clause in a switch statement
- It is an error to use a variable as the value in a case label of a switch statement. (only integral constants are allowed.)
- Do not use the same constant in two case labels in a switch statement.
- Avoid using operands that have side effects in a logical binary expression such as (x-&&++y). The second operand may not be evaluated at all.
- Try to use simple logical expressions.

Case studies

1. Range of Numbers

Problem: A survey of the computer market shows that personal computers are sold at varying cost by the vendors. The following is the list of costs (in hundreds) quoted by some vendors:

35.00,	40.50,	25.00	31.25	68.15,
47.00	26.65	29.00	53.45	62.50

Determine the average cost and the range of values.

Problem analysis: Range is one of the measures of dispersion used in statistical analysis of a series of values. The range of any series is the difference between the highest and the lowest values in the series. That is

$$\text{Range} = \text{highest value} - \text{lowest value}$$

It is therefore necessary to find the highest and the lowest values in the series.

Program:

A program to determine the range of values and the average cost of a personal computer in the market is given in fig. 5.16.

Program

```
main ()
{
    int count;
    float value, high low, sum, average, range;
    sum = 0;
    count = 0;
    print("Enter numbers in a line :
        input a NEGATIVE number to end\n");
```

input:

```
scanf("%f", &value);
if(value < 0) goto output;
    count = count +1
if( count ==1)
```

```

        high = low = value
else if (value > high)
    high = value;
else if ( value < low)
    low = value;
sum = sum + value;
goto input;

```

Output:

```

average = sum/count;
range = high – low;
printf(“\n\n”);
printf(“Total values : %d\n”, count);
printf(“Highest-value: %f\nlowest-value : %f\n”,
    high, low);
printf(“Range : %f\nAverage s : %f\n”,
    range, average);
}

```

Output

Enter numbers in a line : input a NEGATIVE number to end

35 40.50 25 31.25 68.15 47 26.65 29 53.45 62.50 -1

Total values : 10

Highest – values : 68.150002

Lowest – value : 25.00000

Range : 43.150002

Average : 41.849998

Fig .5.16 Calculation of range of values

When the value is read the first time, it is assigned to two buckets, high and low, through the statement

```
high = low = value;
```

For subsequent values, the value read is compared with high; if it is larger, the value is assigned to high. Otherwise, the value is compared with low, if it is smaller, the value is assigned to low. Note that at a given point, the buckets high and low hold the highest and the lowest values read as far.

The values are read in a loop created by the goto input statement. The control is transferred out of the loop by inputting a negative number, this is caused by the statement

```
if(value <=0) goto output;
```

Note that this program can be written without using goto statements. Try.

2. Pay-Bill Calculations

Problem: A manufacturing company has classified its executives into four levels for the benefit of certain perks. The levels and corresponding perks are shown below:

Level	Perks	
	Conveyance	Entertainment allowance
1	1000	500
2	750	200

3	500	100
4	250	---

An executive's gross salary includes basic pay, house rent allowance at 25% of basic pay and other perks. Income tax is withheld from the salary on a percentage basis as follows:

Gross salary	Tax rate
Gross \leq 2000	No tax deduction
2000 < Gross \leq 4000	3%
4000 < Gross \leq 5000	5%
Gross > 5000	8%

Write a program that will read an executive's job number, level number and basic pay and then compute the net salary after withholding income tax.

Problem analysis:

$$\text{Gross salary} = \text{basic pay} + \text{house rent allowance} + \text{perks}$$

$$\text{Net salary} = \text{Gross salary} - \text{income tax.}$$

The computation of perks depends on the level, while the income tax depends on the gross salary. The major steps are:

1. Read data.
2. Decide level number and calculate perks
3. Calculate gross salary
4. Calculate income tax.
5. Compute net salary.
- 6 Print the results.

Program:

A Program and the results of the test data are given in Fig. 5.17. Note that the last statement should be an executable statement. That is, the label stop: cannot be the last line.

Program

```
#define CA1 1000
#define CA2 750
#define CA3 500
#define CA4 250
#define EA1 500
#define EA2 200
#define EA3 100
#define EA4 0
main ()\
{
    int level, jobnumber;

    float gross;

        basic,
        house_rent,
        perks,
        net,
        incometax;

input:

    printf("\nEnter level, job number, and basic pay\n");

    printf("Enter 0(zero) for level to END\n\n");
```

```
scanf("%d", &level);

if (level ==0) goto stop;

scanf("%d %f", &jobnumber, &basic);

switch (level)

{

case 1:

    Perks = CA1 + EA1;

    Break;

case 2;

    perks = CA2 + EA2;

    break;

case 3:

    perks = CA3 + EA3;

    break;

case 4:

    perks = CA4 + EA4;

    break;

default;

    printf("Error in level code\n");

    goto stop;
```

```

}

house_rent = 0.25 * basic;

gross = basic + house_rent + perks;

if (gross <= 2000)

    incometax = 0;

else if (gross <= 4000)

    incometax = 0.03 * gross;

    else if (gross <=5000)

        incometax = 0.05 * gross;

    else

        incometax = 0.08 * gross;

net = gross – incometax;

printf(“%d %dn %.2f\n” , level, jobnumber, net);

goto input;

stop:  printf(“\n\nEND OF THE PROGRAM”);

}

```

Output

Enter level, job number, and basic pay

Enter 0(zero) for level to END

1 1111 4000

1 1111 5980.00

Enter level, job number, and basic pay

Enter 0 (zero) for level to END

2 2222 3000

2 2222 4465.00

Enter level, job number, and basic pay

Enter 0 (zero) for level to END

3 3333 2000

3 3333 3007.00

Enter level, job number, and basic pay

Enter 0 (zero) for level to END

4 4444 1000

4 4444 1500.00

Enter level, job number, and basic pay

Enter 0 (zero) for level to END

0

END OF THE PROGRAM

Fig 5.17 Pay-bil calculations

6 DECISION MAKING AND LOOPING

6.1 INTRODUCTION

We have seen in the previous chapter that it is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the **if** statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10, we can write a program using the **if** statement as follows:

```
.....  
.....  
sum = 0;  
  
n = 1;  
  
loop:  
sum = sum + n*n;  
  
if (n == 10)  
goto print;  
  
else  
{  
    n = n+1;           n = 10,  
    goto loop;        end of loop  
}  
  
print:  
  
.....  
.....
```

This program does the following things:

1. Initializes the variable *n*.
2. Computes the square of *n* and adds it to *sum*.
3. Tests the value of *n* to see whether it is equal to 10 or not. If it is equal to 10, then the program prints the results.
4. If *n* is less than 10, then it is incremented by one and the control goes back to compute the sum again.

The program evaluates the statement

sum = sum + n*n;

10 times. That is, the loop is executed 10 times. This number can be increased or decreased easily by modifying the relational expression appropriately in the statement **if** (*n*==10). On such occasions where the exact number of repetitions is known, there are more convenient methods of looping in C. These looping capabilities enable us to develop concise programs containing repetitive processes without the use of **goto** statements.

In looping, a sequence of statements is executed until some conditions for termination of the loop are satisfied. A program loop therefore consists of two segments, one known as the *body of the loop* and the other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the *entry-controlled* loop or as the *exit-controlled loop*. The flow charts in Fig.6.1 illustrate these structures. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. The entry-controlled and exit-controlled loops are also known as pre-test and post-test loops respectively.

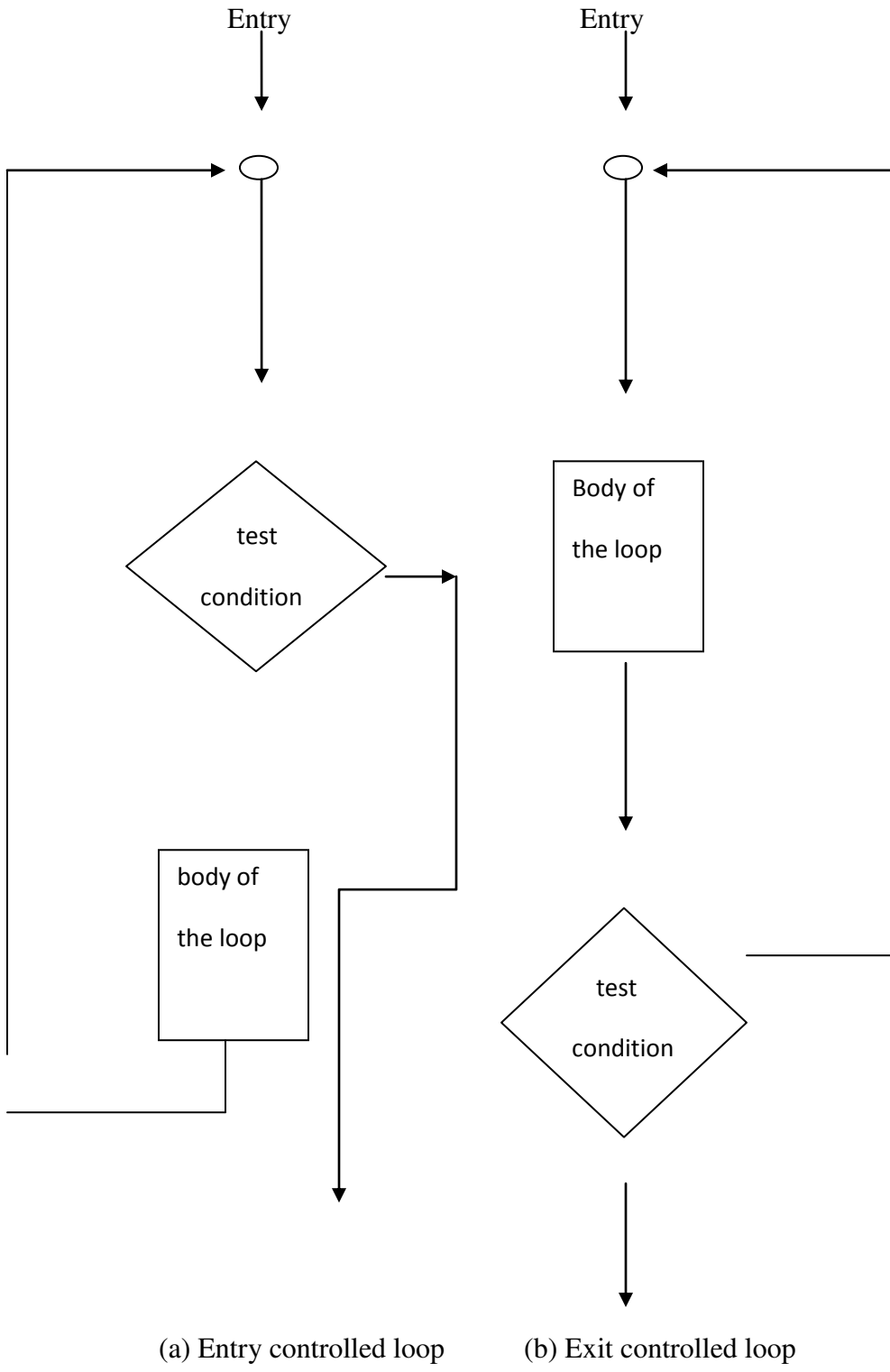


Fig. 6.1 Loop control structures

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control sets up an infinite loop and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides for three constructs for performing loop operations. They are:

1. The **while** statement.
2. The **do** statement.
3. The **for** statements

We shall discuss features and application of each of these statements in this chapter.

Sentinel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control

Expression, the loops may be classified into two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops

When we know in advance exactly how many times the loop will be executed, we use a counter-controlled loop. We use a control variable known as counter. The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we

want to execute the loop may be a constant or a variable that is assigned a value. A counter-controlled loop is sometimes called *definite repetition loop*.

In a sentinel-controlled loop, a special value called a sentinel value is used to change the loop control expression from true to false. For example, when reading data we may indicate the “end of data” by a special value, like -1 and 999. The control variable is called a **sentinel variable**. A sentinel controlled loop is often called indefinite repetition loop because the number of repetitions is not known before the loop begins executing.

6.2 THE WHILE STATEMENT

The simplest of all the looping structures in C is the while statement. We have used while in many of our earlier programs. The basic format of the while statement is

```
while (test condition)  
  
  {  
  
    body of the loop  
  
  }
```

The **while** is an entry-controlled loop statement. The *test-condition* is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop.

On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement. We can rewrite the program loop discussed in Section 6.1 as follows:

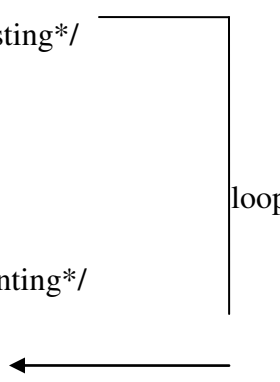
=====

```
sum = 0;

n = 1; /*Initialization*/

while(n <= 10)/*testing*/
{
    sum = sum + n * n;
    n = n+1; /*incrementing*/
}

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



=====

The body of the loop is executed 10 times for n- 1,2,...10, each time adding the square of the value of n, which is incremented inside of the loop. The test condition may also be written as n<11; the result would be the same. This is a typical example of counter-controlled loops. The variable n is called **Counter or control variable**.

Another example of **while** statement, which uses the keyboard input is shown below:

=====

```
character = ' ' ;

while (character != 'y')

character = getchar( );

XXXXXXXXXXXXX;
```

=====

First the **character** is initialized to ' '. The **while** statement then begins by testing whether character is not equal to Y. Since the **character** was initialized to ' ', the test is true and the loop statement

```
character = getchar( );
```

is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed. When Y is pressed, the condition becomes false because character equals Y, and the loop terminates, thus transferring the control to the statement xxxxx; this is a typical example of sentinel-controlled loops. The character constant 'y' is called sentinel value and the variable character is the condition variable, which often referred to as the sentinel variable.

Program 6.1 A program to evaluate the equation $y=x^n$ when n is a non-negative integer, is given in Fig.6.2

The variable y is initialized to 1 and then multiplied by x, n times using the **while** loop, The loop control variable count is initialized outside the loop and incremented inside the loop. When the value of count becomes greater than n, the control exists the loop.

Program

```
main ( )
{
    int count, n;
    float x, y;
    printf("Enter the values of x and n : ");
    scanf("%f%d", &x, &n);
    y = 1.0;
    count = 1;          /*initialization*/
```

```

/*loop begins*/

while (count <= n)    /*testing*/

    {

        y = y*x;

        count++;      /* incrementing*/

    }

/*end of loop*/

printf("\ns = %f; n = %d; x to power n = %f\n",x,n,y);

}

```

Output

Enter the values of x and n: 2.5 4

X = 2.500000; n =4; x to power n= 39.062500

Enter the values of x and n: 0.5 4

X = 0.500000; n = 4; x to power n = 0.62500

Fig 6.2 Program to compute x to the power n using while loop

6.3 THE DO STATEMENT

The **while** loop construct that we have discussed in the previous section, makes a test of condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement. This takes the form:

```

do

{

```

body of the loop

}

while (test-condition);

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test-condition in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the test-condition is evaluated at the bottom of the loop, the **do...while** construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

A simple example of a **do...while** loop is:

```
do
{
printf("input a number\n");
number = getnum ( );
}
while (number > 0);
.....
```

This segment of a program reads a number from the keyboard until a zero or a negative number is keyed in, and assigned to the sentinel variable **number**.

The test conditions may have compound relations as well. For instance, the statement **while (number > 0 && number < 100);**

in the above example would cause the loop to be executed as long as the number keyed in lies between 0 and 100.

Consider another example:

```

=====
i = 1;                                /*Initializing*/
sum = 0;
Do
{
sum = sum + i;
I = I+2;                               /*Incrementing*/
}
while (sum >= 40 || i < 10);          /* Testing*/
printf(“%d %d\n”, i, sum);
=====

```

The loop will be executed as long as one of the two relations is true.

Program 6.2 A program to print the multiplication from 1*1 to 12*10 as shown below is given

Fig.6.3

```

1 2 3 4 .....10
2 4 6 8 .....20
3 6 9 12 .....30
4
-

```

12120

This program contains two do...while loops in nested form. The outer loop is controlled by the variable row and executed 12 times. The inner loop is controlled by the variable column and is executed 10 times, each time the outer loop is executed. That is, the inner loop is executed a total of 120 times, each time printing a value in the table.

Program:

```
#define COLMAX 10
```

```
#define ROWMAX 12
```

```
main()
```

```
    int row, column, y;
```

```
    row = 1;
```

```
    printf(" multiiplication table \n");
```

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

```
    do /*.....OUTER LOOP BEGINS.....*/
```

```
    {
```

```
        column = 1;
```

```
        do /*.....INNERLOOP BEGINS.....*/
```

```
        {
```

```
            y = row * column;
```

```
            printf("%4d", y);
```

```
            Column = column + 1;
```



```

        }

while (column <= COLMAX);

/* ...INNER LOOP ENDS...*/

printf("\n");

row = row + 1;

}

while ( row <= ROWMAX);

/* .....OUTERLOOP ENDS.....*/

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

```

Output

MULTIPLICATION TABLE

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90

```

10 20 30 40 50 60 70 80 90 100
11 22 33 44 55 66 77 88 99 110
12 24 36 48 60 72 84 96 108 120

```

Fig 6.3 Printing of a multiplication table using **do.....while** loop

Notice that the `printf` of the inner loop does not contain any new line character (`\n`). this allows the printing of all row values in one line. The empty `printf` in the outer loop initiates a new line to print the next row.

6.4 THE FOR STATEMENT

Simple ‘for’ loops

The **for** loop is another entry-controlled loop that provides a more concise loop control structure. The general form of the **for** loop is

```

for ( initialization ; test-condition ; increment )
{
    Body of the loop
}

```

The execution of the **for** statements as follows:

1. *Initialization* of the control variable is done first, using assignment statements such as `i = 1` and `count = 0`. The variables **i** and **count** are known as loop-control variables.
2. The value of the control variable is tested using the test-condition. The test-condition is a relational expression, such as `i < 10` that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the controls transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is incremented

using an assignment statement such as $I = i + 1$ and the new value of the controls variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Note: C99 enhances the **for** loop by allowing declaration of variables in the initialization permits portion. See the appendix “C99 Features”.

Consider the following segment of a program

```
    for ( x = 0 ; x <= 9 ; x = x+1)
    {
        printf(“%d”, x);
    }

    printf(“\n”);
```

this for loop is executed 10 times and prints the digit 0 to 9 in one line. The three section enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the increment section, $x = x + 1$.

The **for** statement allows for negative increments. For example, the loop discussed above can be written follows:

```
for ( x = 9 ; x > 0 ; x = x-1 )

printf(“%d”, x);

printf(“\n”);
```

This loop is executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for (x = 9; x < 9; x = x-1)

printf(“%d”, x);
```

Will never be executed because the test condition fails at the very beginning itself.

Let us again consider the problem of sum of squares of integers discussed in section 6.1. this problem can be coded using the for statements follows:

```
-----

sum = 0;

    for (n = 1; n <= 10; n = n+1)

        {

            sum = sum+ n*n;

        }

printf(“sum = %d\n”, sum);

-----
```

The body of the loop

```
sum = sum + n*n;
```

Is executed 10 times for n=1,2,...10 each time incrementing **sum** by the square of the value of n.

One of the important points about the **for** loop is that all the three actions, namely initialization, testing, and incrementing. Are placed in the **for** statement itself, thus making them visible to the programmers and user, in one place. The for statement and its equivalent of while and do statements are shown in table 6.1.

Table 6.1 Comparison of the Three Loops

for	while	do
<pre> for (n=1;n<=10; ++n) { ----- ----- } </pre>	<pre> n = 1; while(n<=10) { ----- ----- n = n+1; } </pre>	<pre> n = 1; do { ----- ----- n =n +1; } while(n<=10); </pre>

Program 6.3 The program in Fig 6.4 uses a for loop to print the “power of 2” table for the power 0 to 20, both positive and negative.

Program

```

main ()
{
    long int p;
    double q;
    int n;
    printf(“-----\n”);
    printf(“2 to power n    n    2 to power -n\n”);
}
                    
```

```

printf("-----\n");

p = 1;

for (n= 0; n < 21; ++n) /* LOOP BEGINS */

{

if (n == 0)

P = 1;

else

p = p * 2;

q = 1.0/(double)p;

printf("%10d %10d %20.121f\n", p,n,q);

}

/* loop ends*?

printf("-----\n");

}

```

Output

```

-----

2 to power n      n      2 to power -n

-----

1                0      1.000000000000

2                1      0.500000000000

4                2      0.200000000000

```

8	3	0.250000000000
16	4	0.062500000000
32	5	0.031250000000
64	6	0.015625000000
128	7	0.007812500000
256	8	0.003906350000
512	9	0.001953125000
1024	10	0.000976562500
2048	11	0.000488281250
4096	12	0.000244140625
8192	13	0.000122070313
16394	14	0.000061035156
32768	15	0.000030517578
65536	16	0.000015258789
131072	17	0.000007629395
262144	18	0.000003814697
524288	19	0.000001907349
1048576	20	0.000000953674

Fig 6.4 program to print 'print power of 2' table using for loop

The program evaluates the value

$$p = 2^n$$

Successively by multiplying 2 by itself n times.

$$q = 2^{-n} = 1/p$$

Note that we have declared pa as long int and q as a double

Program 6.4 the program in fig 6.5 shows how to write a c program to print all the prime number between 1 and n, where 'n', where 'n' is the value supplied by the user.

Program

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

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

```
void main( )  
{  
    int prime (int num);  
    int n,i;  
    int temp;  
    printf("enter the value of n: ");  
    scanf("%d", &n);  
    printf("Prime Number Between 1 And %D Are : \n".n);  
    for(i=2; j<=n; i++)  
    {  
        temp=prime(i);
```



```
        if(temp==-99)

            continue;

    else

        printf("%d\t", i);

    }

    getch( );

}

int prime (int num)

{

    int j;

    for (j=2; j<num; j++)

    {

        if (num%j==0)

            return (-99);

        else

            ;

    }

    if (j==num)

        return(num);

}
```

Output

enter the Value Of N: 20

Prime Number Between 1 And 20 Are:

2 3 5 7 11 13 17 19

Fig.6.5 Program To Print All Prime Number Between 1 And n

Program 6.5 the program in Fig 6.6 shows how to write a c program to print the Fibonacci number

Program

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

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

```
void main ( )  
  
{  
  
int num 1=0, num2=1, n, i, fib;  
  
clrscr( );  
  
printf(“\n\nenter the value of n: “);  
  
scanf(“%d”, &n);  
  
for (i = 1; i <= n-2; i++)  
  
{  
  
fib=num1 + num2;  
  
num=num2;  
  
num2=fib;
```

```

    }

    printf("\nn fibonacci number (for n = %d) = %d, n, fib);

    getch();

}

```

Fig 6.6 program to print nth Fibonacci number

Additional features of for loop

The for loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the for statement. The statements

p = 1;

for (n =0; n<17; ++n)

can be written as

for (p=1, n=0; n<17; ++n)

Note that the initialization section has two parts p=1 and n=1 separated by a comma. Like the initialization section, the increment section may also have more than one part. For example, the loop

for (n =1, m=50; n<=m; n=n+1, m=m-1)

{

p = m/n;

printf("%d %d %d\n", n, m, p);

}

is perfectly valid. The multiple arguments in the increment section are separated by commas.

The third features is that the test-condition may have any compound relation and testing need not be limited only to loop control variable. Consider the example below:

```

sum = 0;

for(i =1; i < 20 && sum < 100; ++i)
{

sum = sum+i;

printf(“%d %d\n”, i, sum);

}

```

The loop uses a compound test condition with the counter variable I and sentinel variable sum. The loop is executed as long as both the conditions $i < 20$ and $sum < 100$ are true. The sum is evaluated inside the loop.

It is also permissible to use expressions in the assignment statements of initialization and increment sections. For example, a statement of the type

```

for ( x = (m+n)/2; x > 0; x = x/2)

```

is perfectly valid.

Another unique aspect of for loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```

-----

m = 5;

for( ; m != 100 ;)
{

printf(“%d\n”, m);

m= m+5;

}

-----

```

Both the initialization and increment sections are omitted in the **for** statements. The initialization has been done before the for statement and the control variable is incremented inside the loop. In such cases, the sections are left 'blank'. However, the semicolons separating the sections must remain. If the test-condition is not present, the for statement sets up an 'infinite' loop. Such loops can be broken using **break** or **goto** statements in the loop.

We can set up time delay loops using the null statements follow:

```
for (j= 1000; j > 0; j = j-1)  
  
;
```

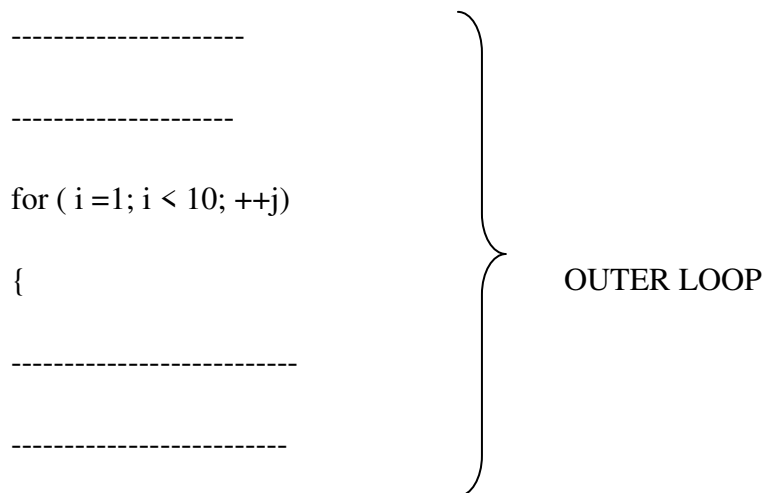
This loop is executed 1000times without producing any output; it simply causes a time delay. Notice that body of the loop contains only semicolon, known as null statement. This can also be written as

```
for(j=1000; j>0; j=j-1)
```

This implies that the C compiler will not give an error message if we place a semicolon by mistake at end of a for statement. The semicolon will be considered as a null statement and the program may produce some nonsense.

Nesting of for loops

Nesting of loops, that is, one for statement with another for statement is allowed in C. for example, two loops can be nested as follows:



```

for ( j =1; j != 5; ++j)  } INNER LOOP
{
-----
-----
}

-----
-----
}

-----
-----

```

The nesting may continue up to any desired level. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each for statement. (ANSI C allows up to 15 levels of nesting. However, some compilers permit more).

The program to print the multiplication table discussed in program 6.2 can be written more concisely using nested for statements as follows:

```

-----

for ( row = 1; row <= ROWMAX; ++row)
{
for ( column = 1; column <= COLMAX; ++column)
{
y = row * column;

```

```
printf(“%4d”, y);
```

```
}
```

```
printf(“\n”);
```

```
}
```

```
-----
```

Program 6.6 A Class of n students take an annual examination in m subjects. A program to read the marks obtained by each student in various subjects and to compute and print the total marks obtained by each of them is given in Fig.6.7.

The program uses two for loops, one for controlling the number of students and the other for controlling the number of subjects. Since both the number of students and the number of subjects are requested by the program, the program may be used for a class of any number of subjects.

The outer loop includes three parts:

1. Reading of roll-numbers of students, one after another,
2. Inner loop, where the marks are read and totalled for each student; and
3. Printing of total marks and declaration of grades.

Program

```
#define FIRST 360
```

```
#define SECOND 240
```

```
main( )
```

```
{
```

```
int n, m, i, j, roll_number, marks, total;
```

```
printf(“Enter number of students and subjects\n”);
```

```

scanf("%d %d", &n, &n);

printf("\n");

for (I =1; I <= n ; ++i)
{

    print("enter roll_sumber : ");

    scanf("%d", &roll_number);

    total = 0 ;

    printf("\nEnter marks of %d subjects for ROLL NO %d\n", m,
    roll_number);

for ( j = 1; j <= m; j++)

{

scanf("%d, &marks);

total = total + marks;

}

    printf("TOTAL MARKS = %d", total);

    if (total >= FIRST )

    printf("(First division )\n\n");

    else if (total >= SECOND )

    printf("(SECOND division )\n\n");

    else

    printf("(*** FAIL***)\n\n");

```



```
}  
  
}
```

Output

Enter number of students and subjects

3 6

Enter roll_number : 8701

Enter marks of 6 subjects for ROLL NO 8701

81 75 83 45 61 59

TOTAL MARKS =404 (First division)

Enter roll number: 8702

Enter marks of 6 subjects for ROLL NO 8702

57 49 55 47 65 41

TOTAL MARKS =308 (second division)

Enter roll number: 8703

Enter marks of 6 subjects for ROLL NO 8703

40 19 31 47 39 25

TOTAL MARKS =(***FAIL***)

Fig.6.7 illustration of nested for loops

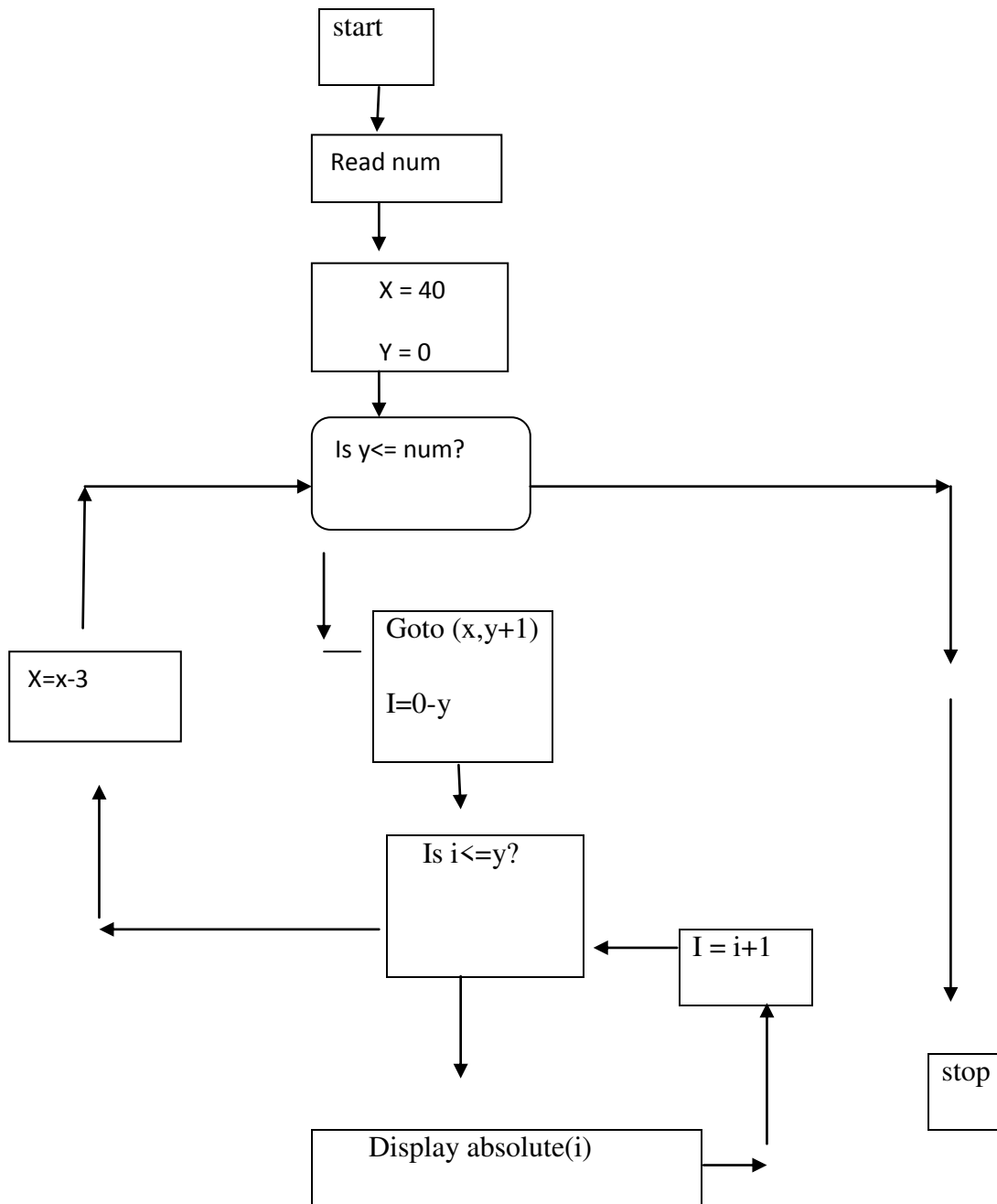
Program 6.7 the program in Fig6.8 show how to write a program to display a pyramid

Algorithm

Steps

- 1 - Start
- 2 - Read a value for generating the pyramid (num)
- 3 - Set $x = 40$
- 4 - Initialize the looping counter $y=0$
- 5 - Repeat steps 6-12 while $y \leq \text{num}$
- 6 - Move to the coordinate position $(x, y+1)$
- 7 - Initialize the loping counter $i=0-y$
- 8 - Repeat steps 9-10 while $i \leq y$
- 9 - Display the absolute value of I, $\text{ab}(i)$
- 10 - $-I = I + 1$
- 11 - $-x = x - 3$
- 12 - $-y = y + 1$
- 13 - -stop

Flow chart



Program

```
#include <stdio.h>

#include <conio.h>

void main ( )

{

int num,i,y,x=40;

clrscr ( );

printf("\nEnter a number for \n generating the pyramid:\n");

scanf("%d", &num);

for(y=0;y<=num;y++)

{

gotoxy(x,y+1);

for(i=0-y;i<=y;i++)

printf("%3d",abs(i));

x=x-3;

}

getch();

}
```

Output

Enter a number for generating the pyramid:

0

```

1 0 1
2 1 0 1 2
3 2 1 0 1 2 3
4 3 2 1 0 1 2 3 4
5 4 3 2 1 0 1 2 3 4 5
6 5 4 3 2 1 0 1 2 3 4 5 6
7 6 5 4 3 2 1 0 2 3 4 5 6 7

```

Fig 6.8 Program to build a pyramid

Selecting a loop

Given a problem, the programmer's first concern is to decide the type of loop structure to be used. To choose one of the three loop supported by C, we may use the following strategy:

- Analyze the problem and see whether it required a pre-test or post-test loop.
- If it requires a post-test loop, then we can use only one loop, do while
- If it requires a pre-test loop, then we can have two choices; for and while
- Decide whether the loop termination requires counter-based control or sentinel- based control;
- Use for loop if the counter-based control is necessary.
- Use while loop if the sentinel-based controls required
- Note that both the counter-controlled and sentinel-controlled loops can be implemented by all the three control structures.

6.5 JUMPS IN LOOPS

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain conditions occur. For example, consider the case of searching for a particular name in a list containing, say, 100 names. A

program loop written for reading and testing the names 100 times must be terminated as soon as the desired name is found. C permits a jump from one statement to another within a loop as well as jump out of a loop.

Jumping out of a loop

An early exit from a loop can be accomplished by using the `break` statement or the `goto` statement. We have already seen the use of the `break` in the `switch` statement and the `goto` in the `if....else` construct. These statements can also be used within `while`, `do` or `for` loops. They are illustrated in Fig.6.9 and Fig.6.10.

When a `break` statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the `break` would only exit from the loop containing it. That is, the `break` will exit only a single loop.

Since a `goto` statement can transfer the control to any place in a program. It is useful to provide branching within a loop. Another important use of `goto` is to exit from deeply nested loop when an error occurs. A simple `break` statement would not work here.

```
while(.....)                                do
{                                             {
.....
.....
.....
.....
if(condition)
```

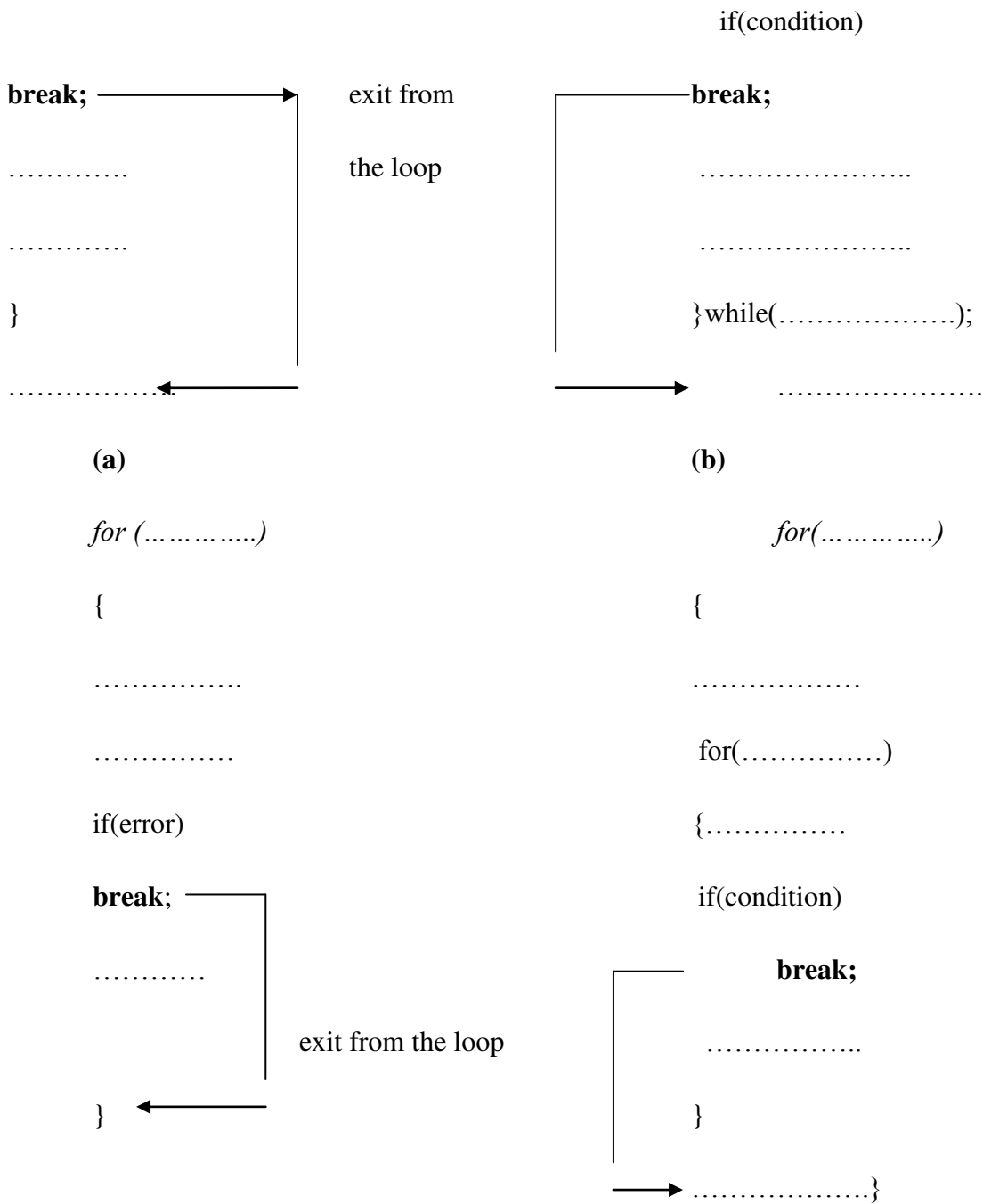


Fig.6.9 exiting a loop with break statement

```

while (.....)
{
if(error)

```

```

for(.....)
{ .....
for(.....)

```

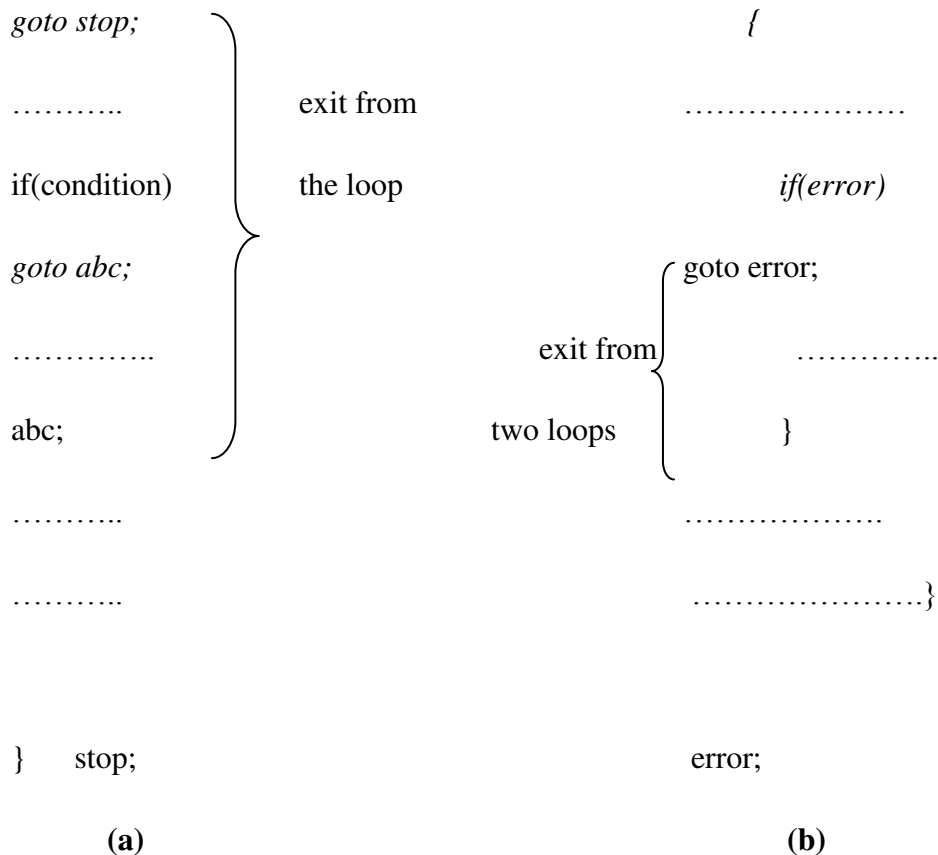


Fig.6.10 Jumping within and exiting from the loops with **goto** statement

Program 6.8 The program in Fig.6.11 illustrates the use of the break statement in a C program.

The program reads a list of positive values and calculates their average. The for loop is written to read 1000 values. However, if we want the program to calculate the average of any set of values less than 1000, then we must enter a ‘negative’ number after the last value in the list, to mark the end of input.

Program

```

main ( )
{
int m;

float x, sum, average;

```



```

printf("this program computes the average of a set of number \n");

printf("enter values one after another \n");

printf("enter a negative number at the end.\n\n");

sum = 0;

for (m = 1 ; m <= 1000 ; ++m)

{

scanf("%f", &x);

if ( x < 0 )

break;

sum += x ;

}

average = sum/(float)(m-1);

printf("\n");

printf("number of values = %d\n", m-1);

printf("sum          = %f\n, sum);

printf("average     = %f\n", average);

}

```

Output

this program computes the average of a set of numbers

enter values one after another

enter a negative number at the end.

21 23 24 22 -1

number of values =6

sum =138.000000

average =23.000000

Fig.6.11 use of break in a program

Each value, when it is read, it tested to see whether it is a positive number or not. If it is positive, the value is added to the sum; otherwise, the loop terminates. On exit, the average of the values read is calculated and results are printed out.

Program 6.9 A program to evaluate the series.

$1/1-x=1+x+x^2+x^3+x^4+\dots+x^n$ for $-1<x<1$ with 0.01 percent accuracy is given in Fig6.12. The goto statement is used to exit the loop on achieving the desired accuracy.

We have used the for statement to perform the repeated addition of each of the terms in the series. Since it is an infinite series, the evaluation of the function is terminated when the term x^n reaches the desired accuracy. The value of n that decides the number of loop operations is not known and therefore we have decided arbitrarily a value of 100, which may or may not result in the desired level of accuracy.

Program

```
#define LOOP 100
```

```
#define ACCURACY 0.0001
```

```
main ( )
```

```
{
```

```
int n;
```

```
float x, term, sum;
```

```

printf("input value of x: ");

scanf("%f", &x);

sum = 0;

for (term = 1, n = 1 ; n <= loop ; ++n)
    {
        sum += term;

        if (term <= ACCURACY)

            goto output;

        /*EXIT FROM THE LOOP*/

        term *= x ;
    }

printf("\n FINAL VALUE OF N IS NOT SUFFICIENT \N");

printf("TO ACHIEVER DEIRED ACCURACYN");

goto end;

Out put

printf("\n EXIT FROM LOOP \n");

printf("sum = %f; no.of term = %d\n", sum, n);

end:

/*Null statement*/

}

```

Output

Input value of x: .21

EXIT FROM LOOP

Sum = 1.265800; No.of term = 7

Input value of x : .75

EXIT FROM THE LOOP

Sum = 3.999774; No.of terms= 34

Input value of x : .99

FINAL VALUE OF N IS NOT SUFFICIENT

TO ACHIEVE DESIRED ACCURACY

Fig.6.12 Use of goto to exit from a loop

The test of accuracy is made using an **if** statement and the **goto** statement exists the loop as soon as the accuracy condition is satisfied. If the number of loop repetitions is not large enough to produce the desired accuracy. The program prints an appropriate message.

Note that the **break** statement is not very convenient to use here. Both the normal exit and the **break** exit will transfer the control to the same statement that appears next to the loop. But, in the present problem, normal exit prints the message

“FINAL VALUE OF N IS NOT SUFFICIENT

TO ACHIEVE DESIRED ACCURACY”

and the forced exit prints the results of evaluation. Notice the use of a null statement at the end. This is necessary because a program should not end with a label.

Structured Programming

Structured programming is an approach to the design and development of programs. It is a discipline of making a program's logic easy to understand by using only the basic three control structures:

- Sequence (straight line) structure
- Selection(branching) structure
- Repetition (looping) structure

While sequence and loop structures are sufficient to meet all the requirements of programming, the selection to be structure proves more convenient in some situations.

The use of structured programming techniques helps ensure well-designed programs that are easier to write, read, debug and maintain compared to those that are unstructured.

Structured programming discourages the implementation of unconditional branching using jump statements such as **goto**, **break** and **continue**. In its purest form, structured programming is synonymous with “goto less programming”.

Do not go to goto statement!

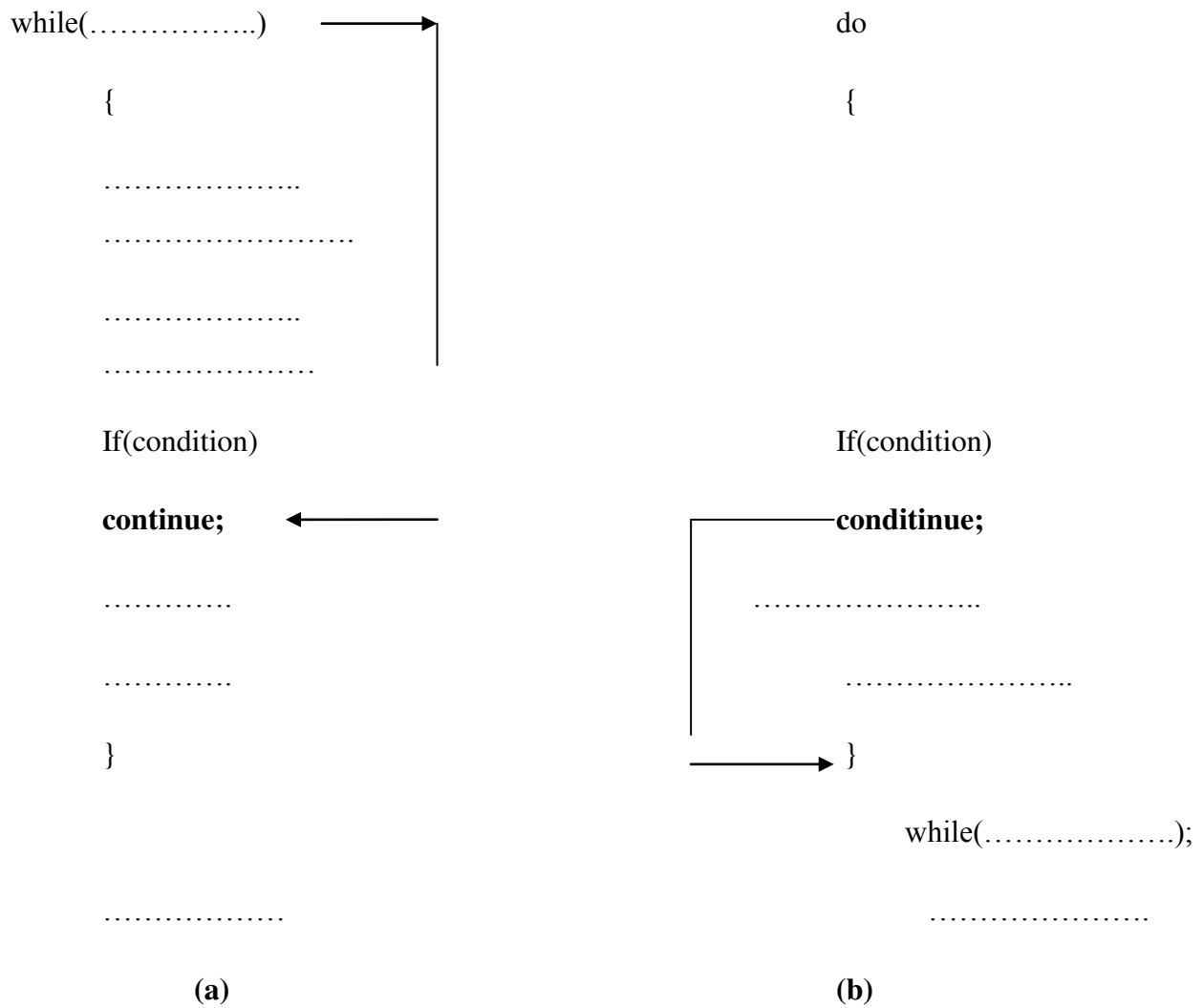
Skipping a part of a loop

During the loop operations, It may be necessary to skip part of the body the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applications belonging to a certain category. On reading the category code of an applicant, a test is made to be whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operations.

Like the **break** statement, C supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue**, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The **continue** statement tells the compiler, “SKIP THE FOLLOWING STATEMENTS

AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is simply **continue;**

The use of the **continue** statement in loops is illustrated in Fig.6.13. In while and do loops, continue causes the control to go directly to the test-condition and then to continue the iteration process. In the case of for loop, the increment section of the loop is executed before the test-condition is evaluated.



```

for (.....)
{
.....
.....
If(.....)
continue; ←
.....

```

Fig 6.13 Bypassing and continuing i loops

Program 6.10 The program in Fig.6.14 illustrates the use of **continue** statement

The program evaluates the square root of a series of numbers and prints the results. The process stops when the number 9999 is typed in.

In case, the series contains any negative numbers, the process of evaluation of square root should be bypassed for such numbers because the square root of a negative number is not defined. The continue statement is used to achieve this. The program also prints a message saying that the number is negative and keeps an account of negative numbers.

The final output includes the number of positive values evaluated and the number of negative items encountered.

Program

```

#include <math.h>

main ()
{
int count, negative;

double number, sqroot;

```

```

printf("Enter 9999 to STOP\n");

Count = 0;

negative = 0;

while (count <= 100)
    {
        printf("enter a number :");

        scanf("%1f", &number);

        if (number == 9999)

            break; /* exit from the loop*/

        if(number < 0)

            {

                printf("Number is negative\n\n");

                negative++;

                continue; /*SIP REST OF THE LOOP*/

            }

        Sqroot = sqrt(number);

        printf("number = %1f\n\n", number, sqroot);

        count++;

    }

printf("number of the items done = %d\n", count);

printf("\n\n negative items = %d\n, negative);

```



```
printf("END OF DATA\n");  
}
```

Output

ENTER 9999 to STOP

Enter a number: 25.0

Number = 25.000000

Squareroot = 5.000000

Enter a number: 40.5

Number =40.500000

Square root =6.363961

Enter a number: -9

Number is negative

Enter a number:16

Number =16.000000

Square root =4.000000

Enter a number:-14.75

Number is negative

Enter a number: 80

Number =80.000000

Square root =8.944272

Enter a number: 9999

Number of items done = 4

Negative items =2

END OF DATA

Fig.6.14 use of continue statement

Avoiding got

As mentioned earlier, it is a good practice to avoid using **goto**. There are many reasons for this. When **goto** is used, may compilers generate a less efficient code. In addition, using many of them makes a program logic complicated and renders the program unreadable. It is possible to avoid using **goto** by careful program design. In case any **goto** is absolutely necessary, it should be documented. The **goto** jumps shown in Fig 6.15 would cause problems and therefore must be avoided.

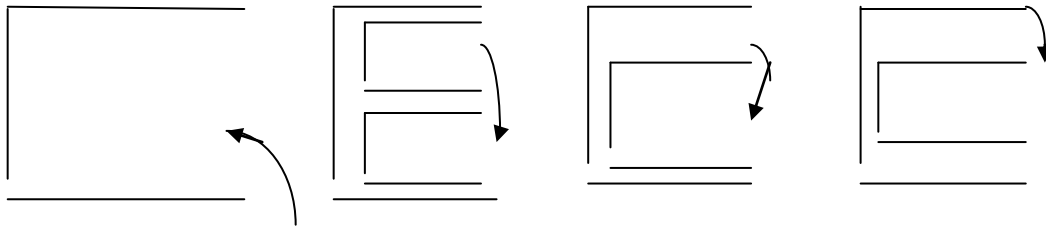


Fig. 6.15 goto jumps to be avoided

JUMPING OUT OF THE PROGRAM

We have just seen that we can jump out of a loop using either the **break** statement or **goto** statement. in a similar way, we can jump out of a program by using the library function `exit ()`. In case, due to some reason, we wish to break out of a program and return to the operating system, we can use the `exit()` function, as shown below.

```
.....  
.....  
if(test-condition)  
  
exit (0);
```

.....
.....

The exit () function takes an integer value as its argument. Normally zero is used to indicate termination and a nonzero value to indicate termination due to some error or abnormal condition. The use of exit() function requires the inclusion of the header file <stdio.h>.

6.6 CONCISE TEST EXPRESSIONS

We often use test expressions in the if, for while and do statements that are evaluated and compared with zero for making branching decisions. Since every integer expression has a true/false value, we need not make explicit comparisons with zero. For instance, the expression x is true whenever x is not zero, and false when x is zero. Applying ! Operator, we can write concise test expressions without using any relational operators.

if(expressions ==0)

is equivalent to

if(expressions)

similarly,

if(expression! =0)

is equivalent to

if (expression)

for example,

If(m%5==0 && n%5==0) is same as if(!(m%5)&&!(n%5))

Case studies

1. Histogram

Problem: In an organization, the employees are grouped according to their basic pay for the purpose of certain perks. The pay-range and the number of employees in each group are as follows:

	Group pay6-aranage	Number of employees
1.	750-1500	12
2.	2501-3000	23
3.	3001-4500	35
4.	4501-6000	20
5.	above 6000	11

Draw a histogram to highlight the group sizes.

Problem analysis:

Given the size of groups, it is required to draw bars representing the sizes of various groups. For each bar, its group number and size are to be written.

Program in Fig. 6.17 reads the number of employees belonging to each group and draws a histogram. The program uses four for loops and two if.....else statements.

Program:

```
#define N 5

main()
{
int value[n];
int i, j, n, x;
for(n=0; n < n; ++n)
```

```

{
printf("enter employees in group - %d :", n+1);

scanf("%d", &x);

value[n] = x;

        printf("%d\n", value[n]);

}

printf("\n");

printf("\n");

for (n = 0; n < n; ++n)

{

        for(i=0; i <= 3 ; i++)

        {

                if ( i ==2 )

                printf("group-%1d |", n+1);

                else

                printf("|");

                for (j=1; j <= value[n]; ++j)

                printf("*");

                if (i == 2)

                printf("(%d)\n", value[n]);

                else

```

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

Output

Enter employees in group – 1: 12

12

Enter employees in group – 2: 23

23

Enter employees in group –3: 35

35

Enter employees in group –4: 20

20

Enter employes in group –5: 11

11

```

|
Group – 1 |*****
          |*****(12)
          |*****
          |
```

Group – 2

|

|*****

|***** (23)

|*****

|

Group – 3

|

|*****

|***** (35)

|*****

|

Group – 4

|

|*****

|***** (20)

|*****

|

Group-5

|

|*****

|*****

|*****

2. Minimum cost

Problem: The cost of operation of a unit consists of a unit two components C1 and C2 which can be expressed as functions of a parameter p as follows:

$$C1 = 30-8p$$

$$C2 = 10+p^2$$

The parameter p ranges from 0 to 10. Determine the value of p with an accuracy of +0.1 where the cost of operation would be minimum.

Problem Analysis:

$$\text{Total cost} = C_1+C_2= 40-8p+p^2$$

The cost is 40 when p = 0, and 33 when p = 1 and 60 when p = 10. The cost, therefore, decreases first and then increases. The program in Fig.6.18 evaluates the cost at successive intervals of p (in steps of 0.1) and stops when the cost begins to increase. The program employs break and continue statements to exit the loop.

Program

```
main()
{
    float p, cost, p1, cost1;
    for (p=1; p <= 10; p = p + 0.1)
    {
        cost = 40 - 8 * p + * p;
        if(p == 0)
        {
            cost1 = cost;
```



```
continue;
    }

if(cost >= cost1)
break;

cost1 = cost;

p1 = p;
    }

p = (p +p1)/2.0;

cost = 40 - 8 * p + p*p;

printf("\n MINIMUM COST = %.2f AT p = %.1f\n", cost, p);
}
```

Output

MINIMUM COST = 24.00 A p = 4.0

Fig. 6.18 Program of minimum cost problem

7 ARRAYS

7.1 INTRODUCTION

So far we have use only the fundamental data types, namely **char**, **int**, **float** ,**double** and variations of **int** and **double**. Although these types are very useful, they are constrained by the fact that a variable of these type scan store only one value at any given time. Therefore, they can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as array that can be used for such applications.

An array is a fixed-size sequence collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept o fan array can be used:

1. List of temperatures recorded every hour in a day, or a month, or a year.
2. List of employees in an organization.
3. List of products and heir cost sold by a store.
4. Test scores of a class of students.
5. List of customers and their telephone numbers.
6. Table of daily rainfall data.

andso on.

Since an array provides a convenient structure for representing data, it is classified as one of the data structures in C. Other data structures include structures, lists, queues and trees. A complete discussion of all data structures is beyond the scope of this text. However, we shall consider structures in chapter 10 and lists chapter 13.

As we mentioned earlier, an array is sequence collection of relate data items that share a common name. For instance, we can use an array name salary to represent a set of salaries of a group of employees in an organization. We can refer to the individual salaries by writing a number called index or subscript in brackets after the array name. For example,

salary [10]

represents the salary of 10th employee. While the complete set of values is referred to as an array individual values are called elements.

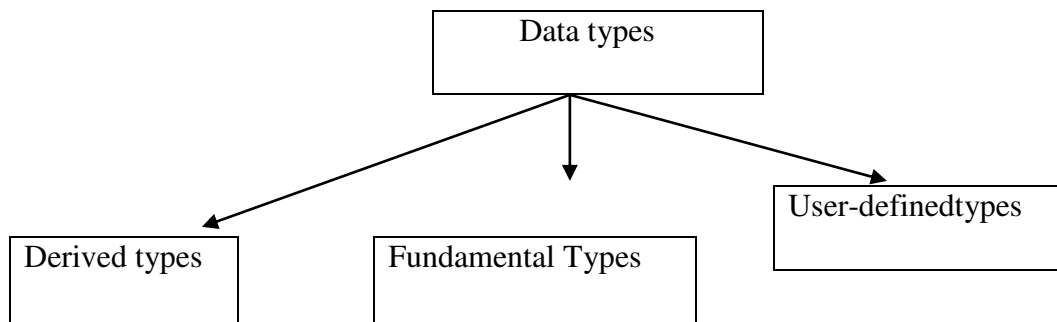
The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, perform calculations, and print out the results.

We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions. In this chapter, we introduce the concept of an array and discuss how to use it create and apply the following types of arrays.

- ▶ One-dimensional arrays.
- ▶ Two- dimensional arrays
- ▶ Multi dimensional arrays

Data structures

C supports a rich set of derived and user-defined data types in addition to a variety of fundamental types as shown below:



- ▶ Arrays
- ▶ Functions
- ▶ Pointers

- Integral types
- Float types
- Character types
- Structures
- Unions
- Enumerations

Arrays and structures are referred to as structured data types because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations. In programming parlance, such data types are known as data structures.

In addition to arrays and structures, C supports creation and manipulation of the following data structures:

- Linked lists
- Stacks
- Queues
- Trees

7.2 ONE-DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation.

$$A = \frac{\sum_{i=1}^n x_i}{n}$$

to calculate the average of n values of. The subscripted variable x_i refers to the i_{th} element of x. In C, single-subscripted variable x_i can be expressed as

x[1],x[2],x[3],...,x[n]

The subscript can begin with number 0. That is

X[0]

is allowed. For example, if we want to represent a set of five numbers, say(35,40,20,57,19) by an array variable number, then we may declare the variable number as follows

int number[5];

and the computer reserves five storage locations as shown below:

```
number[0]
number[1]
number[2]
number[3]
number[4]
```

The values to the array elements can be assigned as follows:

```
Number[0] = 35;
```

```
Number[1] = 40;
```

```
Number[2] = 20;
```

```
Number[3] = 57;
```

```
Number[4] = 19;
```

this would cause the array number to store the values as shown below:

35

40

20

57

```

number[0]
  number[1]
  number[2]
  number[3]
  number[4]

```

these elements may be used in programs just like any other C variable. For example, the following are valid statements:

```

a = number[0] + 10;

number[4] = number[0] + number[2];

number[2] = x[5] + y[10];

value[6] = number[i] * 3;

```

The subscripts of an array can be integer constants, integer variables like `i`, or expressions that yield integers. C performs no bounds checking and, therefore, care should be exercised to ensure that the array indices are within the declared limits.

7.3 DECLARATION OF ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

```

type variable-name[size];

```

The type specifies the type of element that will be contained in the array, such as **int**, **float**, or **char** and the size indicates the maximum number of elements that can be stored inside the array. For example

```

float height[50];

```

declares the height to be array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly

```

int group[10];

```

declares the group as an array to contain a maximum of 10 integer constants. Remember:

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a numeric constant or a symbolic constant.

The C language treats character strings simply as arrays of characters. The size in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

declares the name as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant **int** the string variable name.

“WELL DONE”

Each character of the string is treated as an element of the array name and is stored in the memory as follows:

‘W’

‘E’

‘L’

‘L’

‘ ’

‘D’

‘O’

‘N’

‘E’

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element name [10] holds the null character '\0'. When declaring character arrays, we must allow one extra element space for the null terminator.

Program 7.1 write a program using a single-subscripted variable to evaluate the following expressions:

$$\text{Total} = \sum_{i=1}^{10} x_i^2$$

The values of x_1, x_2, \dots are read from the terminal.

Program in Fig.7.1 uses a one-dimensional array x to read the values and compute the sum of their squares.

Program

```
main( )
{
int i;

float x[10], value, total ;

/*.....READING VALUES INTO ARRAY.....*/

printf("ENTER 10 REALNUMBERS\n");

for(i = 0; i<10; i++)
{

scanf("%f", &value);

x[i] = value;

}

/*.....COMPUTATION OF TOTAL.....*/
```



```

total = 0.0;

for(i=0 ; i<10; i++)

    total = total + x[i] *x[i];

/* . . . . .PRINTING OF x[I] VALUES AND TOTAL. . . */

printf("\n");

for(i=0; i<10 i++)

    printf("x[%2d] = %5.2f\n", i+1, x[i]);

    printf("\ntotal= %.2\n", total);

}

```

Output

```

ENTER 10 REAL NUMBER

1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.10

x[1] = 1.10

x[2] = 2.20

x[3] = 3.30

x[4] = 4.40

x[5] = 5.50

x[6] = 6.60

x[7] = 7.70

x[8] = 8.80

```

x[9] = 9.90

x[10] = 10.10

Total = 446.86

Fig .7.1 Program to illustrate one-dimensional array

Note C99 permits arrays whose size can be specified at run time. See appendix “C99” Features

7.4 INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain “garbage”. An array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

type array-name [size] = (list of values);

The values in the list are separated by commas. For example, the statement

int number[3] = {0,0,0};

Will declare the variable number as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

float total[5] = {0,0,15,75,-10};

will initialize the first three elements to 0.0, 15 75 and -10.0 and the remaining two elements to zero.

The size may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

```
int counter[] {1,1,1,1};
```

will declare the counter array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus, the statement

```
char name[] = {'j', 'o', 'h', 'n', '\0'};
```

declares the name to be an array of five character, initialized with the string “john” ending with the null character alternatively, we can assign the string literal directly as under.

```
char name [ ] = “john”;
```

(Character arrays and strings are discussed in detail in Chapter 8)

Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases, the remaining elements are initialized to zero, if the array type is numeric and NULL if the type is char. For example,

```
int number [5] = {10,20};
```

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0. Similarly, the declaration.

```
char city [5] = {'B'};
```

will initialize the first element to ‘B’ and the remaining four to NULL. It is a good idea, however, to declare the size explicitly, as it allows the compiler to do some error checking.

Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement

```
int number [3] = {10, 20, 30, 40}
```

will not work. It is illegal in C.

Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
-----  
-----  
for (i=0; i < 100 I = i+1)  
{  
    if i < 50  
else  
    sum [i] = 1.0;  
}  
-----  
-----
```

The first 50 elements of the array sum are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.

We can also use a read function such as **scanf** to initialize an array. For example, the statements

```
int x [3];  
scanf_”%d%d%d”, &x[0], &[1], &x[2]);
```

will initialize array elements with the values entered through the keyboard.

Program 7.2

Given below is the list of marks obtained by a class of 50 students in an annual examination.

43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74 81 49 37

40 49 16 75 87 91 33 2 58 78 65 56 76 67 45 54 36 73 12 21

73 49 51 19 39 49 68 93 85 59

Write a program coded in fig 7.2 uses the array group containing 11 elements, one for each range of marks. Each element counts those values falling within the range of values it represents.

For any value, we can determine the correct group element by dividing the value by 10. For example, consider the value 59. The integer division of 59 by 10 yields 5. This is the element into which 59 is counted.

Program

```
#define MAXVAL 50
#define COUNTER 11
main ()
{
    float  value[MAXVAL];
    int    I, low, high;
    int    group [COUNTER] = {0,0,0,0,0,0,0,0,0,0,0}
    /* . . . . . READING AND COUNTING . . . . . */
    for (i = 0; i < MAXVAL ; i++)
    {
        /* . . . . . READING OF VALUES . . . . . */
        Scanf("%f", &value [i]);
        /* . . . . . COUNTING FREQUENCY OF GROUPS . . . . . */
        ++ group [(int) (value[i])/10];
    }
    /* . . . PRINTING OF FREQUENCY TABLE . . . . . */
    printf("\n");
    printf("GROUP RANGE FREQUENCY\n\n");
    for ( i= 0; i < COUNTER; i++)
    {
        low = i * 10;
```

```

        if(i == 10)
            high = 100;
    else
        high = low +9;
    printf("%2d %3d to %3d %d\n",
        i+1, low, high, group[i]);
    }
}

```

Output

```

43 65 51 27 79 11 56 61 82 09 25 36 07 49 55 63 74
81 49 37 40 49 16 75 87 91 33 24 58 78 65 56 76 67
45 54 36 63 12 21 73 49 51 19 39 49 68 93 85 59

```

GROUP	RANGE	FREQUENCY
1	0 to 9	2
2	10 to 19	4
3	20 to 29	4
4	30 to 39	5
5	40 to 49	8
6	50 to 59	8
7	60 to 69	7
8	70 to 79	6
9	80 to 89	4
10	90 to 99	2
11	100 to 100	0

Fig 7.2 Program for frequency counting

Note that we have used an initialization statement.

```
int group [COUNTER] = {0,0,0,0,0,0,0,0,0,0,0};
```

which can be replaced by

```
int group [COUNTER] = {0};
```

This will initialize all the elements to zero.

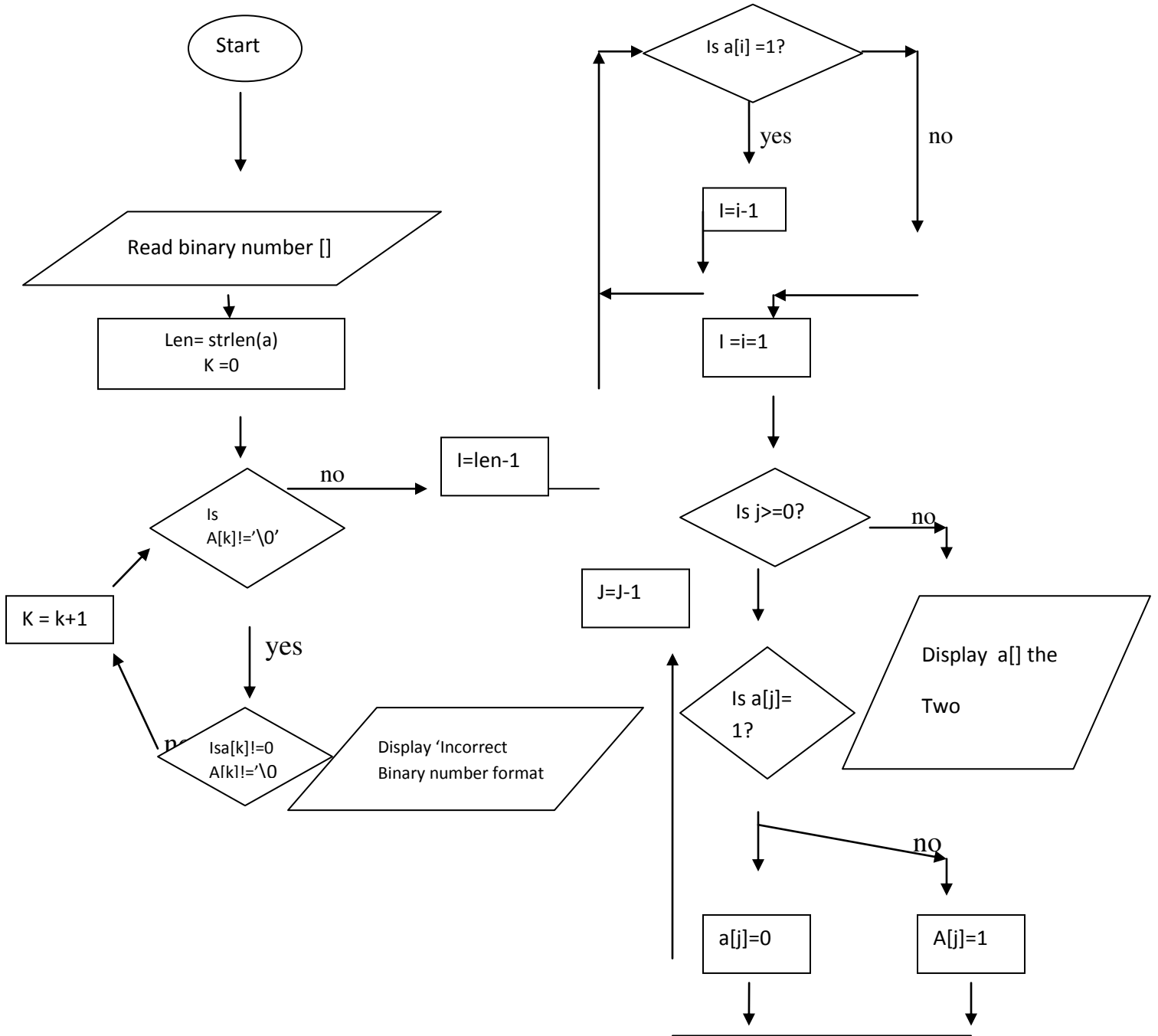
Program 7.3

The program shown in Fig.7.3 shows the algorithm, flowchart and the complete C program to find the two's compliment of a binary number.

Algorithm

- Step 1 - Start
- Step 2 - Read a binary number string (a[])
- Step 3 - Calculate the length of string str (len)
- Step 4 - Initialize the looping counter k=0
- Step 5- Repeat Steps 6-8 while a[k] != '\0'
- Step 6- If a[k]!=0 AND a[k]! goto step 7 else goto step 8
- Step 7- Display error 'Incorrect binary number format' and terminate the program
- Step 8- k = k +1
- Step 9- Initialize the looping counter I = len -1
- Step 10- Repeat step 11 while a[j]! = '1'
- Step 11- i = i+1
- Step12- Initialize the looping counter j = i-1
- Step 13- Repeat step 14-17 while j>=0
- Step14- If a[j] =1 goto step 15 else goto step 16
- Step15- a[j] = '0'
- Step16- a[j] = '1'
- Step17- j= j+1
- Step18- Display a[] as the two's compliment
- Step19- Stop

Flow chart



Program

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main ()
{
    char a[16];
    int i,j,k, len;
    clrscr();
    printf("Enter a binary number:");
    gets(a);
    len=strlen(a);
    for(k=0; a[k]!='\0': k++)
    {
        if(a[k]!='\0'; && a[k]!='1')
        {
            printf("\nIncorrect binary number format ..... the program will quit");
            getch();
            exit(0);
        }
    }
    for(i=len-1; a[i]!='1'; i--)
    ;
    for(j=i-1; j>=0; j--)
    {
        if(a[j]=='1')
        a[j]='0';
        else
        a[j]='1';
    }
    printf("\n2's compliment = %s",a);
```

```
    getch();  
}
```

Output

```
Enter a binary number: 010111001001  
2's compliment = 10100110111
```

Fig 7.3 Algorithm, flowchart and C program to find two's compliment of a binary number

Searching and Sorting

Searching and sorting are the two most frequent operations performed on arrays. Computer Scientists have devised several data structures and searching and sorting techniques that facilitate rapid access to data stored in lists.

Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an ordered list. Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available.

The three simple and most important among them are

- Bubble set
- Selection sort
- Insertion sort

Other sorting techniques include shell sort, merge sort and quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the search key. If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are;

- Sequential search

- Binary search

A detailed discussion on these techniques is beyond the scope of this text. Consult any good book on data structures and algorithms.

7.5 TWO-DIMENSIONAL ARRAYS

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	Item 1	Item2	Item3
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four rows and three columns. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

In mathematics, we represent a particular value in a matrix by using two subscripts such as v_{ij} . here v denotes the entire matrix and v_{ij} refers to the value in the i the row and j th column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be define C as

$$v [4][3]$$

Two-dimensional arrays are declared as follows:

type array_name [row_size][column_size];

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

Two-dimensional arrays are stored in memory, as shown in fig 7.4. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

	Column 0	Column 1	Column 2
Row 1	310	275	365
Row 2	10	190	325
Row 3	310	275	365

Program 7.4 Write a program using a two-dimensional array to compute and print the following information from the table of data discussed above:

The program and its output are shown in fig .7.5. The program uses the variable value in two-dimensions with the index I representing girls and j representing items. The following equations are used in computing the results:

$$(a) \text{ Total sales by } m^{\text{th}} \text{ girl} = \sum_{j=0}^2 \text{value}[m][j] (\text{girl_total})$$

$$(b) \text{ total value of } n^{\text{th}} \text{ term} = \sum_{i=0}^3 \text{value}[i][n] (\text{item_total}[n])$$

$$(c) \text{ grand total} = \sum_{i=0}^3 \sum_{j=0}^2 \text{value}[i][j]$$

$$= \sum_{i=0}^3 \text{girl_total}[i]$$

$$= \sum_{i=0}^2 \text{item_total}[j]$$

Program

```
#define MAXGIRLS 4
```

```
#define MAXITEMS 3
```

```

main ( )

{

    int    value[MAXGIRLS][MAXITEMS];

    int    girl_total[MAXGIRLS], item_total[MAXITEMS];

    int    i, j, grand_total;

/*.....READING OF VALUES AND COMPUTING girl_total.....*/

    printf("Input data\n");

    printf("Enter values, one at a time, row-wise\n\n")

    for ( i =0; i < MAXGIRLS ;i++)

    {

        girls total[i] = 0;

        for (j = 0; j < MAXITEMS ; j++)

        {

            scanf("%d", &value[i] [j]);

            girl_total[i]= girl_total[i] + value[i] [j];

        }

    }

/*.....COMPUTING item_total.....*/

    for ( j=0; j < MAXITEMS ; j++)

    {

        item _total[j] = 0;

```

```

for (i =0; I < MAXGIRLS; i++)

        item_total[j] = item_total[j] + value[i][j];

}

/*.....COMPUTING grand_total.....*/

grand_total = 0;

for (i=0; I < MAXGIRLS; i++)

grand total = grand_total + girl_total [i]

/*.....PRINTING OF RESULTS.....*/

printf("\n GIRLS TOTALS \n\n");

for( i = 0; I < MAXGIRLS; i++)

printf("Salesgirl[%d] = %d\n", i+1, girl_total[i]);

printf("\n ITEM TOTALS\n\n");

for (j = 0; j<MAXITEMS; j++)

printf("Item[%d] = %d\n", j+1, item_total[j] );

printf("\nGrand total = %d\n", grand_total);

}

```

Output

Input data

Enter values, one at a time, row_wise

310 257 365

405 235 240

260 300 380

GIRLS TOTAL

Salesgirl [1] = 950

Salesgirl [2] = 725

Salesgirl [3] = 880

Salesgirl [4] = 940

ITEM TOTALS

Item [1] = 1185

Item [2] = 1000

Item [3] = 1310

Grand total = 3495

Fig 7.5 Illustration of two-dimensional arrays

Program 7.5 Write a program to compute and print a multiplication table for numbers 1 to 5 as shown below.

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	-	-	
4	4	8	-	-	
5	5	10	-	-	25

The program shown in Fig.7.6 uses a two-dimensional array to store the table values. Each value is calculated using the control variables of the nested for loops as follows:

$$\text{Product [i] [j] = row * column}$$

Where i denotes rows and j denotes columns of the product table. Since the indicates i and j range from 0 to 4, we have introduced the following transformation:

$$\text{row} = \text{j} + 1$$

$$\text{column} = \text{j} + 1$$

Program

```
#define ROWS      5

#define COLUMN    5

main ()
{

    int row, column, product[ROW][COLUMN];

    int i,j;

    printf("MULTIPLICATION TABLE\n\n");

    printf("  ");

    for(j=1; j <= COLUMNS ; j++)

        printf("%4d, j);

    printf("\n");

    printf(".....\n");

    for(i = 0; I < ROWS; i++)

    {
```



```

row = i + 1;

printf("%2d |", row);

for(j=1; j <= COLUMNS ; j++)

{

    column = j;

    product[i] [j] = row * column;

    printf"%4d, product[i][j]);

}

printf("\n");

}

}

```

Output

MULTIPLICATION TABLE

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Fig 7.6 Program to print multiplication table using two-dimensional array

7.6 INITIALIZING TWO- DIMENSIONAL ARRAYS

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example

```
int table[2] [3] = { 0,0,0,1,1,1};
```

initializes the elements of the first row zero and the second row to one. The initialization is done row by row. The above element can be equivalently written as

```
int table [2] [3] = {0,0,0}, {1,1,1};
```

by surrounding the elements of the each row by braces.

We can also initialize a two –dimensional array in the form of a matrix as shown below:

```
int table[2] [3] = {  
    {0,0,0},  
    {1,1,1}  
};
```

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension. That is, the statement

```
int table [ ] [3] = {  
    {0,0,0},  
    {1,1,1}  
};
```

is permitted.

If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

```
int table [2] [3] = {  
  
    {1,1}  
  
    {2};
```

Will initialize the first two elements of the first two to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int [3] [5] = { {0}, {0}, {0}};
```

The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero. The following statement will also achieve the same result.

```
int m [3] [5] = {0,0};
```

Program 7.6 A survey to know the popularity of four cars (Ambassador, fiat, Dolphin and Maruti) was conducted in four cities (Bombay, Calcutta, Delhi and Madras). Each person surveyed was asked to give his city and the type of car he was using. The results, in coded form, are tabulated as follows:

M	1	C	2	B	1	D	3	M	2	B	4
C	1	D	3	M	4	B	2	D	1	C	3
D	4	D	4	M	1	M	1	B	3	B	3
C	1	C	1	C	2	M	4	M	4	C	2
D	1	C	2	B	3	M	1	B	1	C	2
D	3	M	4	C	1	D	2	M	3	B	4

Codes represent the following information:

M	- Madras	1	-Ambassador
D	- Delhi	2-	- Flat
C	- Calcutta	3	- Dolphin
B	- Bombay	4	- Maruti

Write a program to produce a table showing popularity of various cars in our cities.

A two-dimensional array frequency is used as an accumulator to store the number of cars used. Under various categories in each city. For example, the element frequency denotes the number of cars of type j used in city i. the frequency is declared as an array of size 5*5 and all the elements are initialized to zero.

The program shown in /fing.7.7 reads the city code and the car code, one set after another, from the terminal. Tabulation ends when the letter X is read in place of a city code.

Program

```
main( )
{
int i, j, car;
int frequency[5] [5] = { {0}, {0}, {0}, {0}, {0} };
char city;
printf("For each person, enter the city code \n");
printf("followed by the car code.\n");
printf("Enter the letter X to indicate end.\n")
/* .....TABULATION BEGINS.....*/
```

```

    for(i = 1; i < 100; i++)
    {
        scanf("%sc", &city);

        if(city == 'X')

            break;

        scanf("%d", &car);

        switch(city)
        {

            case 'B'      : frequency[1] [car]++;

                           break;

            case 'C'      : frequency[2][car]++;

                           break;

            case 'D'      : frequency[3] [car]++;

                           break;

            case 'M'      : frequency[4] [car]++;

                           break;

        }

    }

    /*....TABULATION COMPLETED AND PRTING BEGINS.....*/

    printf("\n\n");

    printf("POPLARITY TABLE\n");

```

```

printf(".....\n");

printf("City Ambassador Fiat Dolphin Maruti \n");

printf(".....\n");

for(i=1; i <=4; i++)

{

    switch(i)

    {

        case 1    : printf("Bombay");

                    break;

        case2    :printf("Calcutta");

                    break;

        case3    :printf("Delhi");

                    break;

        case4    :printf("Madras");

                    break;

    }

for( j =1; j<=4; j++)

    printf("%7d", frequency[i][j]);

    printf("\n");

}

printf (".....\n");

```

```

/*.....PRINTING ENDS.....*/

}

```

Output

For each person, enter the city code

Followed by the car code.

Enter the letter X to indicate end.

```

M   1   C   2   B   1   D   3   M   2   B   4
C   1   D   3   M   4   B   2   D   1   C   3
D   4   D   4   M   1   M   1   B   3   B   3
C   1   C   1   C   2   M   4   M   4   C   2
D   1   C   2   B   3   M   1   B   1   C   2
D   3   M   4   C   1   D   2   M   3   B   4

```

POPULARITY TABLE

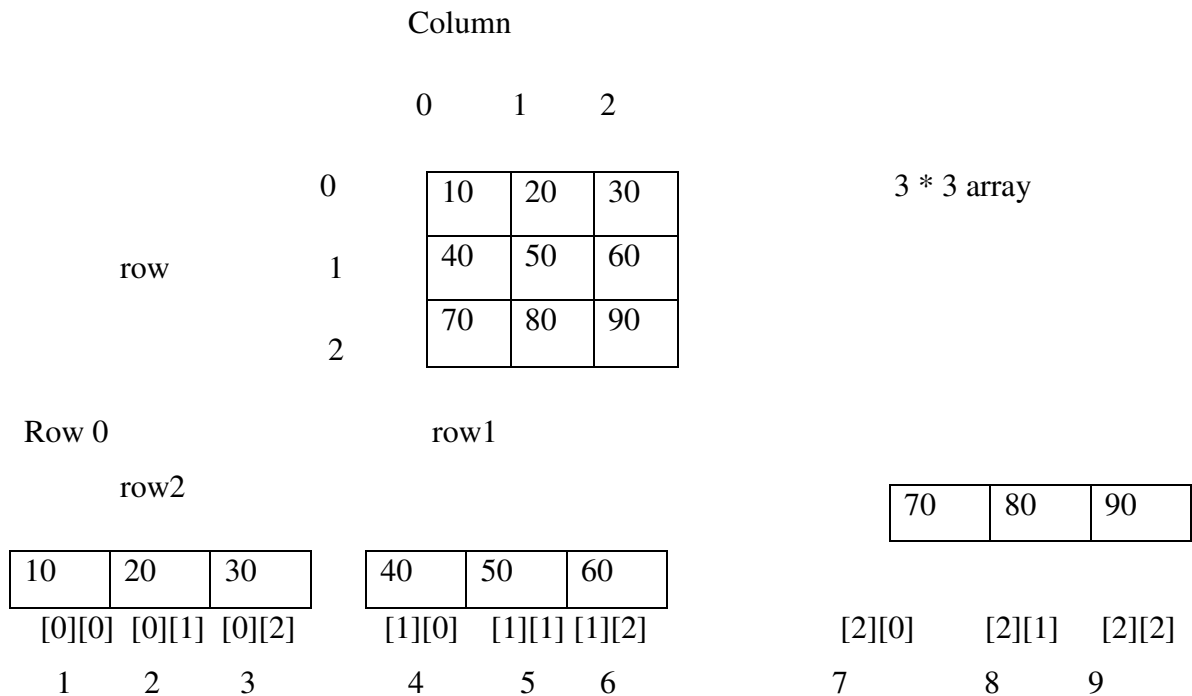
City	Ambassador	Fiat	Dolphin	Maruti
Bombay	2	1	3	2
Calcutta	4	5	2	0
Delhi	2	1	3	2
Madras	4	1	1	4

Fig 7.7 Program to tabulate a survey data

Memory Layout

The subscripts in the definition of a two-dimensional array represent rows and columns. This format maps the way that data elements are laid out in the memory. The elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we

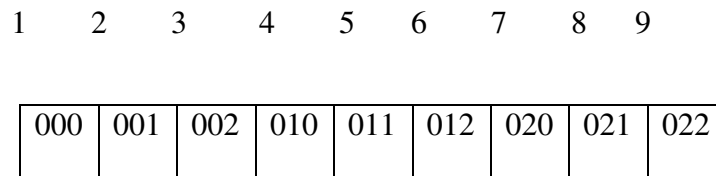
consider the memory as a row of bytes, with the lowest address on the left and the highest address on the right, a simple array will be stored in memory with the first element at the left end and the last element at the right end. Similarly, a two-dimensional array is stored “row-wise, starting from the first row and ending with the last row, treating each row like a simple array. This illustrated below.



Memory Layout

For a multi-dimensional array, the order of storage is that the first element stored has 0 in all its subscripts, the second has all of its subscripts 0 except the far right which has a value of 1 and so on.

The elements of a 2 x 3 array will be stored as under



10 11 12 13 14 15 16 17 18

100	101	102	110	111	112	120	121	122
-----	-----	-----	-----	-----	-----	-----	-----	-----

The far right subscript increments first and the other subscript increment in order from right to left. The sequence numbers 1,2,.....18 represents the location of that element in the list.

Program 7.7 The program in fig.7.8 shows how to multiply the elements of two N*N matrices.

Program

```

#include<stdio.h>

#include<conio.h>

void main ()

{

int  a1[10][10], a2[10][10], c[10][10], I, j, k, a, b;

clrscr ();

printf("Enter the size of the square matrix\n")

scanf("%d", &a);

b = a;

printf("You have to enter the matrix elements in row-wise fashion\n");

for(I = 0; i<a;i++)

{

for(j=0; j<b; j++)

{

```

```

printf("\nEnter the next element in the 1st matrix=");

scanf("%d", &a1[i][j]);

}

}

for(i=0; i<a; i++)

{

for(j=0; j < b; j++)

{

printf("\nEnter the next element in the 2nd matrix=");

scanf("%d", &a2[i][j]);

}

}

printf("\nEnter matrices are\n");

for(i=0;i<a; i++)

{    printf("\n");

for(j = 0; j<b; j++)

printf("%d", a1[i] [j]);

}

printf("\n");

for( i = 0; i<a; i++)

{    pintf("\n");

```

```

for(j=0; j<b; j++)

printf(“ %d”, a2[i][j]);

}

printf(“\n\nproduct of the two matrices is\n”);

for(i=0; i<a; i++)

    for(j=0; j<b; j++)

        {

            c[i] [j] =0;

            for(k=0 k<a; k++)

                c[i] [j] = c[i] [j]+a1[i] [k]*a2[k] [j];

        }

    for(i=0; i<a i++)

        {

            printf(“\n”);

            for(j=0; j<b ;j++)

                printf(“%d”, c[i] [j]);

        }

        getch ();

    }

```

Output

Enter the size of the square matrix

2

You have to enter the matrix element in row-wise fashion

Enter the next element in the 1st matrix = 1

Enter the next element in the 1st matrix = 0

Enter the next element in the 1st matrix = 2

Enter the next element in the 1st matrix = 3

Enter the next element in the 2nd matrix = 4

Enter the next element in the 2nd matrix = 5

Enter the next element in the 2nd matrix = 0

Enter the next element in the 2nd matrix = 2

Entered matrices are

1 0

2 3

4 5

0 2

Product of the two matrices is

4 5

8 16

Fig 7.8 Program for N*N matrix multiplication

Program 7.8 The program in fig.7.9. Shows how to find the transpose of a matrix

Algorithm

Step 1 - start

- Step2 - read a 3 x 3 matrix (a[3][3])
- Step3 - initialize the looping counter i = 0
- Step4 - Repeat steps 5-9 while i<3
- Step5 - Initialize the looping counter j = 0
- step6 - Repeat steps 7-8 while j<3
- step7 - b[i] [j]= a[j] [i]
- step8 - j =j +1
- step9 - i = i +1
- step10 - Display b[] [] as the transpose of the matrix a[] []
- step11 - stop

Program

```
#include<stdio.h>

#include<conio.h>

void main ()

{

int i, j, a[3] [3], b[3] [3];

clrscr();

printf("Enter a 3 X3, b[3] [3];

for(i=0; i<3; i++)

{

for (j=0; j<3 j++)
```

```

        {
            printf("a[ %d][%d] = ",i,j);
            scanf("%d. &a[i][j]);
        }
    }

printf("\nThe entered matrix is :\n")

for(i =0; i<3; i++)
{
    printf("\n");
    for(j=0; j<3; j++)
    {
        printf("%d\t", a[i][j]);
    }

for(i=0;i<3;i++)
{
    for(j=0; j<3; j++)

        b[i] [j] = a[j][i]
}

printf("\n\nthe transpose of the matrix is :\n")

for(i=0; i<3; i++)
{

```

```
printf("\n");  
  
for(j=0; j<3; j++)  
{  
  
printf("%d\t", b[j][i]);  
  
}  
  
}  
  
getch ();  
  
}
```

Output

Enter a 3 X 3 matrix:

```
a[0] [0]    =    1  
a[0] [1]    =    2  
a[0] [2]    =    3  
a[1] [0]    =    4  
a[1] [1]    =    5  
a[1] [2]    =    6  
a[2] [0]    =    7  
a[2] [1]    =    8  
a[2] [2]    =    9
```

The entered matrix is:

```
1    2    3
```

```

4    5    6
7    8    9

```

The transpose of the matrix is:

```

1    4    7
2    5    8
3    6    9

```

Fig 7.9 Program to find transpose of a matrix

7.7 MULTI-DIMENSIONAL ARRAYS

C allows arrays of three or more dimensions. The exact limit is determinate by the compiler. The general form of a multi-dimensional array is
type array_name [s1][s2][s3].....[sm];

where s is the size of the ith dimension. some example are:

```
int survey[3][5][2];
```

```
float table[5][4][3][3];
```

Survey is a three-dimensional array declared to contain 180 integer type elements. Similarly table is a four-dimensional array containing 300 elements of floating-point type.

The array survey may represent a survey data of rainfall during the last three years from January to December in five cities.

If the first index denotes year, the second city and the third month, then the element survey [2][3][10] denotes the rainfall in the month of October during the second year in city-3.

Remember that a three-dimensional array can be represented as a series of two-dimensional arrays as shown below:

Year 1

Month city	1	2	12
1				
2				
5				

Year 2

Month city	1	2	12
1				
2				
5				

ANSI C does not specify any limit for array dimension. However, most compilers permit seven to ten dimensions. Some allow even more.

7.8 DYNAMIC ARRAYS

So far, we created arrays at compile time. An array created at compile time by specifying size in the source code has fixed size and cannot be modified at run time. The process of allocating memory at compile time is known as static memory allocation and the arrays that receive static memory allocation are called static arrays. This approach works fine as long as we know exactly what our data requirements are.

Consider a situation where we want to use an array that can vary greatly in size. We must guess what will be the largest size ever needed and created the array accordingly. A difficult task in fact! Modern languages like C do not have this limitation. In C it is possible to allocate memory to arrays at run time. This feature is known as dynamic memory allocation and the arrays created at run time are called dynamic arrays, this effectively postpones the array definition to run time.

Dynamic arrays are created using what are known as pointer variables and memory management functions **malloc**, **calloc** and **realloc**. These functions are included in the header file `<stdlib.h>`. The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues. We discuss in detail pointers and pointer variables in Chapter 11 and creating and managing linked lists in Chapter 13.

7.9 MORE ABOUT ARRAYS

What we have discussed in this chapter are the basic concepts of arrays and their applications to a limited extent. There are some more important aspects of application of arrays they include:

- Using pointers for accessing arrays;
- Passing arrays as function parameters;
- Arrays as members of structures;
- Using structure type data as array elements;
- Arrays as dynamic data structures; and
- Manipulating character arrays and strings.

These aspects of arrays are covered later in the following chapters:

Chapter 8 : strings

Chapter 9 : Functions

Chapter 10 : Structures

Chapter 11 : Pointers

Just Remember

- We need to specify three things, namely, name, type and size, when we declare an array.
- Always remember that subscripts begin at 0 (not 1) and end at size-1.
- Defining the size of an array as a symbolic constant makes a program more scalable.
- Be aware of the difference between the “kth element” and the “element k”. The kth element has a subscript k-1, whereas the element k has a subscript of k itself.
- Do not forget to initialize the elements; otherwise they will contain “garbage”.
- Supplying more initializers in the initializer list is a compile time error.
- Use of invalid subscript is one of the common errors. An incorrect or invalid index may cause unexpected results.
- When using expressions for subscripts, make sure that their results do not go outside the permissible range of 0 to size-1. Referring to an element outside the array bounds is an error.
- When using control structures for looping through an array, use proper relational expressions to eliminate “off-by-one” errors. For example, for an array of size 5, the following for statements are wrong:

```
for(i=1; i<=5; i++)
```

```
for(i=0; i<=5; i++)
```

```
for(i=0; i ==5; i++)
```

```
for(i=0; i<4; i++)
```

- Referring a two-dimensional array element like `x[i,j]` instead of `x[i][j]` is a compile time error.
- When initializing character arrays, provide enough space for the terminating null character.
- Make sure that the subscript variables have been properly initialized before they are used.

- Leaving out the subscript reference operator [] in an assignment operation is compile time error.
- During initializations of multi-dimensional arrays, it is an error to omit the array size for any dimension other than the first.

Case Studies

1. Median of a List of Numbers

Problem:

When all the items in a list are arranged in an order, the middle value which divides the items into two parts with equal number of items on either side is called the median. Odd numbers of items have just one middle value while even numbers of items have two middle values. The median for even number of items is therefore designated as the average of the two middle values.

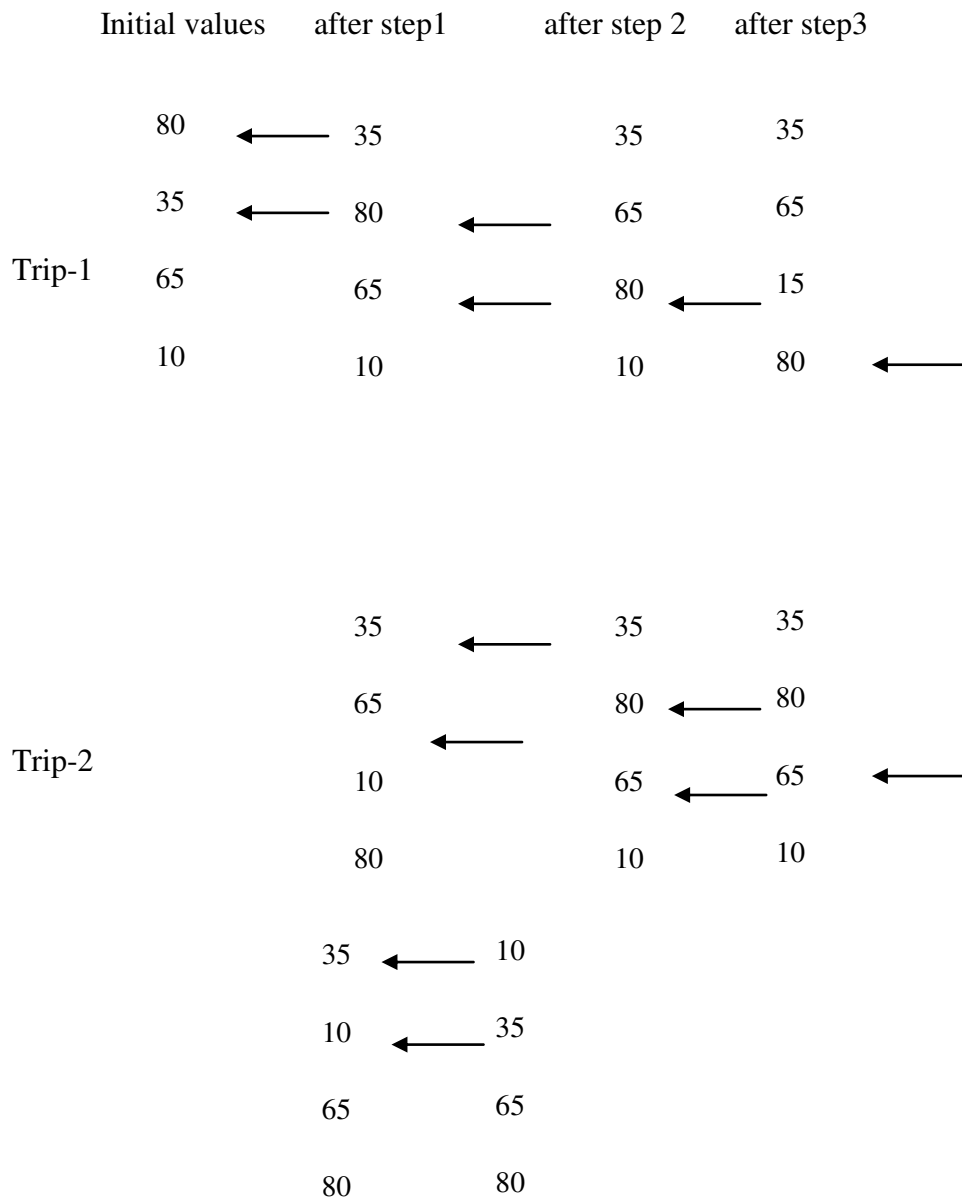
The major steps for finding the median are as follows:

1. Read the items into an array while keeping a count of the items.
2. Sort the items in increasing order.
3. Compute median.

The program and sample output are shown in fig. 7.10. The sorting algorithm used as follows:

1. Compare the first two elements in the list, say $a[1]$, and $a[2]$. If $a[2]$ is smaller than $a[1]$, then interchange their values.
2. Compare $a[2]$ and $a[3]$; interchange them if $a[3]$ is smaller than $a[2]$;
3. Continue this process till the last two elements are compared and interchanged.
4. Repeat the above steps $n-1$ times.

In repeated trips through the array, the smallest elements 'bubble up' to the top. Because of this bubbling up effect, this algorithm is called bubble sorting. The bubbling effect is illustrated below for four items,



During the first trip three pairs of items are compared and interchanged whenever needed. It should be noted that the number 80, the largest among the items, has been moved to the bottom at the end of the first trip. This means that the element 80 (the last item in the new list) need not be considered any further. Therefore, trip-2 requires only two pairs to be compared. This time, the number 65(the second largest value) has been moved down the list. Notice that each trip brings the smallest value 10 up by one level.

The number of steps required in a strip is reduced by one for each trip made. The entire process will be over when a trip contains only one step. If the list contains n elements, then the number of comparisons involved would be $n(n-1)/2$

Program

```
#define N 10

main()

{

int i, j, n;

float meadian, a[N];

printf("Enter the number of items\n");

scanf("%d", &n);

/*reading items into array a*/

printf(" inpu %d values \n", n);

for(i=1; i <=n ; i++)

scanf("%f", &a[i]);

/*sorting begins */

{ /* trip-i begins*/

for(i=1; i <=n-i; j++)

if(a[i] <= a[j+1])

t = a[j];

a[j] = a[j+1];

a[j+1] = t;
```

```

        }
else
continue;
    }
}

/*sorting ends*/

/*calculation of median*/

if(n % == 0)

median = (a[n/2] + a[n/2+1])/2.0;

else

median = a[n/2 + 1];

/* printing */

for(i=1; i <= n; i++)

printf("%f", a[i]);

printf("\n \n"median is %f\n", median);

}

```

Output

Enter the number of items

5

Input 5 values

1.111 2.222 3.333 4.444 5.555

5.555000 4.444000 3.333000 2.222000 1.111000

Median is 3.333000

Enter the number of items

6

Input 6 values

3 5 8 9 4 6

9.000000 8.000000 6.000000 5.000000 4.000000 3.000000

Median is 5.500000

Fig. 7.10 Program to sort list of numbers and determine median

2. Calculation of Standard Deviation

In statistics, standard deviation is used to measure deviation of data from its mean. The formula for calculating standard deviation of n items is

$$s = \sqrt{\text{variance}}$$

where

$$\text{variance} = \frac{1}{n} \sum_{i=1}^n (x_i - m)^2$$

and

$$m = \text{mean} = \frac{1}{n} \sum_{i=1}^n x_i$$

The algorithm for calculation the standard deviation is as follows:

1. Read n items
2. Calculate sum and mean of the items.
3. Calculate variance.

4. Calculate standard deviation.

Complete program with sample output is shown in fig.7.11

Program

```
#include<math.h>

#define MAXSIZE 100

main ()
{
    int i,n;

    float value [MAXSIZE], deviation,
           sum, sumsqr, mean, variance, stddeviation;

    sum = sumsqr = n = 0;

    printf("Inut values input-1 to end\n");

    for(i=1; i< MAXSIZE; i++)
    {
        scanf("%f, &value[i]);

        if(value[i] == -1);

        break;

        sum += value[i];

        n += 1;

    }

    mean = sum/(float)n;
```

```

for (i = 1; i<=n; i++)
{
    deviation = value[i] – mean
    sumsqr += deviation * deviation;
}

variance = sumsqr/(float);

stddeviation = sqrt(variance);

printf(“\nNumber of items: %d\n”,n);

prinnt(“Mean :%f\n”, mean);

printf(“Standard deviation: %f\n”, stddeviation);

}

```

Output

Input values: input -1 to end

65 9 27 78 12 20 33 49 -1

Number of items: 8

Mean: 36.625000

Standard deviation: 23.510303

Fig 7.11 Program to calculate standard deviation

3. Evaluating a Test

A test consisting of 25 multiple – choice item is administered to a batch of 3 student. Correct answers and student response are tabulated as shown below:


```

#define ITEMS 25

main ()
{
    char key [ITEMS+1], response[ITEMS+1];

    int count, I, student, n,

        correct[ITEMS+1];

    /* Reading of Correct answers */

    printf("Input key to the items\n");

    for(i=0; i<ITEMS; i++)

        scanf("%c", &key[i]);

        scanf("%c", &key[i])

        key[i] = "\0";

    /* Evaluation begins */

    for (student = 1; student <= STUDENTS; student++)

    {

        /*Reading student responses and counting correct ones*/

        count = 0;

        printf("\n");

        printf("Input responses of student-%d\n", student);

        for(i=0, i<ITEMS; i++)

            scanf("%c", &response[i]);

```

```

scanf("%c, &response[i]);

response[i] = '\0';

for( i=0; i<ITEMS; i++)

    correct[i] =0;

for(i=0; i<ITEMS; i++)

    if(response[i] == key[i])

    {

        count = count + 1;

        correct[i] = 1;

    }

/* printing of results */

printf("\n");

printf("Student-%d\n", student);

printf("Score is %d out of %d\n", count, ITEMS);

printf("Response to the items below are wrong\n");

n = 0;

for(i=0; I <ITEMS i++)

    if(correct[i] ==0)

    {

printf("%d", i+1);

n = n+1;

```

```

    }

    if(n ==0)

        printf("NIL\n");

    printf("\n");

}    /* Go to next student */

*/ Evaluation and printing ends */

}

```

Output

Input key t the items

abcdabcdabcdabcdabcdabcdabcdabcd

Input responses of student-1

abcdabcdabcdabcdabcd

Student-1

Score is 25 out of 25

Response to the following items wrong

NIL

Input responses of student-2

Abcdabcdabcdabcdabcd

Student-2

Score is 13 out of 25

Response to the following items are wrong

5 6 7 8 17 18 21 22 23 25

Input response of student-3

Aaaaaaaaaaaaaaaaaaaaaa

Student-3

Score is 7 out of 25

Response to the following items is wrong

2 3 4 6 7 8 10 11 12 14 15 16 18 19 20 22 23 24

Fig 7.12 Program to evaluate responses to a multiple-choice test

4. Production and Sales Analysis

A company manufactures five categories of product and the numbers of items manufactured and sold are recorded product-wise every week in a month. The company reviews its production schedule at every month-end. The review may require one or more of the following information:

- a) Value of weekly production and sales.
- b) Total value of all the products manufactured.
- c) Total value of all the products sold.
- d) Total value of each products sold.
- e) Total value of each product, manufactured and sold.

Let us represent the products manufactured and sold by two two-dimensional arrays M and S respectively. Then

M =

M11	M12	M13	M14	M15
M21	M22	M23	M24	M25
M31	M32	M33	M34	M35
M41	M42	M43	M44	M45

S =

S11	S12	S13	S14	S15
S21	S22	S23	S24	S25
S31	S32	S33	S34	S35
S41	S42	S43	S44	S45

Where M_{ij} represents the number of j th type product manufactured in i th week and S_{ij} the number of j th product sold in i th week. We may also represent the cost of each product by a single dimensional array C as follows:

Where C_j is the cost of j th type product.

We shall represent the value of products manufactured and sold by two value arrays, namely, $Mvalue$ and $Svalue$. Then

$$Mvalue[i][j] = M_{ij} \times C_j$$

$$Svalue[i][j] = S_{ij} \times C_j$$

A program to generate the required outputs for the review meeting is shown in Fig.7.13. the following additional variables are used:

$Mweek[i]$ = value of all the products manufactured in week i

$$= \sum_{j=1}^5 Mvalue[i][j]$$

$Sweek[i]$ = Value of all the products in week i

$$= \sum_{j=1}^5 Svalue[i][j]$$

$Mproduct[j]$ = Value of j th type product manufactured during the month

$$= \sum_{i=1}^4 Mvalue[i][j]$$

$Sproduct[j]$ = Value of j th type product sold during the month

$$= \sum_{i=1}^4 Svalue[i][j]$$

Mtotal = Total value of all the products manufactured during the month

$$= \sum_{i=1}^4 Mweek[i] \sum_{j=1}^5 Mproduct[j]$$

Stotal = Total value of all the products sold during the month

$$= \sum_{i=1}^4 Sweek[i] \sum_{j=1}^5 Sproduct[j]$$

Program

```

main ()
{
int M[5] [6], S[5][6], C[6],
Mvalue [5] [6], Svalue[5][6],
Mweek[5], Sweek[5],
Mproducts[6], Sproducts[6],

Mtotal, Stotal, i, j, number;

/* Input data */

printf("Enter products manufactured week_wise \n");

printf("M11, M12, -----, M21, M22,-----etc\n");

for(i=1; i<=4, i++)

for(j=1; j<=5 j++)

scanf("%d", &M[i][j]);

printf("Enter products sold week_wise\n");

```

```

printf("S11, S12,-----, S21,S22-----etc\n");

for(i=1; i<=; i++)

for(j=1; j<=5; j++)

scanf("%d" &S[i][j]);

printf("Ente cost of each product\n");

for(j=1; j<=5; j++)

scanf("%d", &C[j]);

/* Value matrices of production and sales */

for(i=1; i<=4; i++)

for(j=1;j<=5;j++)
{

Mvalue[i][j] = M[i][j] * C[j]

Svalue[i][j] = S[i][j] *C[j]

}

/* Total value of weekly production and sales */

for(i=1; i<=4; i++)

{

Mweek[i] = 0;

Sweek[i] = 0;

for(j=1; j<=5; j++)

{

```

```

Mweek[i]   += Mvalue[i][j];

Sweek[i]   += Svalue[i][j];

}

}

/* Monthly value of product_wise production and sales */

for(j=1; j<=5; j++)

{

    Mproduct[j] = 0;

    Sproduct [j] = 0;

    for(i=1; I <=4; i++)

    {

        Mproduct[j] += Mvalue[i][j];

        Sproduct[j] += Svalue[i][j];

    }

}

/* Grand total of production and sales values */

Mtotal = Stotal = 0;

for(i=1; i<=4; i++)

{

Mtotal     += Mweek[i];

Stotal     += Sweek[i];

```

```

}

/*****

Selection and printing of information required

*****/

printf("\n\n");

printf("Following is the list of things you can\n")

printf("request for Enter appropriate item number\n");

printf("and press RETURN key\n\n");

printf("1 . Value matrices of production & sales\n");

printf("2. Total value of weekly production & sales\n");

printf("3. Product_wise monthly value of production &*);

printf(" Sales\n");

printf("4. Grand total value of production & sales \n");

printf("5. Exit\n");

number = 0;

while (1)

{      /* Beginning of while loop */

printf("\n\n ENTER YOUR CHOICE:");

scanf("5d", &number);

printf("\n");

if(number ==5)

```

```

{
printf("GOOD BYE\n\n");
break;
}
switch(number)
{
    /*Beginning of switch */
    /* VALUE MATRICES */
    Case 1;
printf(" VALUE MATRIX OF PRODUCTION\n\n");
for(i=1 i<=4; i++)
{
printf("week(%d)\t", i);
for(j=1, j<=5; j++)
printf("%7d", Mvalue[i][j]);
printf("\n");
}
printf("\n VALUE MATRIX OF SALEES\n\n");
for(i=1 i<=4 i++)
{
printf("\n Week(%d)\t",i);
for(j=1; j<=5; j++)

```

```
printf(“%7d”, Svalue[i][j]);
```

```
printf(“\n”);
```

```
}
```

```
break;
```

```
/* WEEKLY ANALYSIS */
```

```
Case 2:
```

```
printf(“ TOTAL WEEKLY PRODUCTION & SALES\n\n”);
```

```
printf(“          PRODUCTION  SALESS\n”);
```

```
printf(“          -----  -----  \n”);
```

```
for(i=1; i <=4; i++)
```

```
printf(“ Week(%d)\t”, i);
```

```
printf(“ %7d\t%7d\n”, Mweek[i], Sweek[i]);
```

```
}
```

```
break;
```

```
/* PRODUCT WISE ANALYSIS */
```

```
Case 3:
```

```
printf(“ PRODUCT_WISETOTALPRODUCTION &”);
```

```
printf(“SALES\n\n”);
```

```
printf(“          PRODUCTION  SALESS\n”);
```

```
printf(“          -----  -----  \n”);
```

```
for(j=1; j<=5; j++)
```

```

printf(" Product(%d)\t", i);

printf(" %7d\t%7d\n", Mproduct[j], Sproduct[j]);

}

break;

/* GRAND TOTALS */

Case 4:

printf(" GRAND TOTAL OF PRODUCTION * SALES \n");

printf("\n Total production = %d\n", Mtotal);

printf("Total sales = %d\n", Stotal);

break;

/* DEFAULT */

default      :

printf(" Wrong choice, select again\n\n");

break;

}      /* End of switch */

}      /* End of while loop */

printf(" Exit from the program\n\n");

}      /* End of main */

```

Output

Enter products manufactured week_wise

M11, M12, -----, M21, M22, -----etc

11 15 12 14 13

13 13 14 15 12

12 16 10 15 14

14 11 15 13 12

S11, S12, -----S21, S22, -----etc

10 13 9 12 11

12 10 12 14 10

11 14 10 14 12

12 10 13 11 10

Enter cost of each product

10 20 30 15 25

Following is the list of things you can

Request for. Enter appropriate item number

And press RETURN key

1. Value matrices of production & sales
2. Total value of weekly production & sales
3. Product_wise monthly value of production & sales
4. Grand total value of production & sales
5. Exit

ENTER YOUR CHOICE: 1

VALUE MATRIX OF PRODUCTION

Week(1)	110	300	360	210	325
Week(2)	130	260	420	225	300
Week(3)	120	320	300	225	350
Week(4)	140	220	450	185	300

VALUE MATRIX OF SALES

Week(1)	100	260	270	180	275
Week(2)	120	200	360	210	250
Week(3)	110	280	300	210	300
Week(4)	120	200	390	165	250

ENTER YOUR CHOICE: 2

TOTAL WEEKLY PRODUUCTION & SALES

	<u>PRODUCTION</u>	<u>SALE</u>
Week(1)	1305	1085
Week(2)	1335	1140
Week(3)	1315	1200
Week(4)	1305	1125

ENTER YOUR CHOICE:3

PRODUCT_WISE TOTAL PRODUCTION & SALES

	PRODUCTION	SALE
Product(1)	500	450

Product(2)	1100	940
Product(3)	1530	1320
Product(4)	855	765
Product(5)	1275	1075

ENTER YOUR CHOICE: 4

GRAND TOTAL OF PRODUCTION & SALES

Total production = 5260

Total sales = 4550

ENTER YOUR CHOICE:5

GOOD BYE

Exit from the program

Fig. 7.13 Program for production and sales analysis.

8 CHARACTER ARRAYS AND STRINGS

Key Terms

String | strcat | strcmp | strcpy | strstr

8.1 INTRODUCTION

A string is sequence of characters that is treated as single data item. We have used string in a number of examples in the past any group of characters (except double sign) defined between double quotation marks is a string constant. Example:

```
'Man is obviously made to think'
```

If we want to include a double quote in the sting to be printed, then we may use it with a back slash as shown below.

```
"\" Man is obviously made to think,\" said Pascal,"
```

For example,

```
printf ("\" Well Done!\"")
```

will output the string

```
"Well Done !"
```

While the statement

```
printf ("Well Done !");
```

will output the string

```
Well Done !
```

Character string is often used to build meaningful and readable programs. The common operations performed on character strings include:

- Reading and writing strings.
- Combining strings together.
- Copying one string to another
- Comparing strings for equality.
- Extracting a portion of a string.

In this chapter, we shall discuss these operations in detail and examine library functions that implement them.

8.2 DECLARING AND INITIALIZING STRING VARIABLES

C does not support sting as a data type. However, it allows us to represent strings as character arrays. In C, therefore, a string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of string variable is:

```
char string_name [size];
```

The size determines the number of characters in the string_name. Some examples are:

```
char city [10];
```

```
char name[30];
```

When the compiler assigns a character string to character array, it automatically supplies a null character ('0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the sting plus one.

Like numeric arrays, character arrays may be initialized when they are declared, C permits a character array to be initialized in either of the following two forms:

```
char city-[9] = "NEW YORK";
```

```
char city-[9] = {'N','E','W',' ','Y','O','R','K','\0'};
```

The reason that city had to be 9 elements long is that the string NEW YORK contains 8 characters and one element space is provided for the null terminator. Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator.

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example, the statement

```
char string [ ] = {'G','O','O','D','\0'};
```

defines the array string as a five element array.

We can also declare the size much larger than the string size in the initialize. That is, the statement.

```
char str [10] = 'GOOD';
```

is permitted. In this case, the computer creates a character array of size. 10, places the value "GOOD" in it, terminates with the null character, and initializes all other elements to NULL. The storage will look like:

G	O	O	D	\0	\0	\0	\0	\0	\0
---	---	---	---	----	----	----	----	----	----

However, the following declaration is illegal.

```
char str2[3] = 'GOOD';
```

This will result in a compile time error. Also not that we cannot separate the initialization from declaration. That is,

```
char str3[5];  
  
str3 = "GOOD";
```

is not allowed. Similarly,

```
char s1[4] = "abc";  
  
char s2[4];  
  
s2 = s1; /* Error */
```

is not allowed. An array name cannot be used as the left operand of an assignment operator.

Terminating Null Character

You must be wondering, ‘why do we need a terminating null character?’ as we know, a string is not a data type in /c., but it is considered a data structure stored in an array the string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the ‘end-of-string’ marker.

8.3 READING STRINGS FROM TERMINAL

Using scanf Function

The familiar input function **scanf** can be used with %s format specification to read in a string of characters. Example:

```
char address[10]  
  
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal.

NEW YORK

Then only the string “NEW “ will be read into the array address, since the blank space after the word “NEW” will terminate the reading of string.

The scanf function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character. Note that unlike previous scanf calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

The address array is created in the memory as shown below:

N	E	W	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Note that the unused locations are filled with garbage.

If we want to read the entire line, “NEW YORK”, then we may use two character arrays of appropriate sizes. That is,

```
char adr1[5], adr2[5];  
  
scanf(“%s %s”, adr1, adr2);
```

With the line of text

NEW YORK

Will assign the string “NEW “ to adr1 and “YORK”” to adr2.

Program 8.1

Write a program to read a series of words from a terminal using scanf function.

The program shown in fig.8.1 reads four words and displays then on the screen. Note that the string ‘Oxford Road’ is treated as two words while the string ‘Oxford-Road’ as one word.

Program

```
main ()
```

```

{
    char word1[40], word2[40], word3[40], word4[40]

    printf("Enter text :\n");

    scanf("%s %s", word1, word2);

    scanf("%s", word3);

    scanf("%s", word4)

    printf("\n");

    printf("word1 = %s\nword2 = %s\n", word1, word2);

    printf("word3= %s\nword4 = %s\n", word3, word4);

}

```

Output

Enter text :

Oxford Road, Londan M17ED

word1 = Oxford

word2 = Road,

word3 = Londan

word4 = M17ED

Enter text :

Oxford-Road, Londan-M17ED United Kingdom

word1 = Oxford-Road

word2 = Londan-M17ED

word3 = United

word4 = Kingdom

Fig. 8.1 Reading a series of words using scanf function

we can also specify the field width using the form `%ws` in the scanf statement for reading a specified number of characters from the input string. Example:

```
scanf("%ws", name);
```

here, two things may happen.

1. The width `w` is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.
2. The width `w` is less than the number of characters in the string. The excess characters will be truncated and left unread.

Consider the following statements:

```
Char name[10]
```

```
Scanf("%5s", name);
```

The input string RAM will be stored as

R	A	M	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

The input string KRISHNA will be stored as:

K	R	I	S	H	\0	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Reading a Line Text

We have seen just now that **scanf** with `%s` or `%ws` can read only strings without whitespaces. That is, they cannot be used for reading a text containing more than one word. However, C supports a format specification known as the edit set conversion code `%[.]` that can

be used to read a line containing a variety of characters, including whitespaces. Recall that we have used this conversion code in Chapter 4.

For example, the program segment

```
char line [80];  
  
scanf("%[^\n]", line);  
  
printf("%s", line);
```

Will read a line of input from the keyboard and display the same on the screen. We would very rarely use this method, as C supports an intrinsic string function to do this job. This is discussed in the next section.

Using `getchar` and `gets` Functions

We have discussed in Chapter 4 as to how to read a single character from the terminal, using the function `getchar`. We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, entire line of text can be read and stored in an array. The reading is terminated when the newline character(‘\n’) is entered and the null character is then inserted at the end of the string. The `getchar` function call takes the form:

```
char ch;  
  
Ch = getchar();
```

Note that the `getchar` function has no parameters.

Program 8.2 Write a program to read a line of text containing a series of words from the terminal.

The program shown in fig.8.2 can read a line of text (up to a maximum of 80 characters) into the string `line` using `getchar` function. Every time a character is read, it is assigned to its location in the string `line` and then tested for newline character. When the newline character is replaced by the null character to indicate the end of character string.

When the loop is exited, the value of the index `c` is one number higher than the last character position in the string (since it has been incremented after assigning the new character to the string). Therefore the index value `c-1` gives the position where the null character is to be stored.

Program

```
#include <stdio.h>

main ()
{
    char line[81], character

    int c;

    c = 0;

    printf("Enter text. Press <Return> at end\n");

    do
    {
        character = getchar();

        line[c] = character

        c++;

    }

    while(character != '\n');

    c = c -1;

    line[c] = '\0';

    printf("\n%s\n", line);
```

```
}
```

Output

Enter text, press <Return> at end

Programming in C is interesting

Programming in C is interesting

Enter text. Press <Return> at end

National Center for Expert System, Hyderabad.

National Center for Expert System, Hyderabad.

Fig .8.2 Program to read a line of text from terminal

Another and more convenient method of reading a string of text containing whitespace is to use the library function `gets` available in the `<stdio.h>` header file. This is a simple function with one string parameter and called as under:

```
gets(str);
```

`str` is a string variable declared properly. It reads characters into `str` from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike `scanf`, it does not skip whitespaces. For example the code segment

```
char line[80];
```

```
gets (line);
```

```
printf ("%s", line);
```

reads a line of text from the keyboard and displays it on the screen. The last two statements may be combined as follows:

```
printf ("%s", gets(line));
```

(Be careful not to input more character that can be stored in the string variable used. Since C does not check array-bounds, it may cause problems)

C does not provide operators that work on strings directly. For instance we cannot assign one string to another directly. For example, the assignment statements.

```
string = "ABC";  
string2 = string2;
```

are not valid. If we really want to copy the characters in string2 into string1, we may do so on a character-by-character basis.

Program 8.3 Write a program to copy one string into another and count the number of characters copied.

The program is shown in fig.8.3. We use a for loop to copy the characters contained inside string2 into the string1. The loop is terminated when the null character is reached. Note that we are again assigning a null character to the string1.

Program

```
main ( )  
{  
    char string1[80], string[80];  
    int i;  
    printf("Enter a string \n");  
    printf("?");  
    scanf("%s", string2);  
    for( i = 0; string2[i] != 0; i++)  
        string1[i] = string2[i]
```

```
string1[i] = '\0';  
  
printf("\n");  
  
printf("%s\n", string1);  
  
printf("Number of characters = %d\n", i);  
  
}
```

Output

```
Enter a string  
?Manchester  
Manchester  
  
Number of characters = 10  
  
Enter a string  
? Westminister  
Westminister  
  
Number of characters = 11
```

Fig 8.3 Copying one string into another

Program 8.4 The program in fig 8.4 shows how to write a program to find the number of vowels and consonants in a text string. Elucidate the program and flowchart for the program.

Algorithm

Step 1 – Start

Step 2 – Read a text string (str)

Step 3 – Set vow = 0. Cons = 0, i =0

Step 4 – Repeat steps 5-8 while {str[i]!= '\0'}

Step 5 – if str[i] = 'a' OR str[i] = 'A' OR str[i] = 'e' OR str[i] = 'E' OR str[i] = 'i'

OR str[i] = 'I' OR str[i] = 'o' OR str[i] = 'O' OR str[i] = 'u' OR str[i] = 'U'

goto Step 6 else goto step 7

Step 6 – Increment the vowels counter by 1 (vow = vow+1)

Step 7 – Increment the consonants counter by 1 (cons = cons+1)

Step 8 – i = i + 1

Step 9 – Display the number of vowels and consonants (vow, cons)

Step 10 – Stop

Flow chart

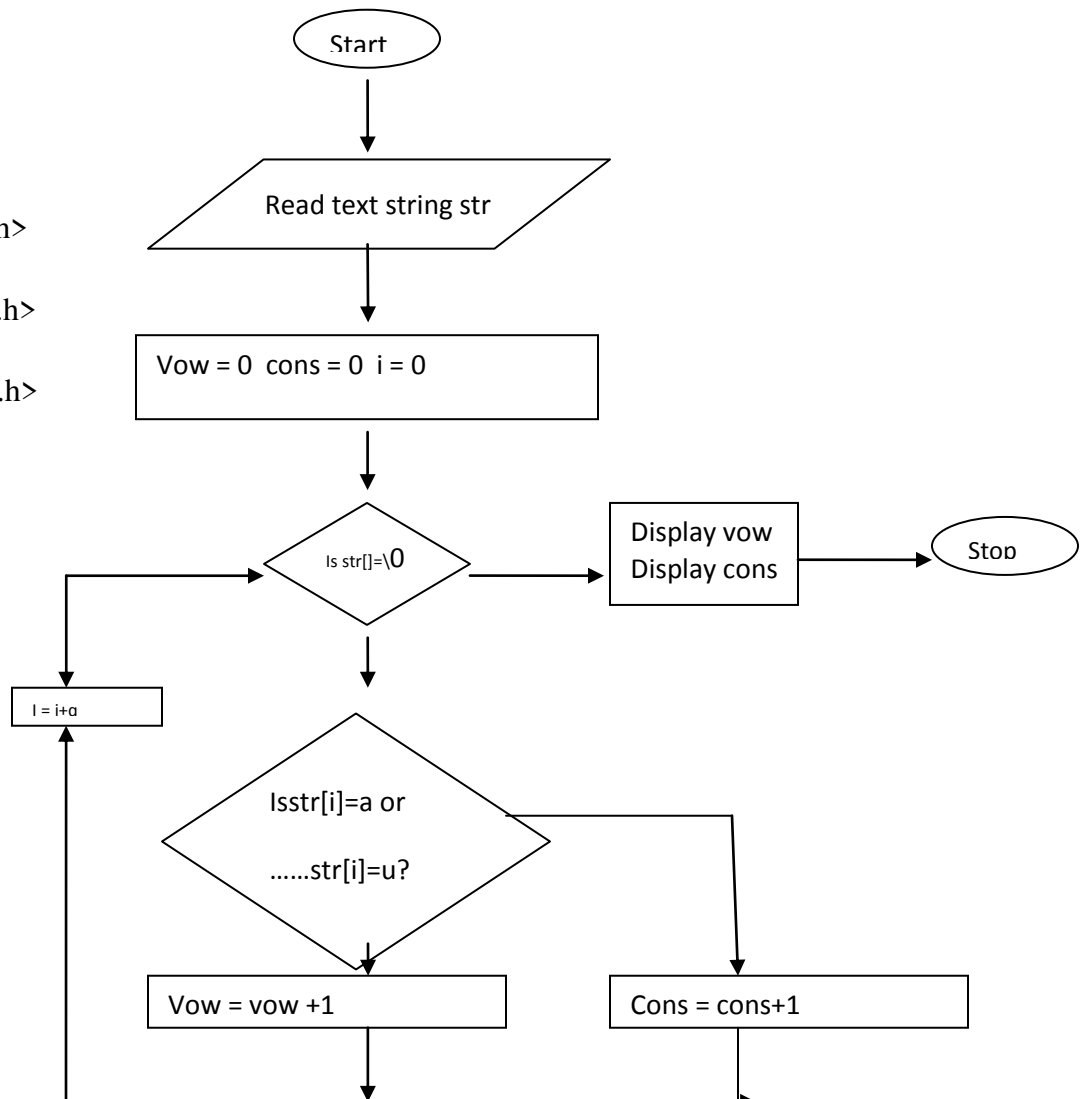
Program

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

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

```
#include<string.h>
```

```
void main ()
```



```

{
char str[30];

int vow = 0, cons = 0, i= 0;

    clrscr ();

    printf("Enter a string:");

    gets (str);

    while (str[i] != '\0')

    {

        if(str[i] == 'a' || str[i] == 'A' || str[i] == 'e' || str[i] == 'E' || str[i] == 'i'

        || str[i] == 'I' || str[i] == 'o' || str[i] == 'O' || str[i] == 'u' || str[i] == 'U'

        vow++;

    else

        cons++;

    i++;

    }

    printf("\nNumber of Vowels = %d", vow);

    printf("\nNumber of Consonants = %d",cons);

    getch();

}

```


Output

Enter a string : Chennai

Number of Vowels = 3

Number of Consonants = 4

Fig 8.4 Program to find the number of vowel and consonants in a text string

8.4 WRITING STRINGS TO SCREEN

Using printf Function

We have used extensively the printf function with `%s` format to print strings to the screen. The format `%s` can be used to display an array of characters that is terminated by the null character. For example, the statement

```
printf (“%s”, name);
```

can be used to display the entire contents of the array name.

We can also specify the precision with which the array is displayed. For instance, the specification

```
%10.4
```

indicates that the first four characters are to be printed in a field width of 10 columns.

However, if we include the minus sign in the specification (e.g. `%-10.4s`), the string will be printed left-justified. The program 8.4 illustrates the effect of various `%s` specifications.

Program 8.5 Write a program to store the string ‘United Kingdom’ in the array country and display the string under various format specifications.

The program and its output are shown in fig 8.5. The output illustrates the following features of the `%s` specifications.

1. When the field width is less than the length of the string, the entire string is printed.
2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed left-justified.
5. The specification `% .ns` prints the first `n` characters of the string.

Program

```
main ()
{
    char country[15] = "United Kingdom";

    printf("\n\n");

    printf(*123456789012345*\n");

    printf("-----\n*");

    printf("%15s\n", country);

    printf("5s\n", country);

    printf("15.6s\n", country);

    printf("%-15.6s\n*", country);

    printf("%.3s\n", country);

    printf("%s\n", country);

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

Output

```
*123456789012345*
```

```
-----
```

```
United Kingdom
```

```
United Kingdom
```

```
    United
```

```
United
```

```
Uni
```

```
United Kingdom
```

```
-----
```

Fig 8.5 Writing strings using %s format

The printf of UNIX supports another nice feature that allows for variable field width or precision. For instance

```
printf(“ %*, *s\n”, w, d, string);
```

prints the first d characters of the string in the field width of w.

This feature comes in handy for printing a sequence of characters. Program 8.5 illustrates this.

Program 8.6 Write a program using for loop to print the following output.

```
C
```

```
CP
```

```
Cpr
```

```
CPro
```

```

.....
.....
CProgramming
CProgramming
.....
.....
CPro
CPr
CP
C

```

The outputs of the program in Fig.8.6 for variable specifications %12*s,%*s and %*1s are shown in Fig.8.7. which further illustrates the variable field width and the precision specifications.

Program

```

main ( )
{
    int c, d;

    char string[] = "CProgramming";

    printf(".....\n");

    for (c =0 ; c <=11, c++)
    {
        d = c +1;

```

```

    printf("|%-12,*s\n", d, string);
}

printf("|-----\n");

for ( c = 11; c >=0; c--)
{
    d = c +1;

    printf("|%-12,*s\n", d, string);
}

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

```

Output

```

C
CP
CPr
CPro
CProg
CProgr
CProgra
CProgram
CProgramm
CProgrammi

```

CProgrammin
 CProgramming
 CProgrammin
 CProgrammi
 CProgramm
 CProgram
 CProgra
 CProgr
 CProg
 CPro
 CPr
 CP
 C

Fig 8.6 Illustration of variable field specifications by printing sequences of characters

C	C	C
CP	CP	C
CPr	CPr	C
CPro	CPro	C
CProg	CProg	C
CProgr	CProgr	C
CProgra	CProgra	C

CProgram	CProgram	C
CProgramm	CProgramm	C
CProgrammi	CProgrammi	C
CProgrammin	CProgramming	C
CProgrammin	CProgramming	C
CProgrammi	CProgrammi	C
CProgramm	CProgramm	C
CProgram	CProgram	C
CProgra	CProgra	C
CProgr	CProgr	C
CProg	CProg	C
CPro	CPro	C
CPr	CPr	C
CP	CP	C
C	C	C

Fig 8.7 Further illustrations of variable specifications

Using putchar and puts Functions

Like getchar, C supports another character handling function putchar to output the values of character variables. It takes the following form:

```
char ch = 'A';
```

```
putchar (ch);
```

The function putchar requires one parameter. This statement is equivalent to:

```
printf("%c", ch);
```

We have used putchar function in Chapter 4 to write characters to the screen. We can use this function repeatedly to output a string of characters stored in an array using a loop. Example:

```
char name[6] = "PARIS"
```

```
for (i=0, i<5 i++)
```

```
    putchar (name[i];
```

```
    putchar ('\n');
```

another and more convenient way of printing string values is to use the function puts declared in the header file <stdio.h>. This is a one parameter function and invoked as under:

```
puts (str);
```

where str is a string variable containing a string value. This prints the value of the string variable str and then moves the cursor to the beginning of the next line on the screen. For example, the program segment

```
char line[80];
```

```
gets (line);
```

```
puts (line);
```

reads a line of text from the keyboard and displays it on the screen. Note that the syntax is very simple compared to using the scanf and printf statements.

8.5 ARITHMETIC OPERATIONS ON CHARACTERSS

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted

into an integer value by the system. The integer value depends on the local character set of the system.

To write a character in its integer representation, we may write it as an integer. For example, if the machine uses the ASCII representation, then

```
x = 'a';  
  
printf ("%d\n",x);
```

will display the number 97 on the screen.

It is also possible to perform arithmetic operations on the character constants and variable. For example,

```
x = 'z'-1
```

is valid statement. In ASCII, the value of 'z' is 122 and therefore, the statement will assign the value 121 to the variable x.

We may also use character constants in relational expressions. For example, the expression

```
ch >= 'A' && ch <= 'Z'
```

would test whether the character contained in the variable ch is an upper-case letter.

We can convert a character digit to its equivalent integer value using the following relationship:

```
x = character - '0';
```

where x is defined as an integer variable and character contains the character digit. For example, let us assume that the character contains the digit '7'.

Then,

```
x = ASCII value of '7' - ASCII value of '0'
```

= 55-48

= 7

The C library supports a function that converts a string of digits into their integer values. The function takes the form

```
x = atoi(string)
```

x is an integer variable and string is a character array containing a string of digits. Consider the following segment of a program:

```
number = "1988";  
  
year   = atoi(number);
```

number is a string variable which is assigned the string constant "1988". The function **atoi** converts the string "1988" (contained in number) to its numeric equivalent 1988 and assigns it to the integer variable year. String conversion functions are stored in the header file <std.lib.h>

Program 8.7 Write a program which would print the alphabet set a to z and A to Z in decimal and character form.

The program is shown in fig.8.8 In ASCII character set, the decimal numbers 65 to 90 represent upper case alphabets and 97 to 122 represent lower case alphabets. The values from 91 to 96 are excluded using an if statement in the for loop.

Program

```
main ()  
  
{  
  
    char c;  
  
    printf("\n\n");  
  
    for(c =65; c <= 122; c = c+1)
```

```

{
    If (c >90 && c <97)

        continue;

    printf("|4%d = %c", c, c);

}

printf("\n");

}

```

Output

| 65 – A | 66 – B | 67 – C | 68 – D | 69 – E | 70 – F

| 71 – A | 72 – B | 73 – C | 74 – D | 75 – E | 76 – L

.....
.....
.....

| 119 – W | 120 – X | 121 – Y | 122 – Z |

8.6 PUTTING STRINGS TOGETHER

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;
```

```
string2 = string1 + "hello";
```

are not valid. The characters from string 1 and string 2 should be copied into the string3 one after the other. The size of the array string3 should be large enough to hold the total character.

The process of combining two strings together is called concatenation. Program 8.9 illustrates the concatenation of three strings.

Program 8.8 The names of employees of an organization are stored in three arrays, namely `first_name`, `second_name`, and `last_name`. Write a program to concatenate the three parts into one string to be called `name`.

The program is given in Fig.8.9. Three for loops are used to copy the three strings. In the first loop, the characters contained in the `first_name` are copied into the variable `name` until the null character is reached. The null character is not copied; instead it is replaced by a space by the assignment statement

```
name[i+j+1] = second_name[j];
```

If `first_name` contains 4 characters, then the value of `i` at this point will be 4 and therefore the first character from `second_name` will be placed in the fifth cell of `name`. Note that we have stored a space in the fourth cell.

In the same way, the statement

```
name [i+j+k+2] = last_name[k]
```

is used to copy the characters from `last_name` into the proper locations of `name`.

At the end, we place a null character to terminate the concatenated string `name`. In this example, it is important to note the use of the expressions `i+j+1` and `i+j+k+2`.

Program

```
main ()  
  
{  
  
int i, j, k ;  
  
char first_name[10] = {"VISQANATH"}  
  
char second_name[10] = {"PRATAP"};
```

```

char last_name[10] = {"SINGH"};

char name[30];

/* copy first name into name*/

for (i = 0; first_name[i] !='\0' i++)

    name[i] = first_name[i];

/* End first_name with a space */

    name[i] = ' ';

/* Copy second _name into name */

for ( j = 0; second_name[j] != '\0'; j++)

name [i+j+1] = ' ';

/* End second_name with a space */

    name [i+j+1] = ' ';

/* copy last_name into name */

for ( k= 0 last_name[k] !='\0' ; k++)

    name[i+j+2] = last_name[k];

/* End name with null character */

name [i+j+k+2] = '\0';

print ("\n\n");

printf("%s\n", name);

}

```

Output

VISWANATH PRATAP SINGH

Fig 8.9 Concatenation of strings

8.7 COMPARISON OF TWO STRINGS

Once again, C does not permit the comparison of two strings directly. That is, the statements such as

```
if(name1 == name2)
```

```
if(name == "ABC")
```

are not permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character, whichever occurs first. The following segment of a program illustrates this.

```
i = 0;

while(str1[i] == str2[i] &&str1[i] != '\0'

    && str2[i] !='\0')

i = i+1;

if (str1[i] == '\0' && str2[i] == '\0')

    printf("strings are equal\n");

else

    printf("strings are not equal\n");
```

8.8 STRING-HANDLING FUNCTIONS

Fortunately, the C library supports large number of string-handling functions that can be used to carry out many of the string manipulations discussed so far. Following are the most commonly used string-handling functions.

Function	Action
strcat()	concatenates two strings
strcmp()	compares two strings
strcpy()	copies one string over another
strlen()	finds the length of a string

We shall discuss briefly how each of these functions can be used in the processing of strings.

strcat () function

The strcat function joins two strings together. It takes the following form:

strcat(string1, string2)

string1 and string2 are character arrays, when the function strcat is executed, string2 is appended to string1. It does so by removing the null character at the end of string1 and placing string2 from there. The string at string2 remains unchanged. For example, consider the following three strings.

0 1 2 3 4 5 6 7 8 9 0 1

Part 1 =

V	E	R	Y								
---	---	---	---	--	--	--	--	--	--	--	--

0 1 2 3 4 5 6

Part 2 =

G	O	O	D	\0		
---	---	---	---	----	--	--

0 1 2 3 4 5 6

Part 3 =

B	A	D	\0			
---	---	---	----	--	--	--

strcat(part1,part2); will result in:

0 1 2 3 4 5 6 7 8 9 0 1

Part 1 =

V	E	R	Y		G	O	O	D	\0		
---	---	---	---	--	---	---	---	---	----	--	--

0 1 2 3 4 5 6

Part 2 =

G	O	O	D	\0		
---	---	---	---	----	--	--

While the statement strcat(part1,part3);

Will result in:

0 1 2 3 4 5 6 7 8 9 0 1

Part 1 =

V	E	R	Y		B	A	D	\0			
---	---	---	---	--	---	---	---	----	--	--	--

0 1 2 3 4 5 6

Part 3 =

B	A	D	\0			
---	---	---	----	--	--	--

We must make sure that the size of string1 (to which string2 is appended) is large enough to accommodate the final string.

strcat function may also append a string constant to a string variable. The following is valid:

```
strcat(part1, "GOOD");
```

C permits nesting of strcat functions. For example, the statement

```
strcat(strcat(string1, string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in string1.

strcmp() function

The strcmp function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first no matching characters in the strings. It takes the form:

```
strcmp(string1,string2);
```

string1 and string2 may be string variables or string constants. Examples are:

```
strcmp(name1, name2);
```

```
strcmp(name1, "John");
```

```
strcmp("Rom", "Ram");
```

Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

```
strcmp("their", "there");
```

Will return a value of -9 which is the numeric difference between ASCII "i" ASCII "r". that is "i" minus "r" in ASCII code is -9. If the value is negative, string 1 is alphabetically above string2.

strcpy() function

The strcpy function works almost like a string-assignment operator. It takes the form:

```
strcpy(string1,string2)
```

and assigns the contents of string2 to string1. String2 may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

will assign the string "DELHI" to the string variable city. Similarly, the statement

```
strcpy(city1, city2)
```

will assign the contents of the strings variable city2 to the string variable city1. The size of the array city1 should be large enough to receive the contents of city2.

strlen() function

This function counts and returns the number of characters in a string. It takes the form

$$n = \text{strlen}(\text{string})$$

where n is an integer variable, which receives the value of the length of the string. The argument may be a string constant. The counting ends at the first null character.

Program 8.9 s1, s2 and s3 are three string variables write a program to read two string constants into s1 and s2 and compare whether they are equal or not. If they are not, join them together. Then copy the contents of s1 to the variable s3 at the end, the program should print the contents of all the three variable and their lengths.

The program is shown in Fig.8.10. During the first run, the input strings are “New” and “York”. These strings are compared by the statement

$$x = \text{strcmp}(s1, s2);$$

Since they are not equal they are joined together and copied into s3 using the statement

$$\text{strcpy}(s3,s1);$$

The program outputs all the three with their lengths.

During the second run, the two strings s1 and s2 are equal, and therefore, they are not joined together in this case all the three strings contain the same string constant “London”.

Program 8.9

```
#include<string.h>

main( )
{
```

```

char s[2], s2[20], s3[20];

int x, i1, i2, i3;

printf("\n\n enter two string constants\n");

printf("?");

scanf("%s
%s", s1, s2);

/* comparing s1 and s2 */

    x = strcmp(s1, s2);

if(x !=0)

{

    printf("\n\n strings are not equal \n");

    strcat(s1, s2);

    /* joining s1 and s2 */

}

else

    printf("\n\n strings are not equal \n");

/* copying s1 s3*/

strcpy(s3, s1);

/* finding length of strings */

i1 = strlen(s1);

i2 = strlen(s2);

```

```

13 = strlen(s3);

/* out put*.

printf("\n s1 = %s\t length = % characters \n, s1, 11);

printf("\n s2 = %s\t length = % characters \n, s2, 12);

printf("\n s3 = %s\t length = % characters \n, s3, 13);

}

```

Output

Enter two string constants

? newyork

Strings are not equal

S1 = newyork length = 7 characters

S2 = York length = 4 characters

S3 = newyork length = 7 characters

Enter two string constants

? London London

Strings are equal

S1 = London length = 6 characters

S2 = London length = 6 characters

S3 = London length = 6 characters

Fig .8.10 illustration of string Fig.8.11 shows how to write a C program that reads a string and prints if it is a palindrome or not.

Program

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

    char chk= 't', str[30];

    int len, left, right;

    printf("\n enter a string:");

    scanf("%s", &str);

    len= strlen(str);

    left=0;

    right=len-1;

    while(left < right && chk == 't')

    {

        if(str[left] == str[right])

        ;

        else

        chk='f';

        left++;

        right-;
```

```

    }

    if(chk=='t')

        printf("\n the string %s is palindrome ", str);

    else

        printf("\n the string %s is not a palindrome", str);

    getch();

}

```

Output

Enter a string: nitin

The string nitin is a palindrome

Fig.8.11 program to check if a string is palindrome or not

Other string functions

The header file <string.h> contains many more string manipulation functions. They might be useful in certain situations.

strncpy

In addition to the function strcpy that copies one string to another; we have another function strncpy that copies only the left-most n characters of the source string to the target string variable. This is a three-parameter function and is invoked as follows:

```
strncpy(s1, s2, 5);
```

This statement copies the first 5 characters of the source string s2 into the target string s1. Since the first 5 characters may not include the terminating null character, we have to place it explicitly in the 6th position of s2 as shown below:

```
S1[6] = '\0';
```

Now, the string s1 contains a proper string.

strncmp

Variation of the function strcmp is the function strncmp. This function has three parameters as illustrated in the function call below:

```
strncmp(s1, s2, n);
```

This compares the left-most n characters of s1 to s2 and returns

- (a) 0 if they are equal.
- (b) Negative number, if s1 sub-string is less than s2; and
- (c) Positive number, otherwise

strncat

This is another concatenation function that takes three parameters as shown below:

```
strncat (s1, s2, n);
```

This call will concatenate the left-most n characters of s2 to the end of s1. Example:

s1

B	A	L	A	\0							
---	---	---	---	----	--	--	--	--	--	--	--

s2

G	U	R	U	S	A	M	Y	\0
---	---	---	---	---	---	---	---	----

After strncat(s1, s2, 4); execution:

B	A	L	A	G	U	R	U	\0
---	---	---	---	---	---	---	---	----

strstr

It is a two-parameter function that can be used to locate a sub-string in a string. It takes the form:

```
strstr(s1, s2);
```

```
strstr(s1, ABC);
```

The function `strstr` searches the string `s1` to see whether the string `s2` is contained in `s1`. If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a `NULL` pointer. Example,

```
if(strstr(s1, s2) == NULL)

printf("substring is not found");

else

printf("s2 is a substring of s1");
```

we also have function to determine the existence of character in a string. The function call

```
strchr(s1, 'm');
```

will locate the first occurrence of the character 'm' and the call

```
strrchr(s1, 'm');
```

will locate the last occurrence of the character 'm' in the string `s1`.

WARNING

- When allocating space of a string during declaration, remember to count the terminating null character.
- When creating an array to hold a copy of a string variable of unknown size, we can compute the size required using the expression `strlen(string name)+1`.
- When copying or concatenating one string to another, we must ensure that the target (destination) string has enough space to hold the incoming characters. Remember that no error message will be available even if this condition is not satisfied. The copying may overwrite the memory and the program may fail in an unpredictable way.
- When we use `strncpy` to copy a specific number of characters from a source string, we must ensure to append the null character to the target string, in case the number of characters is less than or equal to the source string.

8.9 TABLE OF STRING

We often use lists of character string, such as list of the names of students in a class, list of the names of employees in an organization, list of places, etc. A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list. For example, a character array `student[30]` may be used to store a list of 30 names, each of length not more than 15 characters. Shown below is a table of five cities:

C	h	A	n	d	i	g	a	R	h
M	a	D	r	a	s				
A	h	M	e	d	a	b	a	d	
H	y	D	e	r	a	b	a	d	
b	o	M	b	a	y				

This table can be conveniently stored in a character array `city` by using the following declaration:

```
char city[ ] [ ]  
  
    {  
  
        "Chandigarh",  
  
        "Madras",  
  
        "Ahmedabad",  
  
        "Hyderabad",  
  
        "Bombay"  
  
    };
```

To access the name of the i^{th} city in the list, we write

```
City [I - 1]
```

And therefore city[0] denotes “Chandigarh”, city[1] denotes “madras” and so on. This shows that once an array is declared as two-dimensional, it can be used like a one-dimensional array in further manipulations. That is, the table can be treated as a column of strings.

Program 8.11 Write a program that would sort a list of names in alphabetical order.

A program to sort the list of string in alphabetical order is given in Fig.8.12. it employs the method of bubble sorting described in case study 1 in the previous chapter.

Program

```
#define ITEM 5

#define MAXCHAR 20

main ()
{
    Char string[ITEM] [MAXCHAR], dummy [MAXCHAR]

    int i = 0, j = 0

    /* Reading the list */

    printf("Enter names of %d items /n", ITEM);

    while (i < ITEMS)

        scanf("%s", string[i++]);

    /* sorting begins */

    for (i=1, i < ITEMS; i++) /* Outer loop begins */
    {
        for (j=1; j<= ITEMS-j; j++) /* Inner loop begins*/
        {
```

```

        if (strcmp (string[j-1], string[j]) > 0)

        { /* Exchange of contents */

        strcpy (dummy, string[j-1]);

        strcpy (string[j-1], string[j]);

        strcpy(string[j], dummy);

        }

    } /* Inner loop ends */

} /* Outer loop ends */

/* sorting completed */

printf("\n Alphabetical list \n\n");

for (i=0; I < ITEMS; i++)

    printf("%s", string[i]);

}

```

Output

Enter names of 5 items

London Manchester Delhi Paris Moscow

Alphabetical list

Delhi

London

Manchester

Moscow

Paris

Fig. 8.12 Sorting of strings in alphabetical order

Note that a two-dimensional array is used to store the list of strings. Each string is read using a scanf function with %s format. Remember, if any string contains a white space, then the part of the string after the white space will be treated as another item in the list by the scanf. In such cases, we should read the entire line as a string using a suitable algorithm. For example, we can use gets function to read line of text containing a series of words. We may also use puts function in place of scanf for output.

8.10 OTHER FEATURES OF STRINGS

Other aspects of string we have not discussed in this chapter include:

- Manipulating strings using pointers
- Using string as function parameter.
- Declaring and defining strings as members of structures.

These topics will be dealt with later when we discuss functions, structures and pointers.

Just Remember

- Character constants are enclosed in single quotes and string constants are enclosed in double quotes.
- Allocate sufficient space in a character array to hold the null character at the end.
- Avoid processing single characters as strings.
- Using the address operator & with a string variable in the scanf function call is an error.
- It is a compile time error to assign a string to a character variable.
- Using a string variable name on the left of the assignment operator is illegal.
- When accessing individual characters in a string variable, it is logical error to access outside the array bounds.
- Strings cannot be manipulated with operators. Use string functions.

- Do not use string functions on an array char type that is not terminated with the null character.
- Do not forget to append the null character to the target string when the number of characters copied is less than or equal to the source string.
- Be aware the return values when using the functions strcmp and strncmp for comparing strings.
- When using string functions for copying and concatenating strings, make sure that the target string has enough space to store the resulting string. Otherwise memory overwriting may occur.
- The header file<stdio.h> is required when using standard I/O functions.
- The header file<ctype.h> is required when using character handling functions.
- The header file<stdlib.h> is required when using general utility functions.
- The header file<string.h> is required when using string manipulation functions.

Case Studies

1. Counting Words in a Text

One of the practical applications of string manipulations is counting the words in a text. We assume that a word is a sequence of any characters, except escape characters and blanks, and that two words are separated by one blank character. The algorithm for counting words is as follows:

1. Read a line of text.
2. Beginning from the first character n the line, look for a blank. If a blank is found, increment words by 1.
3. Continue steps 1 and 2 until the last line is completed.

The implementation of this algorithm is shown in Fig.8.13. The first while loop will be executed once for each line of text. The end of text is indicated by pressing the “Return” key an extra time after the entire text has been entered. The extra ”Return” key causes a newline character as input to the last line and as a result, the last line contains only the null character.

The program checks for this special line using the test

```
if(line[0] == '\0')
```

and if the first (and only the first) character in the line is a null character, then counting is terminated. Note the difference between a null character and a blank character.

Program

```
#include <stdio.h>

main ()
{
    char line[81]. Ctr;

    int i,c,

        end = 0;

        characters = 0,

        words = 0;

    printf("KEY IN THE TEXT. \n");

    printf("GIVE NONE SPACE AFTER EACH WORD.\n");

    printf("WHEN COMPLETED, PRESS 'RETURN'/\n\n");

    while ( end == 0)
    {

        /* Reading a line of text */

        c = 0;

        while ((str=getchar()) != '\n')
```

```

        line [c++] = ctr;

line[c] = '\0';

/* counting the words in a line */

if(line[0] == '\0')

    break;

else

{

    words++;

    for(i = 0; line[i] != '\0' i++)

        if(line[i] == ' ' || line[i] == '\t')

            words ++;

    }

/* counting lines and characters */

    lines = lines +1;

    characters = characters + strlen(line);

}

printf("\n");

printf("Number of lines = %d\n",lines);

printf("Number of words = %d\n",words);

printf("Number of characters = %d\n", characters);

}

```

Output

KEY IN THE TEXT.

GIVE ONE SPACE AFTER EACH WORD.

WHEN COMPLETED, PRESS, 'RETURN'

Admiration is a very short-lived passion.

Admiration involves a glorious obliquity of vision.

Always we like those who admire us but we do not

Like those whom we admire.

Fools admire, but men of sense approve.

Number of lines = 5

Number of words = 36

Number of characters = 205

Fig. 8.13 counting of characters, words and lines in a text

The program also counts the number of lines read and the total number of characters in the text. Remember, the last line containing the null string is not counted.

After the first while loop is exited, the program prints the results of counting

2. Processing of a Customer List

Telephone numbers of important customers are recorded as follows:

Full name	Telephone number
Joseph Louis Lagrange	869245
Jean Robert Argand	900823

Carl Freidrich Gauss

806788

It is desired to prepare a revised alphabetical list with surname (last name) first, followed by a comma and the initials of the first and middle names. For example,

Argand, J.R

We create a table of strings, each row representing the details of one person, such as first_name, middle_name, last_name, and telephone_number. The columns are interchanged as required and the list is sorted on the last_name. figure 8.14 shows a program to achieve this.

Program

```
#define CUSTOMERS 10

main ()
{
    char first_name[20][10], second_name[20][10],
        surname[20][10], name[20][10],
        telephone[20][10], dummy[20];

    int i,j;

    printf("Input names and telephone numbers \n");
    printf("?");
    for(i=0; i<CUSTOMERS; i++)
    {
```

```

        Scanf("%s %s %s", first_name[i], second_name[i],
        surname[i], telephone[i]);

/* converting full name to surname with initials */

strcpy(name[i], surname[i]);

strcat(name[i], ",");

dummy [0] = first_name[i][0];

dummy[1] = '\0';

strcat (name[i], dummy);

strcat(name[i], ",");

dummy[0] = second_name[i][0];

dummy[1] = '\0';

strcat(name[i], dummy);

}

/* Alphabetical ordering of surnames*/

for(i=1; I <= CUSTOMERS -1; i++)

    for(j = 1; j<= CUSTOMERS-I; j++)

        if(strcmp (name[j-1], name[j]) > 0)

        {

/* Swaping names */

            strcpy(dummy, name[j-1]);

            strcpy(name[j-1], name[j]);

```

```

        strcpy(name[j], dummy);

        /* Swaping telephone numbes */

        strcpy(dummy, telephone[j-1]);

        strcpy(telephone[j-1], telephone[j]);

        strcpy(telephone[j], dummy);
    }

    /* printing alphabetical list */

    printf("\nCUSTOMERS LIST IN ALPHABETICAL ORDER \n\n");

    for ( i =0; I < CUSTOMERS; i++)

        printf(" %-20s\t %-10s\n",name[i], telephone[i]);

    }

```

Output

Input names and telephone numbers

?Gottfried Wilhelm Leibni 711518

Joseph Louis Lagrange 869245

Jean Robet Argand 900823

Carl Freidrich Gauss 806788

Simon Denis Poisson 853240

Friedrich Wilhelm Bessiel 719731

Charles Francois Sturm 222031

George Gabriel Stokes 545454

Mohandas Karamchand Gandhi 362718

Josian Willard Gibbs 123145

CUSTOMERS LIST IN ALPHABETICAL ORDER

Argand, J, R	900823
Bessel, F,W	719731
Gandhi,M.K	362718
Gauss, C.F	806788
Lagrange,J.L	869245
Leibniz,G.W	711518
Poisson,S.D	853240
Stokes,G.G	545454
Sturm,C.F	22031

Fig 8.14 Program to alphabetize a customer list

9 USER-DEFINED FUNCTIONS

9.1 INTRODUCTION

We have mentioned earlier that one of the strengths of C language is C functions. They are easy to define and use. We have used functions in every program that we have discussed so far. However, they have been primarily limited to three functions, namely, **main**, **printf**, and **scanf**. In this chapter, we shall consider in detail the following:

- How a function is designed?
- How a function is integrated into a program?

- How two or more functions are put together? and
- How they communicate with one another?

C functions can be classified into two categories, namely, library functions and user-defined functions. `main` is an example of user-defined functions. **`printf` and `scanf`** belong to the category of library functions. We have also use other library functions such as **`sqrt`, `cos`, `strcat`**, etc. The main distinction between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library. In fact, this is one of the strengths of C language.

9.2 NEED FOR USER-DEFINED FUNCTIONS

As pointed out earlier, `main` is a specially recognized function in C. Every program must have a `main` function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only `main` function, it leads to a number of problems. The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These independently coded programs are called subprograms that are much easier to understand, and test. In C, such subprograms are referred to as ‘**functions**’.

There are times when certain types of operations or calculations are repeated at many points throughout a program. For instance, we might use the factorial of a number a several points in the program. In such situations, we may repeat the program statements wherever they are needed. Another approach is to design a function that can be called and used whenever required. This save both time and space.

This “division” approach clearly results in a number of advantages.

1. It facilitates top-down modular programming as shown in Fig.9.1. in this programming style the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.

2. The length of a source program can be reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be use by many other programs. This means that a C programmer can build on what others have already done, instead of starting all over again from scratch.

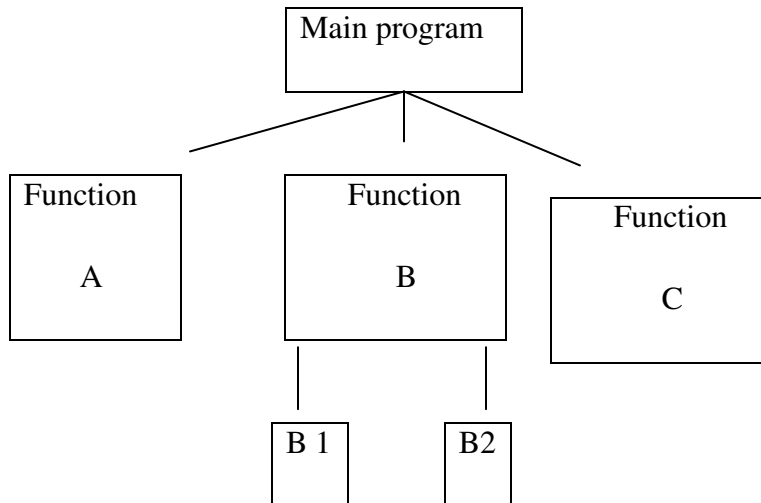


Fig. 9.1 Top down modular programming using functions

9.3 A MULTI-FUNCTION PROGRAM

A function is self-contained block of code of code that performs a particular task. Once a function has been designed and packed, it can be treated as a ‘black box’ that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. All that the program knows about a function is: what goes in and what comes out. Every C program can be designed using a collection of these black boxes known as *functions*.

Consider a set of statements as shown below:

```
void main( )  
  
    {  
  
        int i;
```

```

for(i=1; i<40; i++)

printf(".....");

printf("\n");

}

```

The above set of statements defines a function called **printline**, which could print a line of 390-character length. This function can be used in a program as follows:

```

void printline(void);          /*declaration*/

main()

{

printline( );

printf("This illustrates the use of c functions\n");

printline( );

}

void printline(void)

{

int i;

for(i=1; i<=40; i++)

printf("...");

printf("\n");

}

```

This program will print the following output:

.....
This illustrates the use of C functions
.....

The above program contains two user-defined functions:

main()function

printline()function

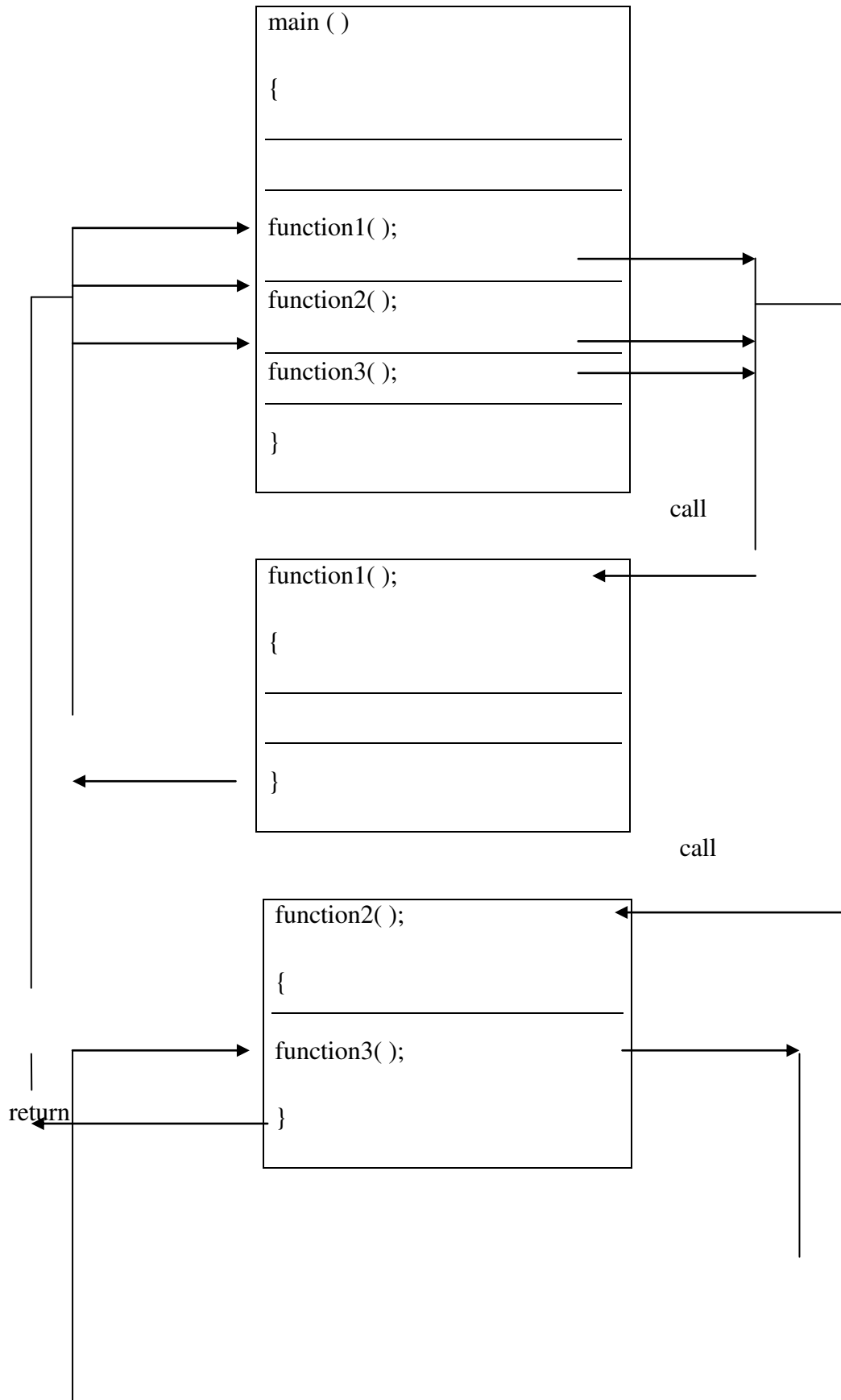
As we know, the program execution always begins with the main function. During execution of the main, the first statement encountered is

printline();

Which indicates that function **printline** is to be executed. At this point, the program control is transferred to the function **printline**. After executing the **printline** function, which outputs a line 39 characters length, the control is transferred back to the **main**. Now, the execution continues at the point where the function call was executed. After executing the **printf** statement, the control is again transferred to the **printline** function for printing the line once more.

The **main** function calls the user-defined **printline** function two times and the library function **printf** once. We may notice that the **printline** function itself calls the library function **printf** 39 times repeatedly.

Any function can call any other function. In fact, it can call itself. A ‘called function’ can also call another function. A function can be called more than once. In fact, this is one of the main features of using functions. Figure 9.2 illustrates the flow of control in a multi-function program. Except the starting point, there are no other predetermine relationships, rules of precedence, or hierarchies among the functions that make up a complete program. The functions can be placed in any order. A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end. See the box “modular programming”.



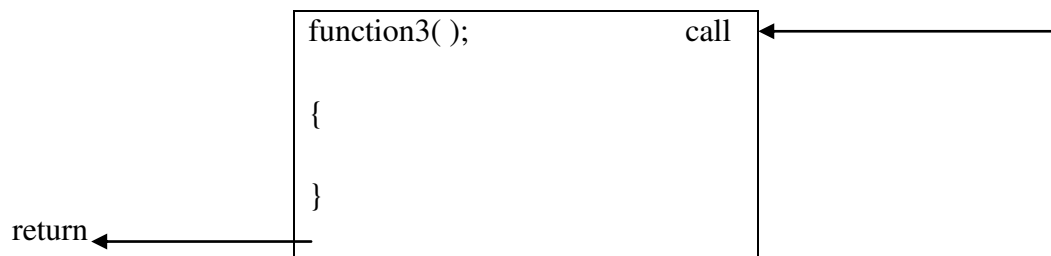


Fig. 9.2 flow of control in a multi-function program

Modular programming

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called modules that are separately named and individually called program units. These modules are carefully integrated to become a software system that satisfies the system requirements. It is basically a “divide-and-conquer” approach.

Modules are identified and designed such that they can be organized into a top-down hierarchical structure (similar to an organization chart). In C, each module refers to a function that is responsible for a single task.

Some characteristics of modular programming are:

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. a module can be called by one and only one higher module.
4. No communication can take place directly between modules that do not have calling-called relationship.
5. All modules are designed as single-entry, single-exit systems using control structures.

9.4 ELEMENTS OF USER-DEFINED FUNCTION

We have discussed and used a variety of data types and variables in our programs so far. However, declaration and use of these variables were primarily done inside the main function. As we mentioned in chapter 4, functions are classified as one of the derived data types in C. We can therefore define functions and use them like any other variables in C programs. It is

therefore not a surprise to note that there exists some similarities between functions and variables in C.

- Both function names and variable names are considered identifiers and therefore they must adhere to the rules for identifiers.
- Like variables, functions have types (such as *int*) associated with them.
- Like variables, function names and their types must be declared and defined before they are used in a program.

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition
2. Function call.
3. Function declaration.

The function definition is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required place in the program. This is known as the function call. The program (or a function) that calls the function is referred to as the calling program or calling function. The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as the function declaration or function prototypes.

9.5 DEFINITION OF FUNCTIONS

A function definition, also known as function implementation shall include the following elements:

1. function name;
2. function type;
3. list of parameters;
4. local variable declarations;
5. function statements; and
6. a return statement.

All the six elements are grouped into two parts, namely

- Function header (first three elements); and
- Function body (second three elements).

a general format of a function definition to implement these two parts is given below:

```
function_type function_name(parameter list)
{
    local variable declaration;

    executable statement1;

    executable statement2;

    .....
    .....

    return statement;
}
```

The first line **function_type function_name** (parameter list) is known as the function header and the statements within the opening and closing braces constitute the *function body*, which is a compound statement.

Function header

The function header consists of three parts: the function type (also known as *return type*), the function name and formal parameter list. Note that a semicolon is not used at the end of the function header.

Name and type

The *function type* specifies the type of value (*like float or double*) that the function is expected to return to the program calling the function. If the return type is not explicitly specified, C will assume that it is an integer type. If the function is not returning anything, then we need to

specify the return type as *void*. Remember, void is one of the fundamental data types in C. It is a good programming practice to code explicitly the return type, even when it is an integer. The value returned is the output produced by the function.

The *function name* is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function. However, care must be exercised to avoid duplicating library routine names or operating system commands.

Formal parameter list

The parameter list declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as *formal parameter*. These parameters can also be used to send values to the calling programs. This aspect will be covered later when we discuss more about functions. The parameters are also known as arguments.

The parameter list contains declaration of variables separated by commas and surrounded by parentheses. Examples:

```
float quadratic(int a, int b, int c){.....}
```

```
double power (double x, int n) {...}
```

```
float mul (float x, float y) {.....}
```

```
int sum (int a, int b) {...}
```

Remember, there is no semicolon after the closing parenthesis. Note that the declaration of parameter variables cannot be combined. That is, **int sum(int a,b)** is illegal.

A function need not always receive values from the calling program. In such cases, functions have no formal parameters. To indicate that the parameter list is empty, we use the keyword void between the parentheses as in

```
void printline(void)
```

```
{
```

```
.....  
.....  
}
```

This function neither receives any input values nor returns back any value. Many compilers accept an empty set of parentheses, without specifying anything as in

```
void printline ( )
```

But, it is a good programming style to use void to indicate a null parameter list.

Function body

The function body contains the declarations and statements necessary for performing the required task. The body enclosed in braces, contains three parts, in the order given below:

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. A return statement that return the value evaluated by the function.

If a function does not return any value (like the printline function), we can omit the return statement. However, note that its return type should be specified as void. Again, it is nice to have a return statement even for void functions.

Some examples of typical function definitions are:

```
(a) float mul (float x, float y)  
  
{  
  
float result;          /* local variable */  
  
result = x * y;        /* computes the product */  
  
return (result);      /* return the result*/  
  
}
```

```

(b) void sum (int a, int b)

{

printf (“sum = %s”, a + b); /* no local variables*/

return;                /* optional*/

}

```

```

(c ) void display (void)

{                                /* no local variables*/

printf(“ no type, no parameters”); /* no return statement*/

}

```

Note :

1. When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a void return.
2. A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.

9.6 RETURN VALUES AND THEIR TYPES

As pointed out earlier, a function may or may not send back any value to the calling function. If it does, it is done through the return statement. While it is possible to pass to the called function any number of values, the called function can only return one value per call, at the most.

The return statement can take one of the following forms:

return;

or

return(expression);

The first, the 'plain' return does not return any value; it acts much as the closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function. An example of the use of a simple return is as follows:

```
if(error);  
return;
```

Note:

In C99, if a function is specified as returning a value, the return must have value associated with it. The second form of return with an expression returns the value of the expression. For example, the function

```
int mul (int x, int y)
```

```
{  
  
    int p;  
  
    p = x*y;  
  
    return(p);  
  
}
```

returns the value of p which is the product of the values of x and y. The last two statements can be combined into one statement as follows:

```
return (x*y);
```

A function may have more than one **return** statement. This situation arises when the value returned is based on certain conditions. For example:

```
if( x<= 0 )  
  
return (0);  
  
else
```



```
return(1);
```

What type of data does a function return? All functions by default return **int** type data. But what happens if a function must return some other type? We can force a function to return a particular type of data by using a *type specifier* in the function header as discussed earlier.

When a value is returned, it is automatically cast to the functions' type. In functions that do computations using **doubles**, yet return **ints**, the returned value will be truncated to an integer. For instance, the function

```
int product (void)
```

```
{
```

```
return (2.5 * 3.0);
```

```
}
```

Will return the value 7, only the integer part of the result.

9.7 FUNCTION CALLS

A function can be called by simply using the function name followed by a list of actual parameters (or arguments), if any, enclosed in parentheses. Example:

```
main()
```

```
{
```

```
int y;
```

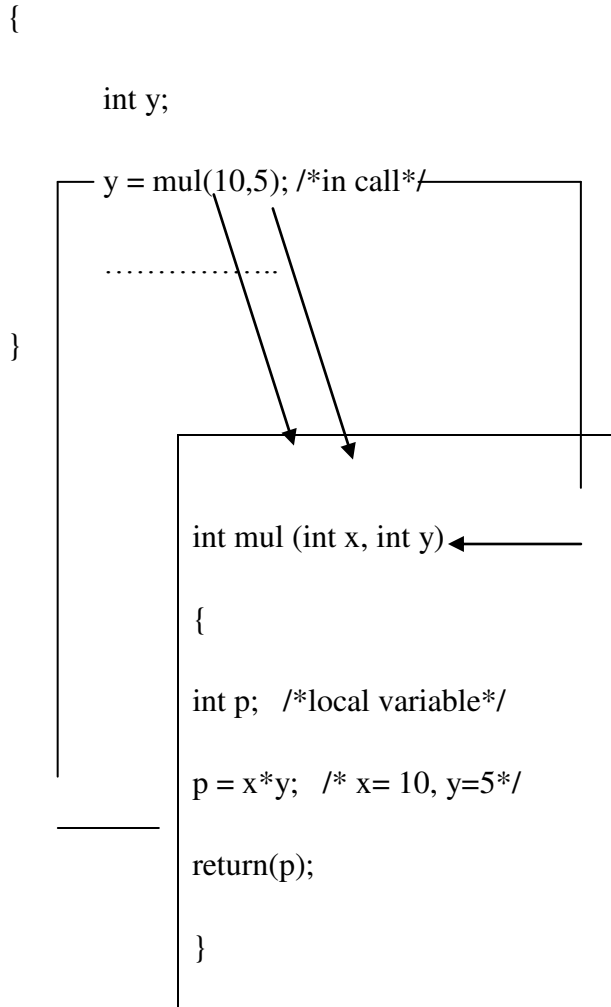
```
y = mul(10,5);    /*function call*/
```

```
printf("%d\n", y);
```

```
}
```

When the compiler encounters a function call, the control is transferred to the function `mul()`. This function is then executed line by line as described and a value is returned when a `return` statement is encountered. This value is assigned to `y`. this is illustrated below:

```
main( )
```



The function call sends two integer values 10 and 5 to the function.

int mul (int x, int y)

which are assigned to x and y respectively. The function computes the product x and y, assigns the result to the local variable p, and then returns the value 25 to the main where it is assigned to y again.

There are many different ways to call a function. Listed below are some of the ways the function mul can be invoked.

mul(10,5)

mul(m,5)

```
mul(10,n)
```

```
mul(m,n)
```

```
mul(m+5,10)
```

```
mul(10, mul(m,n))
```

```
mul(expression1, expression2)
```

Note that the sixth call uses its own call as its one of the parameters. When we use expressions, they should be evaluated to single values that can be passed as actual parameters.

A function which returns a value can be used in expression like any other variable. Each of the following statements is valid:

```
printf(“%d\n”, mul(p,q));
```

```
y = mul(p,q) / (p+q);
```

```
if (mul(m,n)>total) printf(“large”);
```

However, a function cannot be used on the right side of an assignment statement. For instance, `mul(a,b) = 15;`

is invalid.

A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function. The function `printline()` discussed in section 9.3 belongs to this category. Such functions may be called in by simply stating their names as independent statements.

```
main ( )
```

```
{
```

```
printline();
```

```
}
```

Note the presence of a semicolon at the end.

Function call

A function call is a postfix expression. The operator (. .) is at a very high level of precedence. (See table 3.8) therefore, when a function call is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.

In a function call, the function name is the operand and the parentheses set (. .) which contains the actual parameters is the operator. The actual parameters must match the function's formal parameters in type, order and number. Multiple actual parameters must be separated by commas.

Note:

1. If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
2. On the other hand, if the actual are less than the formals, the unmatched formal arguments will be initialized to some garbage.
3. Any mismatch in data types may also result in some garbage values.

9.8 FUNCTION DECLARATION

Like variables, all functions in a C program must be declared, before they are invoked. A function declaration (also known as function prototype) consists of four parts.

- Function type(return type)
- Function name.
- Parameter list.
- Terminating semicolon.

They are coded in the following format:

Function-type function-name(parameter list);

This is very similar to the function header line except the terminating semicolon. For example, mul function defined in the previous section will be declared as:

- `Int mul(int m, int n); /* Function prototype*/`

Points to note

- 1) The parameter list must be separated by commas.
- 2) The parameter names do not need to be the same in prototype declaration and the function definition.
- 3) The types must match the types of parameters in the function definition, in number and order.
- 4) Use of parameter names in the declaration is optional.
- 5) If the function has no formal parameters, the list is written as(void).
- 6) The return type is optional, when the function returns int type data.
- 7) The retype must be void if no value is returned.
- 8) When the declared types do not match with the types in the function definition, compiler will produce an error.

Equally acceptable forms of declaration of mul function are:

```
int mul (int, int);
```

```
mul (int a, int b);
```

```
mul (int, int);
```

When a function does not take any parameters and does not return any value, its prototype is written as:

```
void display (void);
```

A prototype declaration may be placed in two places in a program.

1. Above all the functions (including main).
2. Inside a function definition.

When we place the declaration above all the function (in the global declaration section), the prototype is referred to as a *global prototype*. Such declarations are available for all the function in the program.

When we place it in a function definition (in the local declaration section), the prototype is called a local prototype. Such declarations are primarily used by the functions containing them.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the scope of the function. (Scope is discussed later in this chapter.) It is a good programming style to declare prototypes in the global declaration section before main. It adds flexibility, provides an excellent quick reference to the functions used in the program, and enhances documentation.

Prototypes: yes or no

Prototype declaration is not essential. If a function has not been declared before it is used, C will assume that its details are available at the time of linking. Since the prototype is not available, C will assume that the return type is an integer and that the types of parameters match the formal definitions. If these assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations, preferably in the global declaration section.

Parameters Everywhere!

Parameters (also known as arguments) are used in three places:

1. in declaration(prototypes)
2. in function call, and
3. in function definition.

The parameters used in prototypes and function definitions are called formal parameters and those used in function calls are called actual parameters. Actual parameters used in a calling statement may be simple constants, variables or expressions.

The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

9.9 CATEGORY OF FUNCTIONS

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

Category 1: functions with no arguments and no return values.

Category 2: functions with arguments and no return value.

Category 3: functions with arguments and one return value.

Category 4: functions with no arguments and but a return value.

Category 5: functions that return multiple values.

In section to follow, we shall discuss these categories with examples. Not that, from now on, we shall use the term arguments (rather than parameters) more frequently.

9.10 NO ARGUMENTS AND NO RETURN VALUES

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function. This is depicted in Fig.9.3. The dotted indicate that there is only a transfer of control but not data.

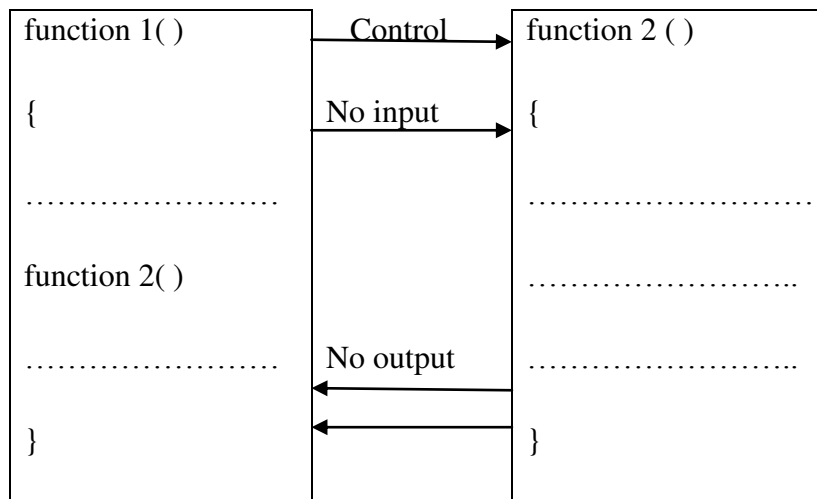


Fig.9.3. No data communication between function.

As pointed out earlier, a function that does not return any value cannot be used in an expression. It can only be used as an independent statement.

Program9.1 Write a program with multiple functions that do not communicate any data between them.

A program with three user-defined functions is given in Fig.9.4. `main` is the calling function that calls `printline` and `value` functions. Since both the called functions contain no arguments, there are no argument declarations. The `printline` function, when encountered, prints a line with a length of 35 characters as prescribed in the function. The `value` function calculates the value of the principal amount after certain period of years and prints the results. The following equation is evaluated repeatedly:

$$\text{value} = \text{principal}(1 + \text{interest-rate})$$

program

```
/* function declaration*/
```

```
void printline (void);
```

```
void value( void);
```

```
main ( )
```

```
{
```

```
printline();
```

```
value ();
```

```
printline( );
```

```
}
```

```
/* function: printline( )*/
```

```
void printline(void) /*contains no arguments*/
```



```

{
int i;

for(i=1; i <= 35; i++)

printf("%c", '_');

printf("\n");

}

/* function2: value()*/

void value(void) /*contains no arguments*/

{

int year, period;

float inrate, sum, principal;

printf("principal amount?");

scanf("%f", &principal);

printf("interest rate? ");

scanf("%f", &inrate);

printf("period? ");

scanf("%d", &period);

sum = principal;

year = 1;

while(year <= period)

{

```

```
sum = sum * (1+inrate);  
  
year = year +1;  
  
}  
  
printf("\n%8.2f  %5d  %12.2f\n", principal, inrate, period,sum);  
  
}
```

Output

principal amount? 5000

interest rate? 0.12

period? 5

5000.0 0.12 5 8811.71

Fig.9.4 Function with no arguments and no return values

It is important to note that the function value receives its data directly from the terminal. The input data include principal amount, interest rate and the period for which the final value is to be calculated. The while loop calculates the final value and the results are printed by the library function printf. When the closing brace of value () is reached, the control is transferred back to the calling function main. Since everything is done by the value itself there is in fact nothing left to be sent back to the called function. Return types of both printline and value are declared as void.

Note that no return statement is employed. When there is nothing to be returned, the return statement is optional. The closing brace of the function signals the end of the execution of the function, thus returning the control, back to the calling function.

9.11 ARGUMENTS BUT NO RETURN VALUES

In Fig.9.4 the main function has no control over the way the functions receive input data. For example, the function `printline` will print the same line each time it is called. Same is the case with the function `value`. We could make the calling function to read data from the terminal and pass it on to the called function. This approach seems to be wiser because the calling function can check for the validity of data, if necessary, before it is handed over to the called function.

The nature of data communication between the calling function and the called function with arguments but no return values is shown in Fig.9.5

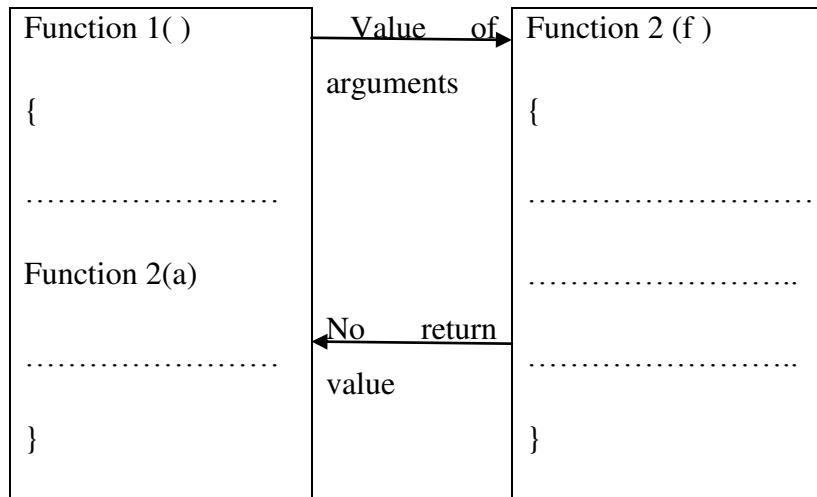


Fig.9.5 One-way data communication

We shall modify the definition of both the called functions to include arguments as follows:

```
void printline(char ch)
```

```
void value(float p, float r, int n)
```

The arguments `ch`, `p`, `r` and `n` are called the formal arguments. The calling function can now send values to these arguments using function calls containing appropriate arguments. For example, the function call

```
value(500.0,12,5)
```

Would send the values 500,0.12,and 5 to the function

void value(float p, float r, int n)

and assign 500 to p, 0.12 to r and 5 to n. the values 500,0.12, and 5 are the actual arguments, which become the values of the formal arguments inside the called function.

The actual and formal arguments should match in number, type, and orgfer. The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument as shown in Fig.9.6

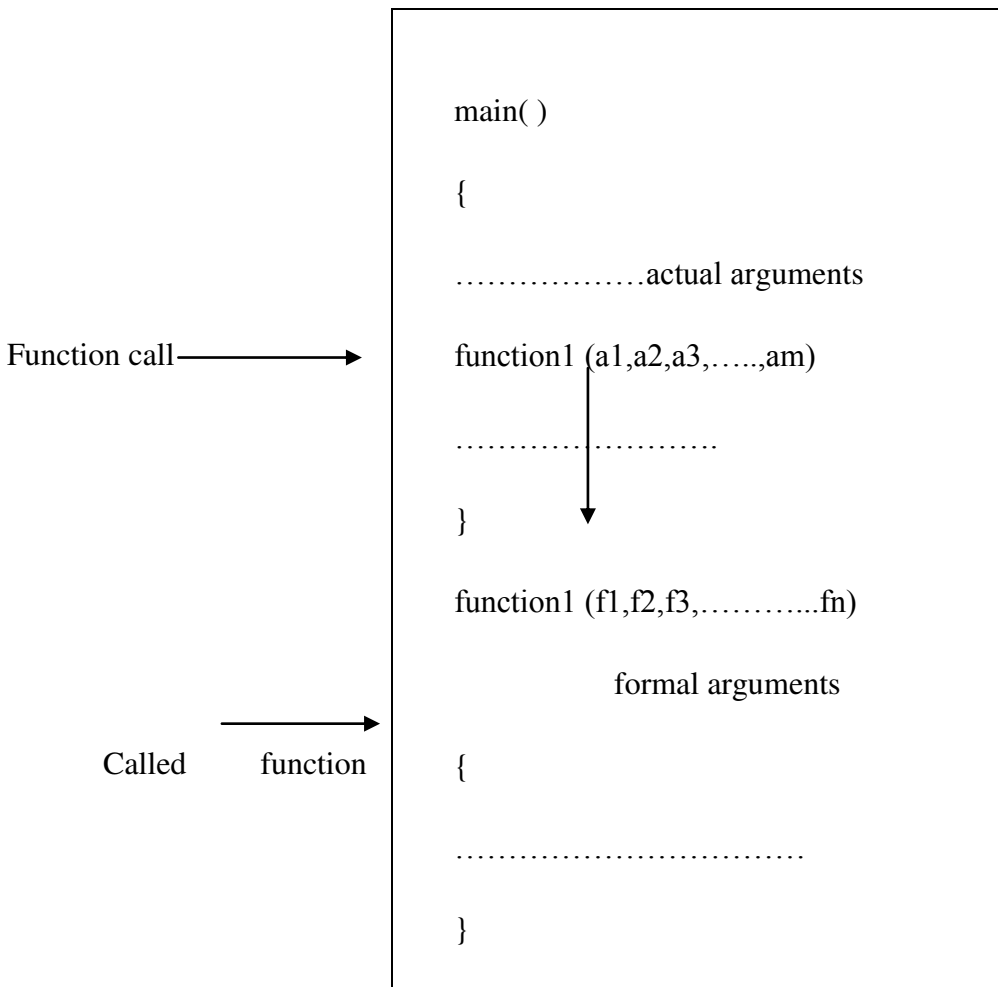


Fig.9.6 Arguments matching between the function call and called function

We should ensure that function call has matching arguments. In case, the actual arguments are more than the formal arguments, ($m > n$), the extra actual arguments are discarded. On the other hand, if the actual argument are less than the formal arguments. The unmatched formal arguments are initialized to some garbage values. Any mismatch in data type may also result in passing of garbage values. Remember, no error message will be generated.

While the formal arguments must be valid variable names, the actual arguments may be variable names, expressions, or constants. The variables used in actual arguments must be assigned values before the function call is made.

Remember that, when a function call is made, only a copy of the values of actual arguments is passed into the *called function*. What occurs inside the function will have no effect on the variables used in the actual argument list.

Program 9.2 Modify the program of program 9.1 to include the arguments in the function calls

The modified program with function arguments is presented in Fig.9.7. Most of the program is identical to the program in Fig.9.4. The input prompt and scanf assignment statement have been moved from value function to main. The variables principal, inrate, and period are declared in main because they are used in main to receive data. The function call

value(principal, inrate, period);

Passes information it contains to the function value.

The function header of value has three formal arguments **p,r** and **n** which correspond to the actual arguments in the function call, namely, **principal, inrate, and period**. On execution of the function call, the values of the actual arguments are assigned to the corresponding formal arguments. In fact, the following assignments are accomplished across the function boundaries:

p=principal;

r=inrate;

n= period;

Program

```

/*prototype*/

void printline(char c);

void value(float, float, int);

main( )

{

float principal, inrate;

int period;

printf("Enter principal amount, interest");

printf("rate, and period\n");

scanf("%f %f %d", &principal, &inrate, &period);

printline('z');

value(principal,inrate,period);

printline('c');

}

void printline(char ch)

{

int i;

for(i=1; i <= 52; i++)

printf("%c", ch);

printf("\n");

}

```

```

void value (float p, float r, int n)
{
int year;

float sum;

sum = p;

year = 1;

while(year <= n)
{
sum = sum * (1+r);

year = year +1;

}

printf(“%f\t%f\t%d\t%f\n”,p,r,n,sum);

}

```

Output

```

Enter principal amount, interest rate, and period

5000 0.12 5

////////////////////////////////////

5000.000000 0.120000 5 8811.708984

cccccccccccccccccccccccccccccccccccccccccccccccccccccccc

```

Fig 9.7 Functions with arguments but no return values

The variables declared inside a function are known as local variables and therefore their values are local to the function and cannot be accessed by any other function. We shall discuss more about this later in the chapter.

The function value calculates the final amount for a given period and prints the results as before. Control is transferred back on reaching the closing brace of the function. Note that the function does not return any value.

The function **printline** is called twice. The first call passes the character 'z', while the second passes the character 'C' to the function. These are assigned to the formal argument **ch** for printing lines (see the output).

Variable number of arguments

Some functions have a variable number of arguments and data types which cannot be known at compile time. The printf and scanf functions are typical examples. The ANSI standard proposes new symbol called the ellipsis to handle such function. The ellipsis consists of three periods (..) and used as shown below:

double area(float d,...)

Both the function declaration and definition should use ellipsis to indicate that the arguments are arbitrary both in number and types.

9.12 ARGUMENTS WITH RETURN VALUES

The function value in Fig.9.7 receives data from the calling function through arguments, but does not send back any value. Rather, it displays the results of calculations at the terminal. However, we may not always wish to have the result of a function displayed. We may use it in the calling function for further processing. Moreover, to assure a high degree of portability between programs, a function should generally be coded without involving any I/O, for example, different programs may require different output formats for display of results. These shortcomings can be overcome by handing over the result of a function to its calling where the returned value can be used as required by the program.

A self-contained and independent function should behave like a ‘black box’ that receives a predefined form of input and outputs desired values. Such function will have two-way data communication as shown in Fig.9.8.

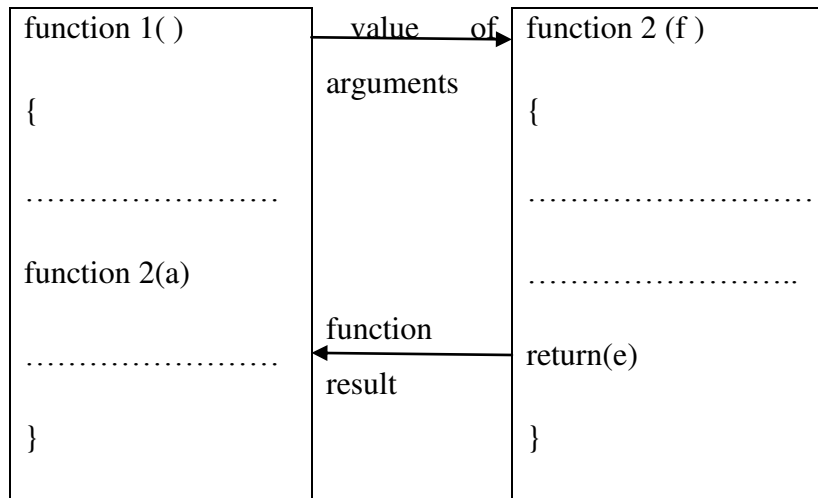


Fig.9.8 Two-way communication between functions

We shall modify the program in fig 9.7 illustrate the use of two-way data communication between the calling and the called functions.

Program9.3 In the program presented in Fig.9.7 modify function value, to return the final amount calculated to the main, which will display the required output at the terminal. Also extend the versatility of the function printline by having it to take the length of the line as an argument.

The modified program with the proposed changes is presented in Fig.9.9. one major change is the movement of the printf statement from value to main.

Program

```

void printline(char ch, int len);

value(float, float, int);

main( )

```

```

{
float principal, inrate, amount;

int period;

printf("Enter principal amount, interest");

printf("rate, and period\n");

scanf("%f %f %d, &principal, &inrate, &period);

printline('*', 52);

amount = value(principal,inrate, period);

printf("\n%f\t%f\t%d\t%f\n\n", principal, inrate, period, amount);

printline('=',52);

}

void printline(char ch, int len)

{

int i;

for(i=1;i<=len;i++)

printf("%c",ch);

printf("\n");

}

value(float p, float r, int n)/*default return type*/

{

int year;

```

```

float sum;

sum = p; year =1;

while(year <= n)

{

sum = sum * (1+r);

year = year +1;

}

return(sum); /      *returns int part of sum*/

}

```

Output

Enter principal amount, interest rate, and period

5000 0.12 5

.....

5000.000000 0.120000 5 8811.000000

Fig .9.9 Function with arguments and return values

The calculated value is passed on to main through statement;

return(sum);

Since, by default, the return type of value function is int, the integer value of sum at this point is returned to main and assigned to the variable amount by the functional call

amount = value(principal, inrate, period);

the following events occur, in order, when the above function call is executed:

1. The function call transfers the control along with copies of the values of the actual arguments to the function value where the formal arguments p,r, and n are assigned the actual values of principal, inrate and period respectively.
2. The called function value is executed line by line in a normal fashion until the return(sum); statement is encountered. At this point, the integer value of sum is passed back to the function-call in the main and the following indirect assignment occurs:

value(principal, inrate, period) = sum;

3. The calling statement is executed normally and the returned value is thus assigned to amount, a float variable.
4. Since amount is a float variable, the returned integer part of sum is converted to floating-point value. See the output.

Another important change is the inclusion of second argument to printline function to receive the value of length of the line from the calling function. Thus, the function call

printline('*', 52);

Will transfer the control to the function printline and assign the following values to formal arguments ch and len;

ch = '*';

len = 52;

Returning float values

We mentioned earlier that a C function returns a value of the type int as the default case when no other type is specified explicitly. For example, the function value of program 9.3 does all calculations using floats but the return statement

return(sum);

returns only the integer part of sum. This due to the absence of the type-specifier in the function header. In this case, we can accept the integer value of sum because the truncated decimal part is insignificant compared to the integer part. However, there will be times when we may find it

necessary to receive the float or double type of data. For example, a function that calculates the mean or standard deviation of a set of values should return the function value in either float or double.

In all such cases, we must explicitly specify the return type in both the function definition and the prototype declaration.

If we have a mismatch between the type of data that the called functions returns and the type of data that the calling function expects, we will have unpredictable results. We must, therefore, be very careful to make sure that both types are compatible.

Program9.4 The program in fig.9.10 shows how to write a C program (float x[], int n) that returns the position of the first minimum value among the first n elements of the given array x.

Program

```
#include<stdio.h>

#include<conio.h>

#include<stdio.h>

int minpos(float [], int);

void main( )

{

    int n;

    float x[10] = { 12.5, 3.0, 45.1, 8.2, 19.3, 1.0, 7.8, 23.7, 29.9, 5.2};

    printf("Enter the value of n:");

    scanf("%d", &n);

    if(n>=1 && n<=10)

        :
```

```

else
{
    printf("invalid value of n.....press any key to terminate the
    program....");
    getch();
    exit(0);
}

printf("within the first %d elements of array, the first minimum value is
stored at index %d", n, minpos(x,n));

getch();
}

int minpos(float a[],int n)
{
    int i,index;
    float min=9999.99;
    for(i=0;i<n;i++)
if(a[i]<min)
    {
        min=a[i];
        index = i;
    }
return(index);

```

```
}
```

Output

Enter the value of n: 5

Within the first 5 elements of array, the first minimum value is stored at index 1

Fig.9.10 Program to return the position of the first minimum value in an array

Program9.5 write a function power that computes x rose to the power y for integers x and y and returns double-type value.

Fig.9.11 shows a power function that returns a double, the prototype declaration

```
double power(int, int);
```

appears in main, before power is called.

Program

```
main( )  
  
{  
  
int x,y;  
  
double power(int, int);    /*prototype declaration*/  
  
printf("enter x,y:");  
  
scanf("%d %d", &x,&y);  
  
printf("%d to power %d is %f\n", x,y,power(x,y));  
  
}  
  
double power(int x, int y);  
  
{
```

```

double p;

p = 1.0; /*x to power zero*/

if(y>=0)

while(y--) /*computes positive powers */

p *= x;

else

while (y++) /*computes negative powers*/

p /= x;

return(p);    /*returns double type*/

}

```

Output

Enter x,y: 16 2

16 to power to 2 is 256.000000

Enter x,y: 16 -2

16 to power -2 is 0.003906

Fig. 9.11 Power functions illustration return of float values

Another way to guarantee that power's type is declared before it is called in main is to define the power function before we define main. Power's type is then known from its definition. So we no longer need its type declaration in main.

9.13 NO ARGUMENTS BUT RETURNS A VALUE

There could be occasions where we may need to design function that may not take any arguments but returns a value to the calling function. A typical example is the getchar function declared in the header file<stdio.h> we have used this function earlier in a number of places.

The `getchar` function has no parameters but it returns an integer type data that represents a character.

We can design similar functions and use in our programs. Example:

```
int get_number(void);
```

```
main
```

```
{
```

```
int m = get_number( );
```

```
printf(“%d”, m);
```

```
}
```

```
int get_number(void)
```

```
{
```

```
int number;
```

```
scanf(“%d”, &number);
```

```
return(number);
```

```
}
```

9.14 FUNCTION THAT RETURNS MULTIPLE VALUES

Up till now, we have illustrated functions that return just one value using a return statement. That is because; a return statement can return only one value. Suppose, however, that we want to get more information from a function. We can achieve this in C using arguments not only to receive information but also to send back information to the calling function. The arguments that are used to “send out” information are *output parameters*.

The mechanism of sending back information through arguments is achieved using what are known as the address operator (&) and indirection operator (*). Let us consider an example to illustrate this.

```
void mathoperation(int x, int y, int *s,int *d);
```

```
main()
{
    int x = 20, y = 10, s, d;
    mathoperation(x,y, &s, &d);
    printf("s=%d\n", s,d);
}
```

```
void mathoperation (int a, int b, int *sum, int *diff);
{
    *sum = a+b;
    *diff = a-b;
}
```

The actual arguments **x** and **y** are input arguments, **s** and **d** are output arguments. In the function call, while we pass the actual values of **x** and **y** to the function, we pass the addresses of locations where the values of **s** and **d** are stored in the memory. (That is why the operator & is called the address operator) when the function is called the following assignments occur.

value of x to a

value of y to b

address of s to sum

address of d to diff

Note that indirection operator `*` in the declaration of **sum** and **diff** in the header indicates these variables are to store addresses, not actual values of variables. Now the variables **sum** and **diff** point to the memory locations of **s** and **d** respectively.

(The operator `*` is known as indirection operator because it gives an indirect reference a variable through its address.)

In the body of the function, we have two statements;

```
*sum = a+b;
```

```
*diff = a-b;
```

The first one adds the values **a** and **b** and the result is stored in the memory location pointed to by **sum**. Remember, this memory location is the same as the memory location of **s**. Therefore, the value is stored in the location pointed to by **sum** is the value of **s**.

Similarly, the value of **a-b** is store in the location pointed to by **diff**, which is the same as the location **d**. After the function call is implemented, the value of **s** is **a+b** and the value of **d** is **a-b**. Out will be:

```
s = 30;
```

```
d = 10;
```

The variables ***sum** and ***diff** are known as pointers and **sum** and **diff** as pointer variables. Since they are declared as **int**, they can point to locations of **int** type data.

The use of pointer variables as actual parameters for communicating data between function is called “pass by pointers” or “call by address or reference”. Pointers and their applications are discussed in detail in chapter 11.

Rules for pass by pointers

1. The types of the actual and formal arguments must be same.

2. The actual arguments (in the function call) must be addresses of variables that are local to the calling function.
3. The formal arguments in the function header must be prefixed by the indirection operator*.
4. In the prototype, the arguments must be prefixed by the symbol *.
5. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed with the indirection operator *.

9.15 NESTING OF FUNCTIONS

C PERMITS NESTING OF functions freely. **main** can call **funcion1**, which calls **function2**, which calls **function3**,and so one. There is in principle no limit as to how deeply functions can be nested.

Consider the following program:

```
float ratio (int x, int y, int z);
```

```
int difference (int x, int y);
```

```
main( )
```

```
{
```

```
int a,b,c;
```

```
scanf(“%d %d %”, &a,&b, &c);
```

```
printf(“%f \n”, ratio(a,b,c));
```

```
}
```

```
float ratio (int x, int y, int z)
```

```
{
```

```
if(difference(y,z))
```

```
return(x/(y-z));
```

```

        else

        return(0.0);

    }

int difference(int p, int q)

{

if(p != q)

return(1);

else

return(0);

}

```

The above program calculates the ratio

$a/b-c$

and prints the result. We have the following three functions:

main()

ratio()

difference()

main reads the value of a, b, and c calls the function **ratio** to calculate the value $a/(b-c)$. This ratio cannot be evaluated if $(b-c) = 0$. Therefore, ratio calls another function difference to test whether the difference $(b-c)$ is zero or no; difference returns 1, if b is not equal to c;

Otherwise returns zero to the function ratio. In turn, ratio calculates the value $a/(b-c)$ if it receives 1 and returns the result in float. In case, ratio receives zero from difference, it sends back 0.0 to main indicating that $(b-c) = 0$.

Nesting of function calls is also possible. For example, statement likes

```
p= mul(mul(5,2),6);
```

is valid. This represents two sequential function calls. The inner function call is evaluated first and the returned value is again used as an actual argument in the outer function call. If **mul** returns the product of its arguments, then the value of p would be 60(=5*2*6).

Note that the nesting does not mean defining one function within another. Doing this is illegal.

9.16 RECURSION

When a called function in turn calls another function a process of ‘chaining’ occurs. Recursion is a special case of this process, where a function calls itself. A very simple example of recursion is presented below:

```
main()
{
    printf(“this is an example of recursion\n”)
    main();
}
```

When executed, this program will produce an output something like this:

This is an example of recursion

This is an example of recursion

This is an example of recursion

This is an ex

Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Another useful example of recursion is the evaluation of factorials of a given number. The factorial of a number n is expressed as a series of repetitive multiplications as shown below:

factorial of n = $n(n-1)(n-2)\dots\dots\dots 1$

For example, factorial of 4 = $4*3*2*1=24$

A function to evaluate factorial of n is as follows:

```
int factorial (int n)
```

```
{  
  
    int fact;  
  
    if(n==1)  
  
        return(1);  
  
    else  
  
        fact = n*factorial(n-1);  
  
        return(fact);  
  
}
```

Let us see how the recursion works. Assume n= 3 since the value of n is not 1, the statement

$fact = n * factorial(n-1);$

will be executed with n = 3 that is

$fact = 3*factorial(2);$

will be evaluated. The expression on the right-hand side includes a call o factorial with n =2

This call will return the following value:

$2*factorial(1)$

Once again. Factorial is called with n = 1. This time, the function returns 1. The sequence of operations can be summarized as follows:

$Fact =3*factorial(2)$

= 3*2*factorial(1)

=3*2*1

=6

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an **if** statement somewhere to force the function to return without the recursive call being executed. Otherwise the function will never return.

9.17 PASSING ARRAYS TO FUNCTIONS

ONE-DIMENSIONAL ARRAYS

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one-dimensional array to a called function, it is sufficient to list the name to the array, without any subscripts, and the size of the array as arguments. For example, the call

largest(a,n)

Will pass the whole array a to the called function. The called function expecting this call must be appropriately defined. The largest function header might look like:

float largest(float array[], int size)

The function largest is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

float array[];

The pair of brackets informs the compiler that the argument array is an array of numbers. It is not necessary to specify the size of the array here.

Let us consider a problem of finding the largest value in an array of elements. The program is as follows:

main()


```

{
    float largest (float a[ ],int n);

    float value[4] = {2.5,-4.75,1.2,3.67};

    printf(“%f\n”, largest(value,4));
}

float largest(float a[], int n)
{
    int i;

    float max;

    max = a[0];

    for(i =1; i < n; i++)

    if(max < a[i])

    max = a[i];

    return(max);
}

```

When the function call largest (value 4) is made, the values of all elements of array value become the corresponding elements of array a in the called function. The largest function finds the largest value in the array and returns the result to the main.

In C, the name of the array represents the address of its first element. By passing the array name, we are in fact, passing the address of the array to the called function. The array in the called function now refers to the same array store in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the function is referred to as pass by address (or pass by pointers). Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

Program 9.6 Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use function to calculate standard deviation and mean.

Standard deviation of a set of n values is given by

$$S.D = \sqrt{\frac{1}{n} \sum_{i=1}^n (\bar{x} - x_i)^2}$$

where \bar{x} is the mean of the values.

Program

```
#include<math.h>

#define size 5

float std_dev(float a[], int n);

float mean(float a[], int n);

main()
{
    float value[size];
    int i;
    printf("enter %d float values\n", size);
    for(i=0; i < size; i++)
        scanf("%f", &value[i]);
    printf("std.deviation is %f\n", std_dev(value, size));
}
```

```

}

float std_dev(float a[], int n)
{
    int i;

    float x, sum = 0.0;

    x = mean(a,n);

    for(i=0; i < n; i++);

    sum += (x-a[i]);

    return(sqrt(sum/(float)n));

}

float mean(float a[], int n)
{
    int i;

    float sum = 0.0;

    for(i=0; i<n. i++)

    sum = sum + a[i];

    return(sum/(float)n);

}

```

Output

Enter 5 float values

35.0 67.0 79.5 14.20 55.75

Std.deviation is 23.231582

Fig.9.12 passing of arrays to a function

A multifunction program consisting of **main**, **std_dev**, and **mean** function is shown in fig.9.12. **main** reads the elements of the array value from the terminal and calls the function **std_dev** to print the standard deviation of the array elements. **std_dev**, in turn, calls another function **mean** to supply the average value of the array elements.

Both **std_dev** and **mean** are defined as floats and therefore they are declared as floats in the global section of the program.

Three rules to pass an array to function

1. The function must be called by passing only the name of the array.
2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
3. The function prototype must show that the argument is an array.

When dealing with array arguments, we should remember one major distinction. If a function changes the values of the elements of an array, then these changes will be made too the original array that passed to the function. When an entire array is passed as an argument, the contents of the array are not copied into the formal parameter array; instead, information about the addresses of array elements is passed on to the function. Therefore, any changes introduced to the array elements are truly reflected in the original array in the calling function. However, this does not apply when an individual element is passed on as argument. Program 9.6 highlights these concepts.

Program 9.7 writes a program that uses a function to sort an array of integers.

A program to sort an array of integers using the function `sort ()` is given in fig.9.13. its output clearly shows that a function can change the values in an array passed as an argument.

Program

```
void sort(int m, int x[ ]);
```

```

main()
{
    int i;

    int marks[5] = {40, 90,73,81, 35};

    printf("Marks before sorting\n");

    for(i = 0; i < 5; i++)

    printf("%d", marks[i]);

    printf("\n\n");

    sort(5, marks);

    printf("marks after sorting\n");

    for(i = 0; i < 5; i++)

    printf("%4d", marks[i]);

    printf("\n");
}

void sort(int m, int x[])
{
    int i, j, t;

    for(i = 1; i <= m-1; i++)

    for(j = 1; j <= m-i; j++)

    if(x[j-1] >= x[j])
    {

```

```

        t = x[j-1];

        x[j-1] =x[i];

        x[j] = t;

    }

}

```

Output

Marks before sorting

40 90 73 81 35

Marks after sorting

35 40 73 81 90

Fig 9.13 sorting of array elements using a function

Two dimensional arrays

Like simple array, we can also pass multi-dimensional arrays to functions. The approach is similar to the one we did with one-dimensional arrays. The rules are simple.

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function headers.

The function given below calculates the average of the values in a two-dimensional matrix.

```
double average(int x[][N], int M, int N)
```

```

{
    int i,j;

```

```

double sum = 0.0;

for(i=0; i<M; i++)

for(j=1; j<N; j++)

sum+=x[i][j];

return(sum/(M*N));

}

```

This function can be used in a main function as illustrated below:

```

main()

{

int M=3, N=2;

double average(int [] [N], int , int);

double mean;

int matrix [M][N]=

{

{1,2},{3,4},{5,6}

};

mean = average(matrix, M, N);

.....

.....}

```

9.18 PASSING STRING TO FUNCTIONS

The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is define. Example:

```
void display(char item_name[])
```

```
{
```

```
.....
```

```
}
```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[ ]);
```

3. A call to the function must have a string array name without subscripts as its actual argument. Example

```
display(name);
```

where name is a properly declared string array in the calling function. we must not here that, like arrays, strings in C cannot be passed by value to functions.

Pass by value versus pass by pointers

The technique used to pass data from one function to another is known as parameter passing. Parameter passing can be done in two ways.

- Pass by value (also known as call by value).
- Pass by pointers(also known as pointers)

In pass by value, values of actual parameters are copied to the variable in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters. This ensures that the original data in the calling function cannot be changed accidentally.

In pass by pointers (also known as pass by address), the memory addresses of the variables rather than the copies of values are sent to the called function. In this case, the called function directly works on the data in the calling function and changed values are available in the calling function for its use.

Pass by pointers method is often used when manipulating arrays and strings. This method is also used when we require multiple values to be returned by the calling function.

9.19 THE SCOPE, VISIBILITY AND LIFETIME OF VARIABLES

Variables in c differ in behaviour from those in most other languages. For example, IN program, a variable retains its value throughout the program. It is not always the case in C. it all depends on the ‘storage’ class a variable may assume.

In C not only do all variables have data type, they also have a storage class. The following variable storage classes are most relevant to functions:

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

We shall briefly discuss the scope, visibility and longevity of each of the above class of variables. The scope of variable determines over what region of the program a variable is actually available for use (‘active’). Longevity refers to the period during which a variable retains a given value during execution of a program (‘alive’). So longevity has a direct effect on the utility of a given variable. The visibility refers to the accessibility of a variable from the memory.

The variables may also be broadly categorized, depending on the place of their declaration, as internal (local) or external (global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

It is very important to understand the concept of storage classes and their utility in order to develop efficient multifunction programs.

Automatic variables

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as local or internal variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable. For instance, the storage class of the variable number in the example below is automatic.

```
main()  
  
    {  
  
    int number;  
  
    .....  
  
    .....  
  
    }
```

We may also use the keyword auto to declare automatic variables explicitly.

```
main()  
  
    {  
  
    auto int number;  
  
    .....  
  
    .....  
  
    }
```

One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program. This assures that we may

declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

Program 9.8 Write a multifunction to illustrate how automatic variables work.

A program with two subprograms **function1** and **function2** is shown in Fig.9.14. *m* is an automatic variable and it is declared at the beginning of each function. *M* is initialized to 10, 100, and 1000 in **function1**, **function2**, and **main** respectively.

When executed **main** calls **function2** which in turn calls **function1**. When **main** is active, *m*=1000; but when **function2** is called, the **main**'s *m* is temporarily put on the shelf and the new local *m*= 100 becomes active. Similarly, when **function1** is called, both previous values of *m* are put on the shelf and the latest value of *m*(=10) becomes active. As soon as **function1** (*m*=10) is finished, **function2** (*m*=100) takes over again. As soon it done, **main** (*m*=100) takes over. The output clearly shows that the value assigned to *m* in one function does not affect its value in the other functions; and the local value of *m* is destroyed when it leaves a function.

Program

```
void function1(void);

void function2(void;

main()
{
    int m =1000;

    function2();

    printf("%d\n", m); /* third output*/
}

void function1(void)
{
```

```

        int m= 10;

        printf(“%d\n”, m); /*first output*/

    }

void function2(void)

{

    int m = 100;

    function();

    printf(“%d\n”, m); /*second output*/

}

```

Output

10

100

1000

Fig .9.14 working of automatic variables

There are two consequences of the scope and longevity of auto variables worth remembering. First, any variable local to main will be normally alive throughout the whole program, although it is active only in main. Secondly, during recursion, the nested variables are unique auto variables, a situation similar to function-nested auto variables with identical names.

External variables

Variable that are both alive and active throughout the entire program are known as external variables. They are also known as global variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function. For example, the external declaration of integer number and float length might appear as:

```
int number;
```

```
float length = 7.5;
```

```
main()
```

```
{
```

```
.....
```

```
}
```

```
function1()
```

```
{
```

```
.....
```

```
.....
```

```
}
```

```
function2()
```

```
{
```

```
.....
```

```
.....
```

```
}
```

The variables **number** and **length** are available for use in all three functions. In case a local variable and a global variable have the same name, the local variable will have precedence over global one in the function where it is declared. Consider the following example:

```
int count;
```

```
main()
```

```
{
```

```

count = 10;

.....

}

function {

{

int count = 0;

.....

count = count +1;

}

```

When the function references the variable **count**, it will be referencing only its local variable, not the global one. The value of **count** in **main** will not be affected.

Program 9.9 Write a multifunction program to illustrate the properties of global variables. A program to illustrate the properties of global variables is presented in Fig.9.15. Note that variable **x** is used in all function but none except fun2, has a definition for **x**. Because **x** has been declared ‘above’ all the functions, it is available to each function without having to pass **x** as a function argument. Further, since the value of **x** is directly available, we need not use `return(x)` statements in fun1 and fun3. However, since fun2 has a definition of **x**, it returns its local value of **x** and therefore uses a return statement. In fun2, the global **x** is not visible. The local **x** hides its visibility here.

Program

```

int fun1(void);

int fun2(void);

int fun3(void);

```

```
int x; /* global */

main()
{
    x = 10; /* global x*/
    printf("x= %d\n,x);
    printf("x= %d\n, fun1());
    printf("x= %d\n, fun2());
    printf("x= %d\n, fun3());
}

fun1(void)
{
    x = x + 10;
}

int fun2(void)
{
    int x; /* local*/
    x = 1;
    return (x);
}

fun3(void)
{
```

```
x = x+10; /*global */  
  
}
```

Output

x = 10

x = 20

x = 1

x = 30

Fig 9.15 illustration of properties of global variables

Once a variable has been declared as global, any function can use it and change its value. Then, subsequent functions can reference only that new value.

Global variables as parameters

Since all functions in a program source file can access global variables, they can be used for passing values between the function. However, using global variables as parameters for passing values poses certain problems.

- The values of global variables which are sent to the called function may be changed inadvertently by the called function.
- Functions are supposed to be independent and isolated modules. This character is lost, if they use global variables.
- It is not immediately apparent to the reader which values are being sent to the called function.
- A function that uses global variables suffers from reusability.

One other aspect of a global variable is that it is available only from the point of declaration to end of the program. Consider a program segment as shown below:

```
main()  
  
{
```



```

y = 5;

.....

.....

}

int y; /*global declaration*/

fun1()

{

y = y+1;

}

```

We have a problem here. As far as main is concerned, y is not defined, so the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement `y = y+1;` in fun1 will, therefore, assign 1 to y.

External Declaration

In the program segment above, the main cannot access the variable y as it has been declared after the main function. This problem can be solved by declaring the variable with the storage class extern.

For example:

```

main()

{

extern int y; /*external declaration*/

.....

.....

```

```

    }

    fun1()

    {

    extern int y; /*external declaration*/

    .....

    }

    int y;      /*declaration*/

```

Although the variable `y` has been defined after both the function, the external declaration of `y` inside the functions informs the compiler that `y` is an integer type defined somewhere else in the program. Note that `extern` declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.

Example:

```

main()

{

int i;

void print_out(void);

extern float height [];

.....

print_out();

}

void print_out(void)

{

```

```

extern float height[];

int i;

.....

}

float height[size];

```

An extern within a function provides the type information to just that one function. We can provide type information to all functions within a file by placing external declarations before any of them.

Example:

```

extern float height[];

main()
{
int i;

void print_out(void);

.....

.....

print_out();

}

void print_out(void)
{
int i;

.....

```

```
.....  
}
```

```
float height[size];
```

The distinction between definition and declaration also applies to function. A function is defined when its parameters and function body are specified. This tells the compiler to allocate space for the function code and provides type information for the parameters. Since functions are external by default, we declare them (in the calling functions) without the qualifier `extern`. Therefore, the declaration

```
void print_out(void);
```

is equivalent to

```
extern void print_out(void);
```

Function declarations outside of any function behave the same way as variable declarations.

Static variables

As the name suggest, the value of static variables persists until the end of the program. A variable can be declared static using the keyword `static` like

```
static int x;
```

```
static float y;
```

A static variable may be either an internal type or an external type depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scopes of internal static variables extend up to the end of the function in which they are defined. Therefore, internal static variables are similar to auto variables, except that they remain in existence (alive); throughout the remainder of the program. Therefore, internal static variables can be used to retain values between function calls. For example, it can be used to count the number of calls made to a function.

Program 9.10 Write a program to illustrate the properties of a static variable.

The program in Fig.9.16 explains the behaviour of a static variable.

Program

```
void stat(void);

main()
{
    int i;

    for(i=1; i<=3; i++)

        stat();
}

void stat(void)
{
    static int x = 0;

    x = x+1;

    printf("x = %d\n", x);
}
```

Output

x = 1

x = 2

x = 3

Fig 9.16 illustration of static variable

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to **stat**, **x** is incremented to 1. Because **x** is static, this value persists and therefore, the next call adds another 1 to **x** giving it a value of 2. The value of **x** becomes three when the third call is made.

Had we declared **x** as an **auto** variable, the output would have been:

x = 1

x = 1

x = 1

This is because each time **stat** is called; the auto variable **x** is initialized to zero. When the function terminates, its value of 1 is lost.

An external static variable is declared outside of all functions and is available to all the functions in that program. The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining 'that' function with the storage class **static**.

Register variables

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables (e.g., loop control variables) in the register will lead to faster execution of programs. This is done as follows:

register int count;

Although, ANSI standard does not restrict its application to any particular data type, most compilers allow only **int** or **char** variables to be placed in the register.

Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, C will automatically convert **register** variables into non-register variables once the limit is reached.

Table 9.1 summarizes the information on the visibility and lifetime of variables of variables in function and files.

Table9.1 scope and lifetime of variables

Storage class	where declared	visibility (active)	lifetime(Alive)
None	Before all function in a file (may be initialized)	Entire file plus other file where variable is declared with extern	Entire program(Global)
Extern	Before all function in a file (cannot be initialized) Extern and the file where originally declared as global.	Entire file plus other files where variable is declared	Global
Static	Before all function in a file	Only in that file	Global
None or auto	Inside a function (or a block)	Only in that function or block	Until end of function
Register	Inside a function or block	Only in that function or block	Until end of function or block
Static	Inside a function	Only in that function	Global

--	--	--	--

Nested blocks

A set of statements enclosed in a set of braces is known a block or a compound statement. Note that all functions including the main use compound statement. A block can have its own declarations and other statements. It is also possible to have a block of such statements inside the body of a function or another block, thus creating what is known as nested blocks as shown below:

```

main()
{
    int a =20;
    int b = 10;
    .....
    {
        int a = 0;
        int c = a + b;
        .....
    }
    b = a;
}

```

When this program is executed, the value c will be 10, not 30. The statement `b = a;` assigns a value of 20 to **b** and not zero. Although the scope of **a** extends up to the end of main it is not “visible” inside the inner block where the variable **a** has been declared again. The inner a

hides the visibility of the outer **a** in the inner block. However, when we leave the inner block, the inner **a** is no longer in scope and the outer **a** becomes visible again.

Remember, the variable **b** is not re-declared in the inner block and therefore it is visible in both the block. That is why when the statement

```
int c = a + b;
```

is evaluated, **a** assumes a values of 0 and **b** assumes a value of 10.

Although main's variables are visible inside the nested block, the reverse is not true.

Scope rules

scope

The region of a program in which a variable is available for use

visibility

The program's ability to access a variable from the memory.

lifetime

The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an auto variable declared in main is the entire program execution time, although its scope is only the main function.
5. The life of an auto variable declared in a function ends when the function is exited.
6. A static local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.

7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

9.20 MULTIFILE PROGRAMS

So far we have been assuming that all functions (including the main) are define in one file. However, in real-life programming environment, we may use more than one source files which may be compiled separately and linked later to form an executable object code. This approach is very useful because any change in one file does not affect other files thus eliminating the need for recompilation of the entire program.

Multiple source files can share a variable provided it is declared as an external variable appropriately. Variables that are shared by two or more files are global variables and therefore we must declare them accordingly in one file and the explicitly define them with **extern** in other files. Fig 9.17 illustrates the use of **extern** declarations in a multifile program.

The function main in **file1** can reference the variable **m** that is declared as global in **file2**. Remember, function1 cannot access the variable **m**. if, however, the **extern int k;** statement is placed before main, then both the function could refer to m. This can also be achieved by using **extern int m;** statement inside each function in file1.

The extern specifier tells the complier that the following variable types and names have already been declared elsewhere and no need to create storage space for them. It is the responsibility of the linker to resolve the reference problem. It is important to note that a multifile global variable should be declared without extern in one (and only one) of the files. The extern declaration is done in places where secondary references are made. If we declare a variable as global in two different files used by a single program, then the linker will have a conflict as to which variable to use and, therefore, issues a warning.

```
file1.c  
  
main()  
  
{
```

```
extern int m;
```

```
int i;
```

```
.....
```

```
.....
```

```
}
```

```
function1()
```

```
{
```

```
int j;
```

```
.....
```

```
}
```

```
file2.c
```

```
int m/*global variable*/
```

```
function2()
```

```
{
```

```
int i;
```

```
.....
```

```
.....
```

```
}
```

```
function3()
```

```
{
```

```
int count;
```

```

.....
.....
}

```

Fig. 9.17 use of extern in a multifile program

the multiple program shown in fig 9.18 can be modified as shown in fig 9.17

file1.c

```
main()
```

```
    int m; /*global variable*/
```

```
{
```

```
    int i;
```

```
    .....
```

```
    .....
```

```
}
```

```
function1()
```

```
{
```

```
    int j;
```

```
    .....
```

```
}
```

file2.c

```
extern int m
```

```

function2()
{
int i;
.....
.....
}

function3()
{
int count;
.....
.....
}

```

Fig 9.18 Another version of a multifile program

When a function is defined in one file and accessed in another, the later file must include a function declaration. The declaration identifies the function as an external function whose definition appears elsewhere. We usually place such declarations at the beginning of the file, before all functions. Although all functions are assumed to be external, it would be a good practice to explicitly declare such functions with the storage class `extern`.

Calculation of area under a curve

One of the applications of computers in numerical analysis is computing the area under a curve. One simple method of calculating the area under a curve is to divide the area into number of trapezoids of same width and summing up the area of individual trapezoids. The area of trapezoids is given by

Area = $0.5 * (h1 + h2) * b$; where $h1$ and $h2$ are the heights of two sides and b is the width as shown in fig.9.19

The program in fig 9.21 calculates the area for a curve of the equation

$F(x) = x^2 + 1$ between any two given limits, say, A and B.

Input

Lower limit(A)

Upper limit(B)

Number of trapezoids

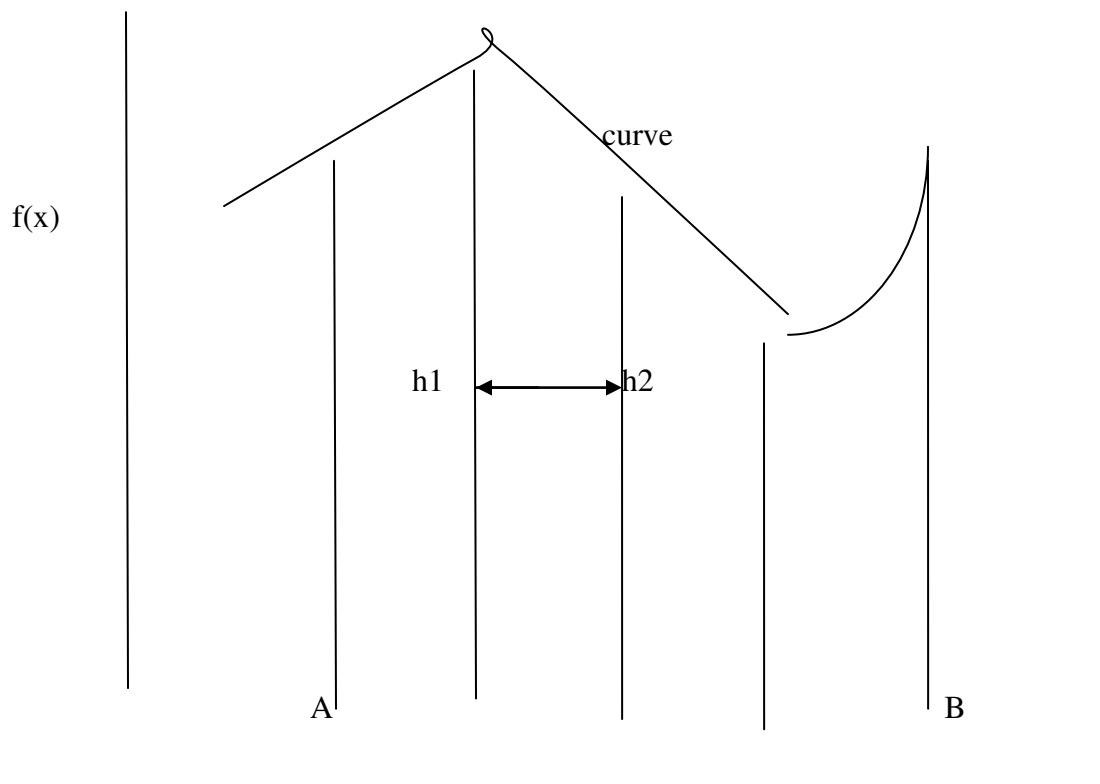


Fig.9.19 Total area under the curve between the given limits.

The output for two runs shows that better accuracy is achieved with larger number of trapezoids. The actual area for the limits 0 and 3 is 12 units (by analytical method).

Program

```
#include<stdio.h>

float start_point, /*global variables*/
end_point, total_area;

int numtraps;

main()
{
    void input(void);

    float find_area(float a, float b, int n);

    print("AREA UNDER A CURVE");

    input();

    total_area = find_area(start_point, end_point, numtraps);

    Printf("total area %f, total_area);
}

void input(void)
{
    printf("\n enter lowe limit:");

    scanf("%f", &start_point);

    printf("enter upper limit:");
```

```

scanf("%f", &end_point);

printf("enter number of traperzoids:");

scanf("%d", &numtraps);
}

float find_area(float a, float b, int n)
{
    float base, lowe, h1, h2; /*local variables*/

    float function_x(float x); /*prototype*/

    float trap_area(float h1, float h2, float base); /*prototype*/

    base = (b-1)/n;

    lower = a;

for(lower =a; lower <= b-base; lower = lower + base)
{
    h1 = function_x (lower);

    h1 = function_x(lower + base);

    total_area += trap_area(h1, h2, base);

}

return(total_area);

float trap_area(float height_1, float height_2, float base)
{
    float area;

```



```
        area = 0.5 * (height_1 + height_2) *base;

        return(area);

    }

float function_x(float x)

{

return(x*x+1);    }
```

Output

Area under a curve

Enter lower limit: 0

Enter upper limit: 3

Enter number of trapezoids: 30

Total area= 12.005000

Area under a curve

Enter lower limit: 0

Enter upper limit: 3

Enter number of trapezoids: 100

Total area= 12.0004738

Fig. 9.21 computing area under a curve

UNIT II

1 STRUCTURES AND UNIONS

Key terms

Array |structure | dot operator| union|bit field

1.1 INTRODUCTION

We have seen that arrays can be used to represent a group of data items that belongs to the same type, such as int or float. However, we cannot use an array if we want to represent a collection of data items of different types using a single name. Fortunately, C supports a constructed data type known as structures, a mechanism for packing data of different items. For example, it can be used to represent a set of attributes, such as student_name, roll_number and marks. The concept of a structure is analogous to that of a record in many other languages. More, examples of such structures are:

time : seconds, minutes, hours

date : day, month, year

book : author, title, price, year

city : name, country, population

address : name, door-number, street, city

inventory : item, stock, value

customer : name, telephone, city, category

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design. This chapter is devoted to the study of structures and their application in program development. Another related concept known as unions is also discussed.

1.2 DEFINING A STRUCTURE

Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book data base consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

```
struct book_bank  
  
{  
  
char title[20];  
  
char author[15];  
  
int pages;  
  
float price;  
  
};
```

The keyword **struct** declares a structure to hold the details of four data fields, namely title, author, pages and price. These fields are called structure elements or members. Each member may belong to a different type of data. `book_bank` is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure.

Note that the above definition has not declared any variables. It simply describes a format called template to represent information as shown below:

Title	Array of 20 characters
author	Array of 15 characters
Pages	Integer
price	Float

the general format of a structure definition is as follows:

```
struct tag_name  
  
{  
  
    data_type member1;  
  
    data_type member2;  
  
    .....  
  
    .....  
  
};
```

In defining a structure you may not the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as book_bank can be used to declare structure variables of its type, later in the program.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a structure is a programmer-defined one.

3. Any array behaves like a built-in data type. All we have to do is to declare an array variable can use it. But in the case of a struct, first we have to design and declare a data structure before the variables of that type are declared and used.

1.3 DECLARING STRUCTURE VARIABLES

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements:

1. The keyword struct.
2. The structure tag name.
3. List of variable names separated by commas.
4. A terminating semicolon.

For example, the statement

```
struct book_bank, book1, book2, book3;
```

declares book1, book2, and book3 as variables of type struct **book_bank**.

Each one of these variables has four members as specified by the template. The complete declaration might look like this:

```
struct book_bank  
  
{  
  
char title[20];  
  
char autghor[15];  
  
int pages;  
  
float price;  
  
};  
  
struct book_bank book1, book2, book3;
```

remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as **book1**. When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration

```
struct book_bank  
  
{  
  
char title[20];  
  
char author[15];  
  
int pages;  
  
float price;  
  
} book1, book2, book3;
```

is valid. The use of tag name is optional here. For example:

```
struct  
  
{  
  
.....  
  
.....  
  
} book1, book2, book3;
```

declares **book1**, **book2**, and **book3** as structure variables representing three books, but does not include a tag name. However, this approach is not recommended for two reasons.

1. Without a tag name, we cannot use it for future declarations:

2. Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the **main**, along with macro definitions, such as **#define**. In such cases, the definition is global and can be used by other functions as well.

TYPE-DEFINED STRUCTURES

We can use the keyword **typedef** to define a structure as follows:

```
typedef struct  
  
{  
  
.....  
  
type member1;  
  
type member2;  
  
.....  
  
.....  
  
} type_name;
```

The `type_name` represents structure definition associated with it and therefore can be used to declare structure variables as shown below:

```
type_name variable1, variable2,.....;
```

Remember that (1) the name `type_name` is the type definition name, not a variable and (2) we cannot define a variable with typedef declaration.

Program1.1 Explain how complex number can be represented using structures. Write two C functions: one to return the sum of two complex numbers passed as parameters.

A complex number has two parts: real and imaginary. Structures can be used to realize complex numbers in C, as shown below:

```
struct complex /*declaring the complex number datatype using structure*/
```

```

{
    double real; /* real part*/
    double imag; /*imaginary part*/
};

```

function to return the sum of two complex numbers

```
struct complex add(struct complex c1, struct complex c1)
```

```

{
    struct complex c3;
    c3.real = c1.real+c2.real;
    c3.img = c1.img+c2.img;
    return(c3);
}

```

function to return the product of two complex numbers

```
struct complex product(struct complex c1, struct complex c1)
```

```

{
    struct complex c3;
    c3.real = c1.real*c2.real-c1.img*c2.img;
    c3.img=c1.real*c2.img+c1.img*c2.real;
    return(c3);
}

```


1.4 ACCESSING STRUCTURE MEMBERS

We can access and assign values to the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be, linked to the structure variables in order to make them meaningful members. For example, the word **title**, has no meaning whereas the phrase ‘title of book3’ has a meaning. The link between a member and a variable is established using the member operator ‘.’ Which is also known as ‘dot operator’ or ‘period operator’. For example,

book1.price

is the variable representing the price of **book1** and can be treated like any other ordinary variable. Here is how we would assign values to the members of **book1**:

```
strcpy(book1.title, "BASIC");
```

```
strcpy(book1.author, "balagurusamhy");
```

```
book1.pages = 250;
```

```
book1.price = 120.50;
```

we can also use **scanf** to give the values through the keyword.

```
scanf("%s\n", book1.title);
```

```
scanf("%d\n", &book1.pages);
```

are valid input statements.

Program1.2 Define a structure type, struct personal that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

Structure definition along with the program is shown in fig.1.1. the **scanf** and **printf** functions illustrate how the member operator ‘.’ Is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

Program

```
struct personal
{
    char name[20];
    int day;
    char month[10];
    int year;
    float salary;
}

main()
{
    struct personal person;
    printf("input values \n");
    scanf("%s %d %s %f,
        person.name,
        &person.day,
        person.month,
        &person.year,
        &person.salary);
    printf("%s %d %s %d %f\n",
        person.name,
        person.day,
```

```

        person.month,
        person.year,
        person.salary);
    }

```

Output

Input values

M.L.Goel 10 January 1945 4500

M.L.Goel 10 January 1945 4500.00

Fig 1.1 Defining and accessing structure members.

1.5 STRUCTURE INTIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```

main()
{
    struct
    {
        int weight;
        float height;
    }
    student = {60, 180.75};
    .....
    .....

```

```
}
```

This assigns the value 60 to student.weight and 180.75 to student.height. There is one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
    struct st_record
    {
        int weight;
        float height;
    };
    struct st_record student1 = {60, 180.75};
    struct st_record student2 = {53, 170.60};
    .....
    .....
}
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct st_record
{
    int weight;
```

```

float height;

} student1 = {60, 180.75};

main()
{
struct st_record student2 = {532,170.60};

.....

.....

}

```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

Note that the compile-time initialization of a structure variable must have the following elements:

1. The keyword **struct**.
2. The structure tag name.
3. The name of the variable to declared.
4. The assignment operator =.
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

RULES FOR INITIALIZING STRUCTURES

There are a few rules to keep in mind while initializing structure variables at compile-time.

1. We cannot initialize individual members inside the structure template.
2. The order of values enclosed in braces must match the order of members in the structure definition.

3. It is permitted to have a partial initialization. We can initialize only the first few members and leave the remaining blank. The uninitialized members should be only at the end of the list.
4. The uninitialized members will be assigned default values as follows:
 - **Zero for integer and floating point numbers**
 - `'\0'` for characters and strings.

1.6 COPYING AND COMPARING STRUCTURE VARIABLES

Two variables of the same structure type can be copied the same way as ordinary variables. If `person1` and `person2` belong to the same structure, then the following statements are valid:

```
person1 = person2;
```

```
person2 = person1;
```

However, the statements such as

```
person1 == person2
```

```
person1 != person2
```

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

Program1.3 Write a program to illustrate the comparison of structure variables.

The program shown in 10.2 illustrates how a structure variable can be copied into another of the same type. It also performs member-wise comparison to decide whether two structure variables are identical.

Program

```
struct class
```

```

{
    int number;\char name[20];

    floatr marks;
};

main()
{
    int x;

    struct class studen1 = { 111,"rao",72.50};

    struct class student2 = {222,"reddy", 67.00};

    struct class student3;

    student3 = student2;

    x = ((student3.number == student2.numbr) &&
        (student3.marks == student2.marks)) ? 1 :0;

    if (x == 1)
    {
        printf("\nstudent2 and student3 are same \n\n");

        printf("%d %s %f\n", student3.number,
            student3.name,
            student3.marks);
    }else
        printf("\nstudent2 and student3 are different\n\n");
}

```

```
}
```

Output

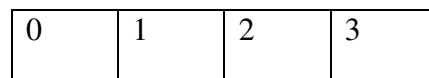
Student2 and student3 are same

222 Reddy 67.000000

Fig.1.2 Comparing and copying structure variables

WORD BOUNDARIES AND SLACK BYTES

Computer stores structures using the concept of “word boundary”. The size of a word boundary is machine dependent. In a computer with two bytes word boundary. The members of a structure are stored left_aligned on the word boundary, as shown below. A character data takes one byte and an integer takes two bytes. One byte between them is left unoccupied. This unoccupied byte is known as the slack byte.



Char slack int

Byte

When we declare structure variables, each one of them may contain slack bytes and the values stored in such slack bytes are undefined. Due to this, even if the members of two variables are equal, their structures do not necessarily compare equal C, therefore, does not permit comparison of structures. However, we can design our own function that could compare individual members to decide whether the structures are equal or not.

1.7 OPERATORS ON INDIVIDUAL MEMBERS

As pointed out earlier, the individual members are identified using the member operator, the dot. A member with the dot operator along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators. Consider the program in fig.10.2 we can perform the following operations:


```
if(student1.number == 111)

student1.marks += 10.00;

float sum = student1.marks + student2.marks;

student2.marks *= 0.5;
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid.

```
student1.numbr ++;

++ student1.number;
```

The precedence of the member operator is higher than all arithmetic and relational operators and therefore no parentheses are required.

THREE WAYS TO ACCESS MEMBERS

We have used the dot operator to access the members of structure variables. In fact, there are two other ways. Consider the following structure:

```
typedef struct

{

int x;

int y;

}vector;

vector v, *ptr;

ptr = & v;
```

The identifier ptr is known as pointer that has been assigned the address of the structure variable. Now, the members can be accessed in three ways.

- Using dot notation : v.x
- Using indirection notation : (*ptr).x
- Using selection notation : ptr→x

The second and third methods will be considered in chapter 11.

1.8 ARRAYS OF STRUCTURES

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure, each element of the array representing a structure variable. For example:

```
struct class student[100];
```

Defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class**. Consider the following declaration:

```
struct marks
{
    int subject1;
    int subject2;
    int subject3;
};

main()
{
    struct marks student[3] =
    {{45,68,81},{75,53,69},{57,36,71}};
```

This declares the student as an array of three elements student[0], student[1], and student[2] and initializes their members as follows:

```
student[0].subject1 = 45;
```

```
student[0].subject2 = 65;
```

```
.....
```

```
.....
```

```
student[2].subject3 = 71;
```

Note that the array is declared just as it would have been with any other array. Since student is an array, we use the usual array-accessing methods to access individual elements and then the member operator to access members. Remember, each element of **student** array is a structure variable with three members.

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array student actually looks as shown in fig.10.3.

Program1.4 For the student array discussed above, write a program to calculate the subject-wise and student-wise totals and store them as a part of the structure. The program is shown in Fig.10.4. we have declared a four-member structure, the fourth one for keeping the student-totals. We have also declared an array total to keep the subject-totals and the grand-total. The grand-total is given by total.total. Note that a member name can be any valid C name and can be the same as an existing structure variable name. The linked name total.total represents the total member of the structure variable total.

45
68
81
75
53
69

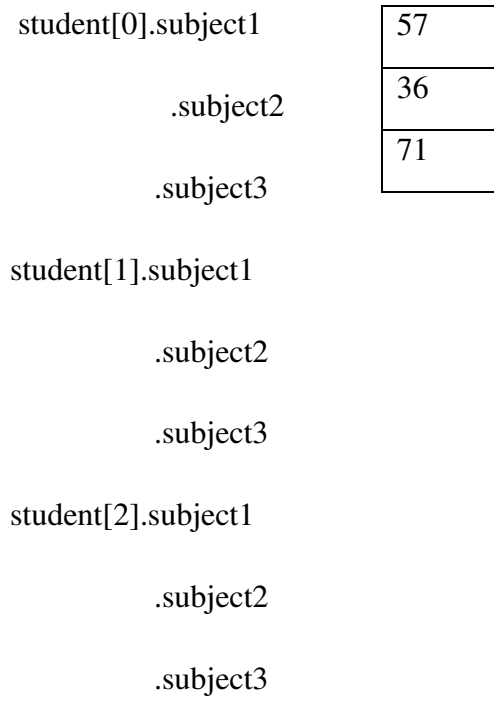


Fig. 1.3 the array student inside memory

Program

```

struct marks
{
    int sub1;
    int sub2;
    int sub3;
    int total;
};

main()
{
    int i;

```

```

struct marks student[3] = {{45,67,81,0},
                           {75,53,69,0}
                           {57,46,71,0}};

struct marks total;

for(i=0; i<=2; i++)
{
student[i].total = student[i].sub1 +
                  student[i].sub2+
                  student[i].sub3;

total.sub1 = total.sub1 + student[i].sub1;
total.sub2 = total.sub2 + student[i].sub2;
total.sub3 = total.sub3 + student[i].sub3;
total.total = total.total + student[i].total;
}

printf("STUDENT   TOTAL \n\n");

for(i=0; i<=2; i++)

printf("student[%d]  %d\n", i+1,student[i].total);

printf("\n SUBJECT TOTAL\n\n");

printf("%s  %d\n%s  %d\n",
       "subject 1 ",total.sub1,
       "subject2 ",total.sub2,

```

```

        "subject3 ",total.sub3);

printf("\nGrand total = %d\n", total.total);

}

```

Output

STUDENT	TOTAL
Student[1]	193
Student[2]	197
Student[3]	164
SUBJECT	TOTAL
Subject 1	177
Subject 2	156
Subject 3	221

Grand total = 554

Fig.1.4 Arrays of structures: illustration of subscripted structure variables

1.9 ARRAYS WITHIN STRUCTURES

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single-dimensional or multi-dimensional arrays of type int or float. For example, the following structure declaration is valid:

```

struct marks
{
int number;

float subject[3];

```

```
} student[2];
```

Here, the member subject contains three elements, subject[0], subject[1] and subject[2]. These elements can be accessed using appropriate subscripts. For example, the name

```
student[1].subject[2];
```

Would refer to the marks obtained in the third subject by the second student.

Program1.5 Rewrite the program of program 10.4 using an array member to represent the three subjects.

The modified program is shown in Fig.10.5. You may notice that the use of array name for subjects has simplified in code.

Program

```
main()
{
    struct marks
    {
        int sub[3];
        int total;
    };
    struct marks student[3] = {45,67,81,0,57,26,72,0};
    struct marks total;
    int i,j;
    for(i=0; i<=2; i++)
    {
```

```

for(j=0; j<=2; j++)
{
student[i].total +=stidemt[i].sub[j];

total.sub[j] += student[i].sub[j];

}

total.total += student[i].total;

}

printf("student total\n\n")

for(i=0; i<=2; i++)

printf("student[%d] %d\n",i+1,student[i].total);

printf("\n subject total\n\n");

for(j=0; j<=2; j++)

printf("subject %d %d", j+1, total.sub[j]);

printf("\ngrand total = %d\n", total.total);

}

```

Output

STUDENT	TOTAL
Student[1]	193
Student[2]	197
Student[3]	164
STUDENT	TOTAL

Student-1		177
Student-2		156
Student-3		221
Grand total	=	554

Fig1.5 Use of subscripted members arrays in structures

1.10 STRUCTURE WITHIN STRUCTURES

Structures within a structure means nesting of structures. Nesting of structures is permitted in C. let us consider the following structure defined to store information about the salary of employees.

```
struct salary  
  
{  
  
    char name;  
  
    char department;  
  
    int basic_pay;  
  
    int dearness_allowance;  
  
    int house_rent_allowance;  
  
    int city_allowance;  
  
}  
  
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below:

```
struct salary  
  
  {  
  
    char name;  
  
    char department;  
  
    struct  
    {  
  
      int dearness;  
  
      int house_rent;  
  
      int city;  
  
    }  
  
    allowance;  
  
  }  
  
  employee;
```

The salary structure contains a member named allowance, which itself is a structure with three members. The members contained in the inner structure namely dearness, house_rent, and city can be referred to as:

employee.allowance.dearness

employee.allowance.house_rent

employee.allowance.city

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following are invalid

```
employee.allowance(actual member is missing)
```

```
employee.house_rent(inner structure variable is missing)
```

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
```

```
{
```

```
.....
```

```
struct
```

```
{
```

```
int dearness;
```

```
.....
```

```
}
```

```
allowance;
```

```
arrears;
```

```
}
```

```
employee[100];
```

The inner structure has two variables, allowance and arrears. This implies that both of them have the same structure template. Note the comma after the name allowance. A base member can be accessed as follows:

```
employee[1].allowance.dearness
```

employee[1].arrears.dearness

We can also use tag names to define inner structures. Example:

```
struct pay
{
    int dearness;
    int house_rent;
    int city;
};

struct salary
{
    char name;
    charn department;
    struct pay allowance;
    struct pay arrears;
};

struct salary employee[100];
```

pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

It is also permissible to nest more than one type of structures.

```
struct personal_record
{
```

```

    struct name_part name;

    struct addr_part address;

    struct date date_of_birth;

    .....

    .....

};

struct personal_record personal;

```

The first member of this structure is name, which is of the type **struct name_part**. Similarly, other members have their structure types.

NOTE:

C permits nesting upto 15 levels. However, C99 allows 63 levels of nesting.

1.11 STRUCTURES AND FUNCTIONS

We know that the main philosophy of C language is the use of functions. And therefore, it is natural that C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All the compilers may not support this method of passing the entire structure as a parameter.

3. The third approach employs a concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function. This method is more efficient as compared to the second one.

In this section, we discuss in detail the second method, while the third approach using pointers is discussed in the next chapter, where pointers are dealt in detail.

The general format of sending a copy of a structure to the called function is:

function_name(structure_variable_name):

The called function takes the following form:

```
data_type function_name(struct_type st_name)
{
.....
.....
return(expression);
}
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as **struct** with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
3. The **return** statement is necessary only when the function is returning some data back to the calling function. The expression may be any simple variable or structure variable or an expression using simple variables.

4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called functions must be declared in the calling function appropriately.

Program 1.6 Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

A program to update an item is shown in Fig.10.6. the function **update** receives a copy of the structure variable **item** as one of its parameters. Note that both the function **update** and the formal parameter **product** are declared as type **struct stores**. It is done so because the function uses the parameter **product** to receive the structure variable item and also to return the update values of **item**.

The entire structure returned by update can be copied into a structure of identical type. The statement

```
item = update(item,p_increment,q_increment);
```

replaces the old values of **item** by the new ones.

Program

```
/* passing a copy of the entire structure */
```

```
    struct stores
```

```
    {
```

```
        char name[20];
```

```
        float price;
```

```
        int quantity;
```

```
    };
```

```
struct stores update(struct stores product, float p, int q);
```

```
float mul(struct stores stock);
```

```

main()
{
    floatr p_increment, value;

    int q_increment;

    struct stores item = ("xyz",25.75,12);

    printf("\n input increment values:");

    printf(" price increment and quantity increment\n");

    scanf("%f %d, &p_increment, &q_increment);

    item = update(itrem, p_increment, q_increment);

    printf("updated values of item\n\n");

    printf("name   :%s\n", item.name);

    printf("price   :%f\n", item.price);

    printf("quantity      :%d\n", item.quantity);

    value = mul(item);

    printf("\nvalue of the item = %f\n", value);

}

```

```

struct stores update(struct stores product, float p, int q)
{
    product.price+=p;

    product.quantity += q;

    return(product);
}

```



```

    }

float mul(struct stores stock)

{

return(stock.price * stock.quantity);

}

```

Output

Input increment: price increment and quantity increment

10 12

Updated values of item

Name :XYZ

Price :35.750000

Quantity: 24

Value of the item = 858.000000

Fig.1.6 Using structure as a function parameter

You may notice that the template of stores is defined before main(). This has made the data type **struct stores** as global and has enabled the functions **update** and **mul** to make use of this definition.

1.12 UNIONS

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in term of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

union item

```
{  
  
int m;  
  
floatx;  
  
char c;  
  
}code;
```

This declares a variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

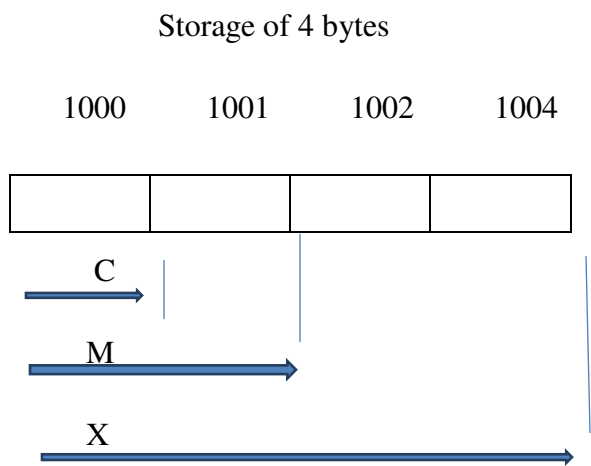


Fig.1.7 Sharing of a storage locating by union members

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. Figure 1.7 shows how all the three variables share the same address. This assumes that a float variable requires 4 bytes of storage.

To access union member, we can use the same syntax that we use for structure members. That is,

code.m

code.x

code.c

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m = 379;
```

```
code.x = 7859.36;
```

```
printf("%d", code.m);
```

would produce erroneous output(which is machine dependent).

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous member's values.

Unions may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside a structure remains the same as for the nested structures.

Unions may be initialized when the variable is declared. but, unlike structures, it can be initialized only with a value of the same type as the first union member. For example, with the preceding, the declaration

```
union item abc = {100};
```

is valid but the declaration

```
union item abc = {10.75};
```

is invalid. This is because the type of the first member is int. other members can be initialized by either assigning values or reading from the keyboard.

1.13 SIZE OF STRUCTURES

We normally use structures, unions, and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator **sizeof** to tell us the size of a structure (or any variable). The expression

sizeof(struct x)

Will evaluate the number of bytes required to hold all the members of the structure **x**. If **y** is a simple structure variable of type **struct.x**, expression

sizeof(y)

would also give the same answer. However, if **y** is an array variable of type **struct x**, then

sizeof(y)

would give the total number of bytes the array **y** requires.

This kind of information would be useful to determine the number of records in a database. For example, the expression

sizeof(y)/sizeof(x)

would give the number of elements in the array **y**.

1.14 BIT FIELDS

So far, we have been using integer fields of size 16 bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small bit field to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits as if it represented an integral quantity.

A bit field is a set of adjacent bits whose size can be from 1 to 16 bits in length. A word can therefore be divided into a number of bit fields. The name and size of bit fields are defined using a structure. The general form of bit field definition is

struct tag_name

```

{
    data-type name1: bit-length;

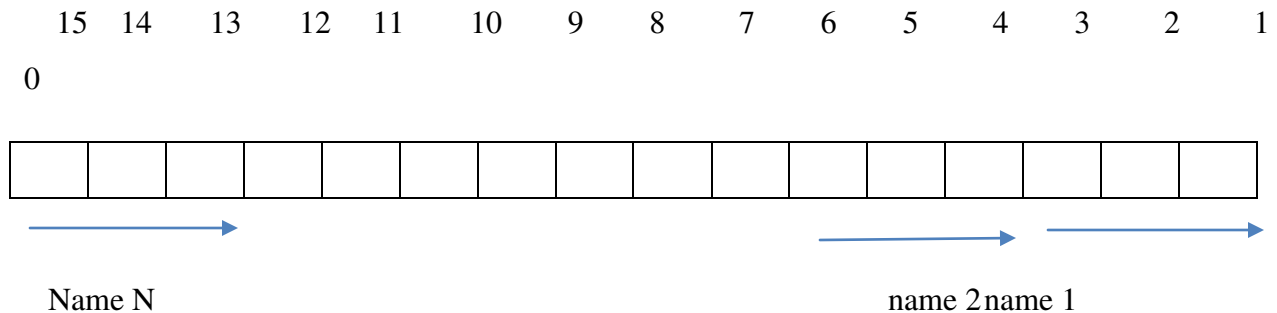
    data-type name2: bit-length;
    .....
    .....

    data-type namen: bit-length;
}

```

The data-type is either int or unsigned int or signed int and the bit-length is the number of bits used for the specified name. remember that a signed bit field should have at least 2 bits (one bit for sign). Note that the field name is followed by a colon. The bit-length is decided by the range of value to be stored. The largest value that can be stored is 2^{n-1} , where n is bit-length.

The internal representation of bit fields is machine dependent. That is, it depends on the size of **int** and the ordering of bits. Some machines store bits from left to right and others from right to left. The sketch below illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.



There are several specific points to observe:

1. The first field always starts with the first bit of the word.
2. A bit field cannot overlap integer boundaries. That is, the sum of lengths of all the fields in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.

3. There can be unnamed fields declared with size. Example:
 - i. **Unsigned** : bit-length
such fields provide padding within the word.
4. There can be unused bits in a word.
5. We cannot take the address of a bit field variable. This means we cannot use scanf to read values int bit fields. We can neither use pointer to access the bit fields.
6. Bit fields cannot be arrayed.
7. Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behavior would be unpredicted.

Suppose, we want to store and use personal information of employees in compressed form this can be done as follows:

```

struct personal
{
    unsigned sex      :    1
    unsigned age      :    7
    unsigned m_status :    1
    unsigned children :    3
    unsigned          :    4
}emp;

```

This defines a variable name emp with four bit fields. The range of values each field could have is follows:

Bit field	bit length	range of value
sex	1	0 or 1
age	7	0 or $127(2^7-1)$

m_status	1	0 or 1
Children	3	0 to 7(2 ³ -1)

Once bit fields are define, they can be referenced just as any other structure-type data item would be referenced. The following assignment statements are valid.

```
emp.sex = 1;
```

```
emp.age = 50;
```

Remember, we cannot use scanf to read values into a bit field. We may have to read into a temporary variable and then assign its value to the bit field. For example:

```
scanf("%d %d", &AGE, &CHILDREN);
```

```
emp.age = AGE;
```

```
emp.children = CHILDREN;
```

One restriction in accessing bit fields is that a pointer cannot be used. However, they can be used in normal expressions like any other variable. For example:

```
sum = sum + emp.age;
```

```
if(emp.m_status).....;
```

```
printf("%d\n", emp.age);
```

Are valid statements.

It is possible to combine normal structure elements with bit field elements. For example:

```
struct personal
```

```
{
```

```
    char name[20]; /*normal variable*/
```

```
    struct addr address; /*structure variable*/
```

```

        unsigned sex : 1;

        unsigned age : 7;

        .....

        .....

    }

    emp[100];

```

This declares emp as a 100 element array of type **struct personal**. This combines normal variable name and structure type variable address with bit fields.

Bit fields are packed into words as they appear in the definition. Consider the following definition.

```

struct pack
{
    unsigned a:2;

    int count;

    unsigned b : 3;

};

```

Here, the bit field a will be in one word, the variable count will be in the second word and the bit5 field b will be in the third word. The fields a and b would not get packed into the same word.

NOTE: Other related topics such as ‘structures with pointers’ and ‘structures and linked lists’ are discussed in chapter 11 and chapter 12, respectively.

2 POINTERS

Key Terms

Pointer| memory | pointer | variables | call by reference| call by value

2.1 INTRODUCTION

A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are undoubtedly one of the most distinct and exciting features of C language. It has added power and flexibility to the language. Although they appear little confusing and difficult to understand for a beginner, they are a powerful tool and handy to use once they are mastered.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They include

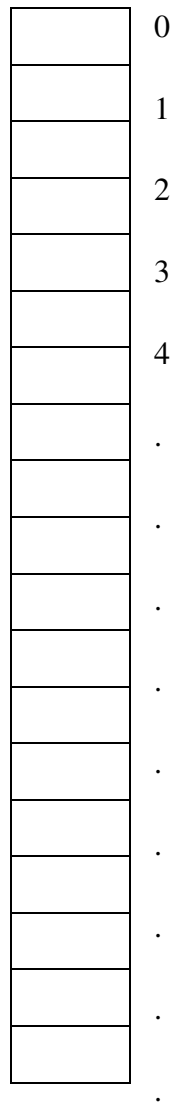
1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Of course, the real power of C lies in the proper use of pointers. In this chapter, we will examine the pointers in detail and illustrate how to use them in program development. Chapter 13 examines the use of pointers for creating and managing linked lists.

2.2 UNDERSTANDING POINTERS

The computer's memory is a sequential collection of storage cells as shown in fig.2.1. Each cell, commonly known as a byte, has a number called address associated with it. Typically, the addresses are numbered consecutively, starting from zero. The last address depends on the memory size. A computer system having 64 K memory will have its last address as 65,535.

Memory cell Address



65,535

Fig. 2.1 Memory organization

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number. Consider the following statement

int quantity = 179;

This statement instruct the system to find a location for the integer variable **quantity** and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for **quantity**. We may represent this as shown in Fig.2.2.(Note that the address of a variable is the address of the first byte occupied by that variable.)

During execution of the program, the system always associates the name **quantity** with the address 5000. (This is something similar to having a house number as well as a house name.) We may have access to the value 179 by using either the name **quantity** or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables that can be stored in memory, like any other variable. Such variables that hold memory addresses are called pointer variables. A pointer variable is, therefore, nothing but a variable that contains an address, which is a location of another variable in memory.

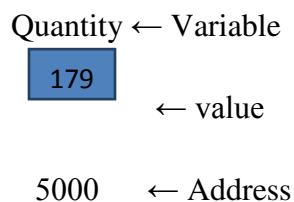


Fig.2.2 Representation of a variable

Remember, since a pointer is a variable, its value is also stored in the memory in another location. Suppose, We assign the address of **quantity** to a variable **p**. The link between the variable **p** and **quantity** can be visualized as shown in Fig.2.3. The address of **p** is 5048.

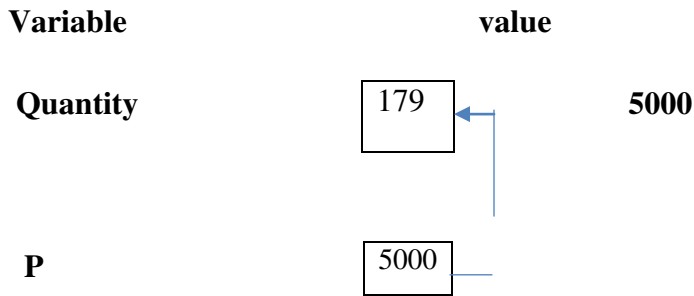
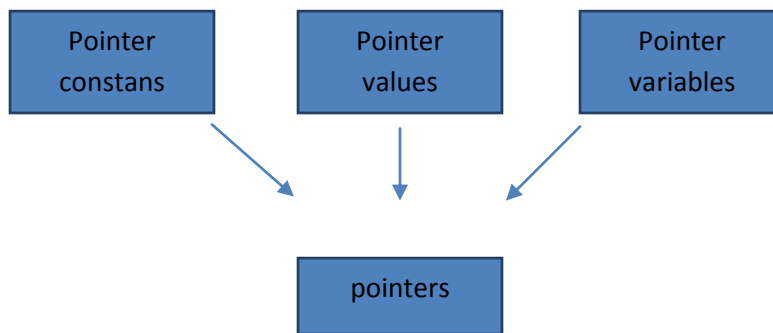


Fig.2.3 Pointer variable

Since the value of the variable **p** is the address of the variable **quantity**, we may access the value of **quantity** by using the value of **p** and therefore, we say that the variable **p** ‘points’ to the variable **quantity**. Thus, **p** gets the name ‘pointer’. (we are not really concerned about the actual values of pointer variables. They may be different every time we run the program. What we are concerned about is the relationship between the variables **p and quantity**.)

Underlying concepts of pointers

Pointers are built on the three underlying concepts as illustrated below:]



Memory addresses within a computer are referred to as pointer constants. We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as pointer value. The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a pointer variable.

2.3 ACCESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. How can we then determine the address of a variable? This can be done with the help of the operator **&** available in C. We have already seen the use of this address operator in the **scanf** function. The operator **&** immediately preceding a variable return the address of the variable associated with it. For example, the statement

$$P = \&\text{quantity}$$

Would assign the address 5000 (the location of quantity) to the variable p. The **&** operator can be remembered as address of'.

The **&** operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1. **&125** (pointing at constants)
2. **int x[10];**

&x (pointing at array names).

3. **&(x+y)** (pointing at expressions).

If **x** is an array, then expressions such as

$$\&x[0] \text{ and } \&x[i+3]$$

are valid and represent the addresses of 0th and (i+3)th elements of x.

Program 2.1 Write a program to print the address of a variable along with its value.

The program shown in Fig. 2.4, declares and initializes four variables and then prints out these values with their respective storage locations not that we have used %u format for printing address values. Memory addresses are unsigned integers.

Program

```
main()
{
    char a;

    int x;

    float p,q;

    a = 'á';

    x = 125;

    p = 10.25, q = 18.76;

    printf("%c is stored at addr %u.\n", a, &a);
    printf("%d is stored at addr %u.\n", x, &x);
    printf("%f is stored at addr %u.\n", p, &p);
    printf("%f is stored at addr %u.\n", q, &q);
}
```

Output

A is stored at addr 4436

125 is stored at addr 4434

10.250000 is stored at addr 4442.

18.760000 is stored at addr 4438.

Fig.2.4 Accessing the address of a variable

2.4 DECLARING POINTER VARIABLES

In C, every variable must be declared for its type. Since pointer variables contain addresses that belongs to a separate data type, they must be declared as pointers before we use them. The declaration of pointer variable takes the following form:

data_type*pt_name;

This tells the compiler three things about the variable `pt_name`.

1. The asterisk(*) tells that the variable **pt_name** is a pointer variable.
2. **Pt_name** needs a memory location.
3. **Pt_name** points to a variable of type `data_type`.

for example,

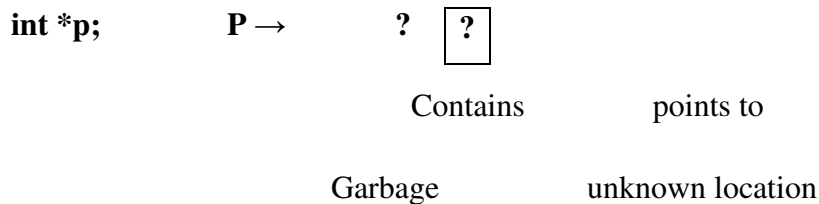
int *p; /*integer pointer */

declares the variable `p` as a pointer variable that points to an integer data type. Remember that the type `int` refers to the data type of the variable being pointed to by `p` and not the type of the value of the pointer. Similarly, the statement

float *x; /*float pointer*/

declares `x` as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables `p` and `x`. Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:



Pointer Declaration style

Pointer variables are declared similarly as normal variables except for the addition of the unary * operator. This symbol can appear anywhere between the type name and the pointer variable name. Programmers use the following styles:

```
int* p; /*style1 */
```

```
int *p; /*style2 */
```

```
int * p; /*style3*/
```

However, the style 2 is becoming increasingly popular due to the following reasons:

1. This style is convenient to have multiple declarations in the same statement. Example: **int *p, x,*q;**

2. This style matches with the format used for accessing the target values, Example: **int x, *p,y;**

```
x = 10;
```

```
p = &x;
```

```
y = *p; /* accessing x through p */
```

```
*p = 20; /* assigning 20 to x */
```

We use in this book the style 2, namely,

```
int*p;
```

2.5 INITIALIZATION OF POINTER VARIABLES

The process of assigning the address of a variable to a pointer variable is known as initialization. As pointer out earlier, all uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;  
  
int *p;  
  
p = &quantity; /* initialization*/
```

We can also combine the initialization with the declaration. That is,

```
int *p = &quantity;
```

is allowed. The only requirement here is that the variable **quantity** must be declared before the initialization takes place. Remember, this an initialization of **p** and not ***p**.

We must ensure that the pointer variables always point to the corresponding type of data. For example.

```
float a,b;  
  
int x, *p;  
  
p = &a;      /* wrong */  
  
b = *p;
```

Will result in erroneous output because we are trying to assign the address of a float variable to an **integer pointer**. **When we declare a pointer to be of int type**, the system assumes that any address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variables and the initialization of the pointer variable in one step. For example,

```
int x, *p = &x;          /* three in one */
```

is perfectly valid. It declares **x** as an integer variable and **p** as a pointer variable and then initializes **p** to the address of **x**. And also remember that the target variable **x** is declared first. The statement

```
int *p = &x, x;
```

is not valid.

We could also define a pointer variable with an initial value of NULL or 0(zero). That is, the following statements are valued

```
int *p = NULL;
```

```
int *p = 0;
```

POINTER FLEXIBILITY

Pointers are flexible. We can make the same pointer to point to different data variables in different statements. Example:

```
int x, y, z, *p;
```

.....

```
p = &x;
```

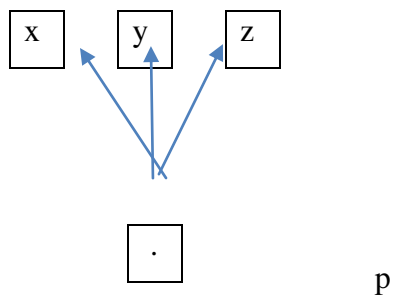
```
int x, y, z, *p;
```

.....

```
p = &y;
```

.....

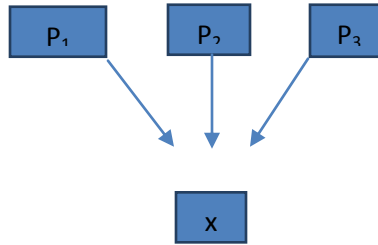
```
p = &z;
```



.....

We can also use different pointer to point to the same data variable. Example:

```
int x;  
int *p1 = &x;  
int *p2 = &x;  
int *p3 = &x;
```



.....

With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;          /* absolute address */
```

11.6 ACCESSING A VARIABLE THROUGH ITS POINTER

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer? This is done by using another unary operator * (asterisk). Usually known as the indirection operator. Another name for the indirection operator is the dereferencing operator. Consider the following statements:

```
int quantity, *p, n;  
quantity = 179;  
p = &quantity;  
n = *p;
```

The first line declares quantity and n as integer variables and p as a pointer variable pointing to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to the pointer variable **p**. The fourth line contains the indirection operator *. When the operator* is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer

value is the address. In this case, ***p** returns the value of the variable **quantity**, because **p** is the address of **quantity**. The ***** can be remembered as ‘value at address’. Thus the value of **n** would be 179. The two statements

```
p = &quantity;
```

```
n = *p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

```
n = quantity;
```

In C, the assignment of pointers and addresses is always done symbolically, by means of symbolic names. You cannot access the value stored at the address 5368 by writing ***5368**. It will not work. Program 11.2 illustrates the distinction between pointer value and the value it points to.

Program 2.2 Write a program to illustrate the use of indirection operator ‘*****’ to access the value pointed to by a pointer.

The program and output are shown in Fig.2.5 the program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
x = *(&x) = *ptr = y
```

```
&x = &*ptr
```

Program

```
main()
```

```
{
```

```
    int x,y;
```

```

int *ptr;

x = 10;

ptr = &x;

y = *ptr;

printf("value of x is %d\n\n", x);

printf("%d is stored at addr %u\n", x, &x);

printf("%d is stored at addr %u\n", *&x, &x);

printf("%d is stored at addr %u\n",*ptr, ptr);

printf("%d is stored at addr %u\n", ptr, &ptr);

printf("%d is stored at addr %u\n",y, &y);

*ptr = 25;

printf("\nNow x = %d\n", x);

}

```

Output

```

Value of x is 10

10    is stored at addr 4104

10    is stored at addr 4104

10    is stored at addr 4104

4104  is stored at addr 4106

10    is stored at addr 4108

Now x = 25

```

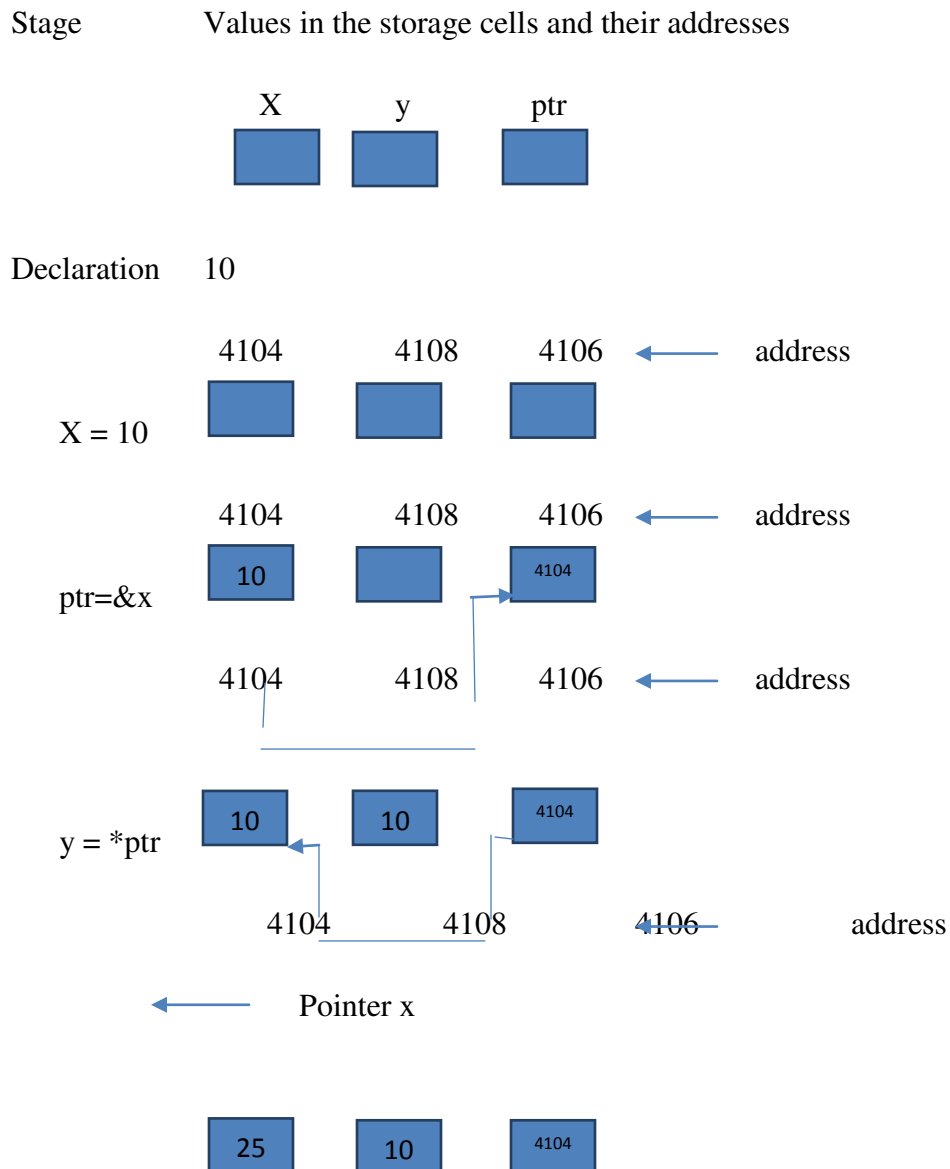
Fig. 2.5 Accessing a variable through its pointer

The Actions performed by the program are illustrated in Fig. 2.6. The statement **ptr = &x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y**.

Note the use of the assignment statement

***ptr = 25;**

This statement puts the value of 25 at the memory location whose address is the value of **ptr**. We know that the value of **ptr** is the address of **x** and therefore, the old value of **x** is replaced by 25. This, in effect, is equivalent to assigning 25 to **x**. This shows how we can change the value of a variable indirectly using a pointer and indirection operator.



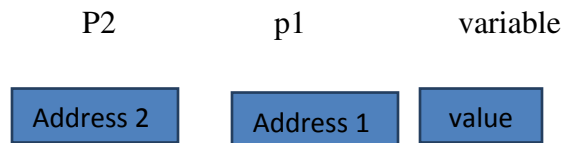
*ptr=25.

4104 4108 4106 ← address

Fig. 2.6 illustration of pointer assignments

2.7 CHAIN OF POINTERS

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown.



Here, the pointer variable p2 contains the address of the pointer variable p1, which points to the location that contains desired value. This is known as multiple indirections.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. example:

```
int **p2;
```

This declaration tells the compiler that p2 is a pointer to a pointer of int type. Remember, the pointer p2 is not a pointer to an integer, but rather a pointer to an integer pointer.

We can access the target value indirectly pointed to by applying the indirection operator twice. Consider the following code:

```
main ()
{
    int x, *p1, **p2;
    x = 100;
    p1 = &x;      /* address of x */
    p2 = &p1;      /* address of p1 */
```

```

    printf("%d", **p2);
}

```

This code will display the value 100. Here, p1 is declared as a pointer to an integer and p2 as a pointer to a pointer to an integer.

2.8 POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. For example, if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```

y= *p1 * *p2;      same as (*p1) * (*p2)

sum = sum + *p1;

z = 5* - *p2/*p1;  same as (5 * (-(*p2)))/(*p1)

*p2 = *p2 + 10;

```

Note that there is a blank space between / and * in the item 3 above. The following is wrong.

```
z = 5* - *p2 / *p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. P1 + 4, p2-2 and p1-p2 are all allowed. If p1 and p2 are both pointers to the same array, then p2 – p1 gives the number of elements between p1 and p2.

We may also use short-hand operators with the pointers.

```

p1++;

-p2;

sum += *p2;

```

in addition to arithmetic operations discussed above, pointers can also be compared using relational operators. The expressions such as p1 > p2, p1 == p2, and p1 != p2 are allowed.

However, any comparison of pointers that refer to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.

We may not use pointers in division or multiplication. For example, expressions such as

$p1/p2$ or $p1 * p2$ or $p1/3$

Are not allowed. Similarly, two pointers cannot be added. That is, $p1 + p2$ is illegal.

Program 2.3 Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig.2.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

$4 * - *p2 / *p1 + 10$

Is evaluated as follows:

$((4 * (- *p2)) / (*p1)) + 10$

When $*p1 = 12$ and $*p2 = 4$, this expression evaluates to 9. Remember, since all the variables are of type int, the entire evaluation is carried out using the integer arithmetic.

Program

main()

{

int a, b, *p1, *p2, x, y, z;

a = 12;

b = 4;

p1 = &a;

p2 = &b;

x = *p1 * *p2 / *p1 + 10;

```
printf("address of a = %u\n", p1);  
printf("address of b = %u\n", p2);  
printf("\n");  
printf("a = %d, b = %d\n", a, b);  
printf("x = %d, y = %d\n", x, y);  
*p2 = *p2 + 3;  
*p1 = *p2 - 5;  
z = *p1 * *p2 - 6;  
printf("\na= %d, b = %d", a, b);  
printf('z = %d\n', z);  
}
```

Output

Address of a = 4020

Address of b = 4016

a = 12, b = 4

x = 42, y = 9

a = 2, b = 7, z = 8

Fig.2.7 Evaluation of pointer expressions

2.9 POINTER INCREMENTS AND SCALE FACTOR

We have seen that the pointers can be incremented like

```
P1 = P2 + 2;
```

```
P1 = P1 + 1;
```

And so on. Remember, however, an expression like

```
P1++;
```

Will cause the pointer p1 to point to the next value of its type. For example, if p1 is an integer pointer with an initial value, say 2800, then after the operation `p1 = p1 + 1`, the value of p1 will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the length of the data type that it points to. This length called the scale factor.

For an IBM PC, the length of various data types are as follows:

characters	1 bytes
integers	2 bytes
floats	4 bytes
long integers	4 bytes
doubles	8 bytes

The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, then **sizeof(x)** returns the number of bytes needed for the variable. (Systems like Pentium use 4 bytes for storing integers and 2 bytes for short integers).

RULES OF POINTER OPERATIONS

The following Rules apply when performing operations on pointer variables.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variables can be assigned the values of another pointer variable.

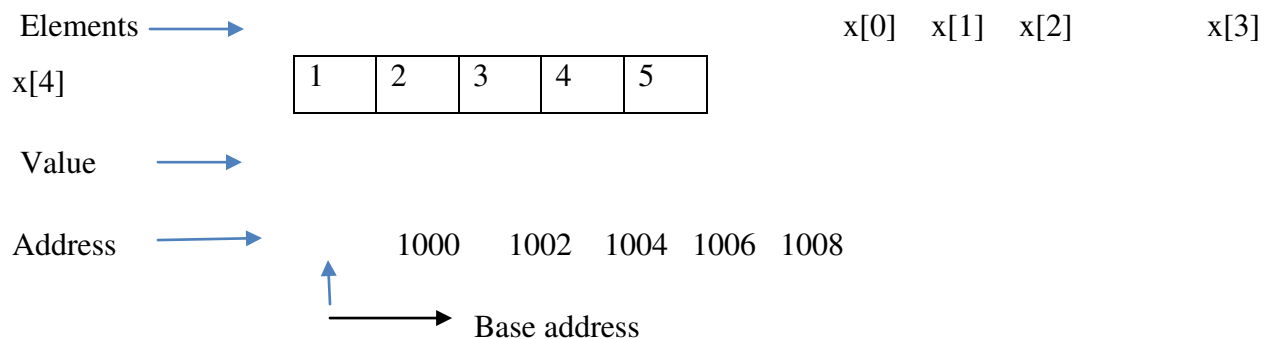
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variables can be pre-fixed or post-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the some array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data types, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an arbitrary address (i.e., `&x = 10;` is illegal).

2.10 POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array `x` as follows:

```
int x[5] = {1,2,3,4,5};
```

Suppose the base address of `x` is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:



The name `x` is defined as a constant pointer pointing to the first element, `x[0]` and therefore the value of `x` is 1000, the location where `x[0]` is stored. That is,

$$x = \&x[0] = 1000$$

If we declare `p` as an integer pointer, then we can make the pointer `p` to point to the array `x` by the following assignment.

$$p = x;$$

This equivalent to

$$p = \&x[0];$$

Now, we can access every value of `x` using `p++` to move from one element to another. The relationship between `p` and `x` is shown as:

$$p = \&x[0](=1000)$$

$$p+1 = \&x[1](=1002)$$

$$p+2 = \&x[2](=1004)$$

$$p+3 = \&x[3](=1006)$$

$$p+4 = \&x[4](=1008)$$

you may notice that the address of an element is calculated using its index and the scale factor of the type. For instance,

$$\text{address of } x[3] = \text{base address} + (3 * \text{scale factor of int})$$

$$= 1000 + (3 * 2) = 1006$$

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that `*(o+3)` gives the value of `x[3]`. The pointer accessing method is much faster than array indexing.

The program 2.4 illustrates the use of pointer accessing method.

Program 2.4 Write a program using pointer to compute the sum of all elements stored in an array.

The program shown in Fig.2.8 illustrates how pointer can be used to traverse an array element. Since incrementing an array pointer causes it to point to the next element, we need only to add one to p each time we go through the loop.

Program

```
main()
{
    int *p, sum, i;

    int x[5] = {5, 9, 6, 3, 7};

    i = 0;

    p = x; / initializing with base address of x */

    printf('Element value Address\n\n');

    while (i < 5)
    {

        printf(" x[%d] %d %u\n", i, *p, p);

        sum = sum + *p;    /* accessing array element */

        i++, p++;        /* incrementing pointer*/

    }

    printf('\n sum = %d\n', sum);

    printf('\n &x[0] = %u\n', &x[0]);
```

```

        printf("\n p = %u\n", p);
    }

```

Output

Element	value	Address
x[0]	5	166
x[1]	9	168
x[2]	6	170
x[3]	3	172
x[4]	7	174

sum = 55
&x[0] = 166
p = 176

Fig. 2.8 Accessing one- dimensional array elements using the pointer.

It is possible to avoid the loop control variable i as shown:

```

.....

    p = x;

    while (p <= &x[4])

    {

        sum += *p;

        p++;

    }

```

.....

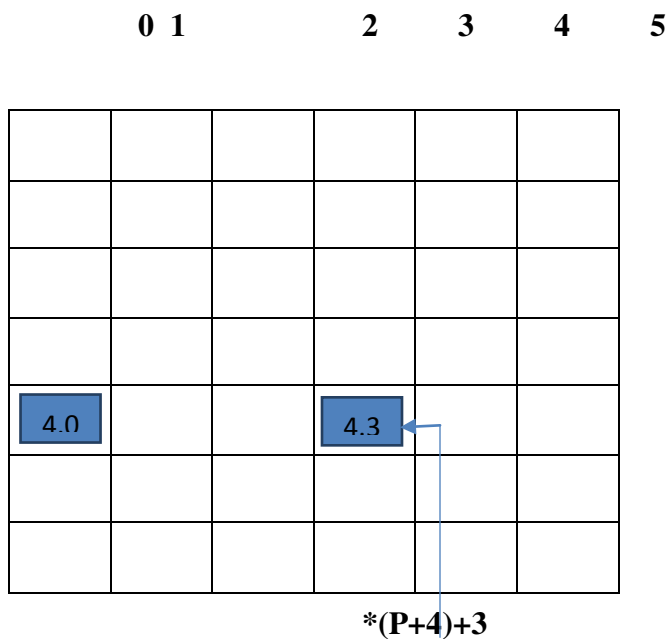
Here, we compare the pointer p with the address of the last element to determine when the array has been traversed.

Pointer can be used to manipulate two-dimensional arrays as well. We know that in a one-dimensional array x , the expression

$$*(x+i) \text{ or } *(p+i)$$

Represents the element $x[i]$. Similarly, an element in a two-dimensional array can be represented by the pointer expressions as follows:

$$***(a+i)+j \text{ or } ***(p+i)j$$



p \longrightarrow pointer to first row

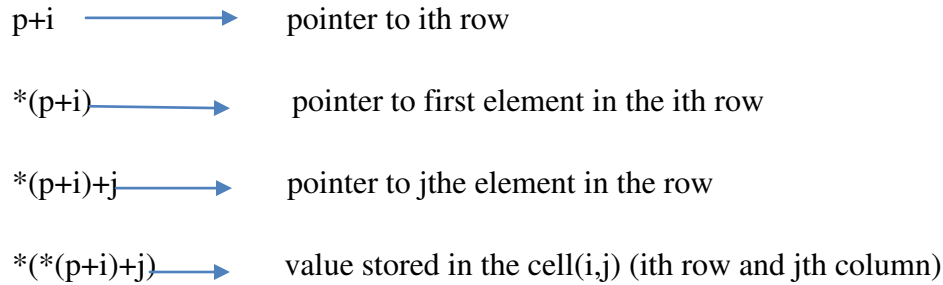


Fig.2.9 Pointer to two-dimension arrays

Fig 2.9 illustrates how this expression represents the element $a[i][j]$. The base address of the array a is $\&a[0][0]$ and starting at this address, the compiler allocates contiguous space for all the elements row-wise. That is, the first element of the second row is placed immediately after the last element of the first row, and so on. Suppose we declare an array as follows:

```
int a[3][4] = { {15,27,11,35},
               {22,19,31,17},
               {31,23,14,36}};
```

The elements of a will be stored as:

15	27	11	35	22	19	31	17	31	23	14	36
----	----	----	----	----	----	----	----	----	----	----	----

If we declare p as an int pointer with the initial address of $\&a[0][0]$, then

$$A[i][j] \text{ is equivalent to } *(p+4\times i+j)$$

You may notice that, if we increment i by 1, the p is incremented by 4, the size of each row. Then the element $a[2][3]$ is given by $*(p+2\times 4+3) = *(p+11)$.

This is the reason why, when a two-dimensional array is declared, we must specify the size of each row so that the compiler can determine the correct storage mapping.

2.11 POINTERS AND CHARACTER STRINGS

We have seen in chapter that strings are treated like character arrays and therefore, they are declared and initialized as follows:

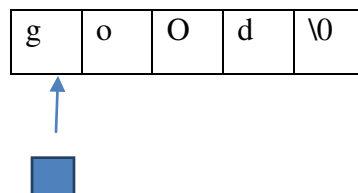
```
char str[5] = "good";
```

The compiler automatically inserts the null character '\0' at the end of the string. C supports an alternative method to create strings using pointer variables of type char. Example:

```
char *str = "good";
```

This creates a string for the literal and then stores its address in the pointer variable str.

The pointer str now points to the first character of the string "good" as:



we can also use the run-time assignment for giving values to a string pointer. Example

```
char * string1;
```

```
string1 = "good";
```

note that the assignment

```
string1 = "good";
```

is not string copy, because the variables **string1** is a pointer, not a string.

(as pointer out in chapter 8, C does not support copying one string to another through the assignment operation.)

We can print the content of the string string1 using either printf or puts function as follows;

```
printf("%s", string1);
```

```
puts(string1);
```



```

{
char *name;

int length;

    char *cptr = name;

    name = "name";

    printf("%s\n", name);

while(*cptr != '\0')
{
    printf("%c is stored at address %u\n", *cptr, cptr);

    cptr++;

}

    length = cptr - name;

    printf("\nlength of the string = %d\n", length);

}

```

Output

DELHI

D is stored at address 54

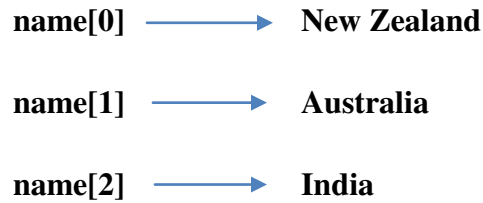
E is stored at address 55

L is stored at address 56

H is stored at address 57

I is stored at address 58

Declares name to be an array of three pointers to characters, each pointer pointing to a particular name as:



This declaration allocates only 28 bytes, sufficient to hold all the characters as shown

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	I	a	\0						

The following statement would print out all the three names:

```

for(i=0; i <= 2; i++)
printf(“%s\n”, name[i]);

```

To access the jth character in the ith name, we may write as

```

*(name[i]+j)

```

The character arrays with the rows of varying length are called ‘ragged arrays’ and are better handled by pointers.

Remember the difference between the notations ***p[3]** and **(*p)[3]**. Since * has a lower precedence than [], ***p[3]** declares p as an array of 3 pointers while **(*p)[3]** declares p as a pointer to an array of three elements.

2.13 POINTERS AS FUNCTION ARGUMENTS

We have seen earlier that when an array is passed to a function as an *argument*, only the address of the first element of the array is passed, but not the actual values of the array elements. If x is an array, when we call sort(x), the address of x[0] is passed to the function sort. The function uses this address for manipulating the array elements. Similarly, we can pass the

address of a variable as an argument to function in the normal fashion. We used this method when discussing functions that return multiple values (see chapter 9).

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling function using pointers to pass the addresses of variables is known as *'call by reference'*. (You know, the process of passing the actual value of variables is known as *'call by value'*.) the function which is called by *'reference'* can change the value of the variable used in the call.

Consider the following code:

```
main()
{
    int x;

    x = 20;

    change(&x);      /* call by reference or address */

    printf("%d\n", x);
}

change(int *p)
{
    *p = *p + 10;
}
```

When the function **change()** is called, the address of the variable **x**, not its value, is passed into the function **change()**, the variable **p** is declared as a pointer and therefore **p** is the address of the variable **x**. the statement,

```
*p = *p + 10;
```

Means ‘add 10 to the value stored at the address **p**’. since **p** represents the address of **x**, the value of **x** is changed from 20 to 30. Therefore, the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function. Note that this mechanism is also known as “**call by address**” or “pass by pointers”.

Program 2.6 Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig.2.11 shows how the contents of two location can be exchanged using their address locations. The function **exchange()** receives the addresses of the variables **x** and **y** and exchange their contents.

Program

```
Void exchange (int *, int *); /* prototyhpe */
```

```
main()
{
    int x, y;

    x = 100;

    y = 200;

    printf("Before exchange : x = %d y = %d\n\n", x, y);

    exchange(&x, &y); /* call*/

    printf("After exchange : x = %d y = %d\n\n", x, y);
}

exchange (int *a, int *b)
{
```



```

    int t;

    t = *a; /* Assign the value at address a to t*/

    *a = *b;    /* put b int a */

    *b = t; /* put t int b*/

}

```

Output

Before exchange: x = 100 y = 200

After exchange : x = 200 y = 100

Fig. 2.11 Passing of pointers as function parameter

You may not the following points:

1. The function parameters are declared as pointers.
2. The dereferenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

The use of pointer to access array elements is very common in C . We have used a pointer to traverse array elements in program 11.4. We can also use this technique in designing user-defined function discussed in chapter 9. Let us consider the problem sorting an array of integers discussed in program 9.6. The function **sort** may be written using pointers(instead of array indexing) as shown:

```

void sort(int m, int *x)

{

    int i, j, temp;

    for(i=1; i<= m-1; i++)

        for (j=1; j<=m-1; j++)

```

```

        if (*(x+j-1) >= *(x+j))
        {
            temp = *(x+j-1);
            *(x+j-1) = *(x+j);
            *(x+j) = temp;
        }
    }

```

Note that we have used the pointer `x` (instead of array `x[]`) to receive the address of array passed and therefore the pointer `x` can be used to access the array elements (as pointed out in section 2.10)

This function can be used to sort an array of integers as follows:

```

.....

int score[4] = {45, 90, 71, 83};

.....

sort(4, score); /*function call*/

.....

```

the calling function must use the following prototype declaration.

```

void sort(int, int *);

```

This tells the compiler that the formal argument that receives the array is a pointer, not array variable. Pointer parameters are commonly employed in string function. Consider the function `copy` which copies one string to another.

```

copy (char *s1, char *s2)
{

```

```
while (*s1++ = *s2++) != '\0');  
}
```

This copies the contents of s2 into the string s1 parameters s1 and s2 are the pointers to character strings, whose initial values are passed from the calling function. For example, the calling statement

```
copy(name1, name2);
```

will assign the address of the first element of **name1** to **s1** and the address of the first element of **name2** to **s2**. Note that the value of ***s++** is character that **s2** pointed to before **s2** was incremented. Due to the postfix ++, **s2** is incremented only after the current value has been fetched. Similarly, **s1** is incremented only after the assignment has been completed.

Each character, after it has been copied, is compared with '\0' and therefore copying is terminated as soon as the '\0' is copied.

Program 2.7 The program of Fig.2.12 shows how to calculate the sum of two numbers which are passed as arguments using the call by reference method.

Program

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

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

```
Void swap (int *p, *q);
```

```
main()
```

```
{
```

```
int x = 0;
```

```
in y = 20;
```

```
clrstr();
```

```

    printf("\n value of x and y before swapping are x = %d and y = %d", x, y);

    swap(&x, &y);

    printf("\n\nvalue of x and y after swapping are x = %d and y = %d", x,y);

    getch();

}

void swap(int *p, int *q)
{
    int r;

    r = *p;

    *p = *q;

    *q = r;

}

```

Output

value of x and before swapping are x = 10 and y = 20

value of x and y after swapping are x = 20 and y =10

Fig. 2.12 Program to pass the arguments using call by reference method

2.14 FUNCTION RETURNING POINTERS

We have seen so far that a function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in C, we can also force a function to return a pointer to the calling function. Consider the following code:

```

int *larger(int *, int *);

main()

```

```

{
    int a= 10;

    int b = 20;

    int *p;

    p = larger (&a, &b); /*function call*/

    printf("%d", *p);
}

int *larger(int *x, int *y)

{
    if(*x>*y)

        return(x); /* address of a */

    else

        return(y); /* address of b */

}

```

The function **larger** receives the addresses of the variables **a** and **b**, decides which one is larger using the pointers **x** and **y** and then return the address of its location. The returned value is then assigned to the pointer variable **p** in the calling function. In this case, the address of **b** is returned and assigned to **p** and therefore the output will be the value of **b**, namely, 20.

Note that the address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

2.15 POINTERS TO FUNCTIONS

A function, like a variable, has a type and an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

Type (*fptr());

This tells the compiler that **fptr** is a pointer to a function, which return type value. The parentheses around ***fptr** are necessary. Remember that a statement like

Type *gptr();

Would declare **gptr** as a function returning a pointer to type.

We can make a function pointer to point a specific function by simply assigning the name of function to the pointer. For example, the statements

```
double mul(int, int);
```

```
double(*p1());
```

```
p1 = mul;
```

Declare **p1** as a point to function and **mul** as a function and then make **p1** to point to the function **mul**. To call the function **mul**, we may now use the pointer **p1** with list of parameters. That is

```
(*p1)(x,y) /*function call*/
```

is equivalent to

```
mul(x,y)
```

Note the parentheses around ***p1**.

Program 2.8 Write a program that uses a function pointer as a function argument.

A program to print the function values over a given range of values shown in Fig. 2.13. The printing is done by the function table by evaluating the function passed to it by the main.

With **table**, we declare the parameter f as a pointer to a function as follows:

```
double (*f)();
```

The value returned by the function is of type **double**. When **table** is called in the statement

```
table (y, 0.0, 2, 0.5);
```

We pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list.

During the execution of **table**, the statement

```
value = (*f)(a);
```

Calls the function **y** which is pointed to by **f**, passing it the parameter **a**. Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

```
table(cos, 0.0, PI, 0.5);
```

Passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

Program

```
#include<math.h
```

```
#define PI 3.1415926
```

```
double y(double);
```

```
double cos(double);
```

```
double table(double(*f)(),double, double, double);
```

```
main()
```

```
{
```

```

printf("Table of y(x) = 2*x* x-x+1\n\n");

table(y, 0.0, 2.0, 0.5);

printf("\nTable of cos(x)\n\n");

table(cos,0.0,,Pi,0.5);

}

double table(double(*f)(),double min, double max, double step)

{

    double a, value;

    for(a=min; a<= max; a+= step)

        {

            value = (*f)(a);

            printf("%5.2f %10.4\n", a, value);

        }

}

double y(double x)

{

    return(2*x*x-x+1);

}

```

Output

Table of $y(x) = 2*x*x-x+1$

0.0	1.0000
0.50	1.0000

1.00	2.0000
------	--------

1.50	4.0000
------	--------

2.00	7.0000
------	--------

Table of cos(x)

0.0	1.0000
-----	--------

0.50	0.8776
------	--------

1.00	0.5403
------	--------

1.50	0.0707
------	--------

2.00	-0.4161
------	---------

2.50	-0.8011
------	---------

3.00	-0.9900
------	---------

Fig 2.13 Use of pointers to functions

COMPATIBILITY AND CASTING

A variable declared as a pointer is not just a pointer type variable. It is also a pointer to a specific fundamental data type, such as a character. A pointer therefore always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as *incompatibility of pointers*.

All the pointer variables store memory addresses, which are compatible, but what not compatibility is is the underlying data type to which they point to. We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using cast operator, as was do with the fundamental types. Example:

```
int x;
```

```
char *p;
```

```
p = (char *) & x;
```

In such cases, we must ensure that all operations that use the pointer **p** must apply casting properly.

We have an exception. The exception is the void pointer (void*). The void pointer is a generic pointer that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

```
void *vp;
```

Remember that since a void pointer has no object type, it cannot be de-referenced.

2.16 POINTERS AND STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is array variable of **struct type**. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
struct inventory  
  
{  
  
    char name[30];  
  
    int number;  
  
    float price;  
  
} product[2], *ptr;
```

This statement declares **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to data objects data of type **struct inventory**. The assignment

```
ptr = product;
```

would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product[0]**. Its members can be accessed using the following notation.

Ptr → **name**

Ptr → **number**

Ptr → **price**

The symbol → is called the *arrow* operator (also known as member selection operator) and is made up of a minus sign and a greater than sign. Note that **ptr →** is simply another way of writing **product[0]**.

When the pointer **ptr** is incremented by one, it is made to point to the next record, i.e., **product[1]**. The following for statement will print the values of members of all the elements of **product** array.

```
for(ptr = product; ptr < product+2; ptr++)  
    printf("%s %d %f\n", ptr → name, ptr → number, ptr → price);
```

we could also use the notation

```
(*ptr).number
```

To access the member **number**. The parentheses around ***ptr** are necessary because the member operator **'.'** has a higher precedence than the operator *****.

Program 2.9 Write a program to illustrate the use of structure pointers.

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 2.14. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent**) is also used as the loop control index in for loops.

Program

```
struct invent  
{
```

```

char *name[20];

int number;

float price;

};

main()
{

    struct invent product[3], *ptr;

    printf("input\n\n");

    for(ptr = product; ptr < product+3; ptr++)

        scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);

    printf("\noutput\n\n");

    ptr = product;

while(ptr < product + 3)
{

    printf("%-20s %5d %10.2f\n", ptr->name,

            ptr->number,

            ptr->price);

    ptr++;

}

}

```

Output

INPUT

Washing_machine 5 7500

Electric_iron 12 350

Two_in_one 7 1250

OUTPUT

Washing_machine 5 7500.00

Electric_iron 12 350.00

Two_in_one 7 1250.00

Fig. 2.14 pointer to structure variables

While using structure pointer, we should take care of the precedence of operators.

The operators ‘ \rightarrow ’ and ‘.’, and [] enjoy the highest priority among the operators. They bind very tightly with their operands. For example, given the definition

```
struct
{
    int count;
    float *p;    /* pointer inside the struct */
} ptr;
```

Then the statement

```
++ptr→count;
```

Increments **count**, not **ptr**. However,

(++ptr) →count;

Increments **ptr** first, and then links **count**. The statement

ptr++→count;

Is legal and increments **ptr** after accessing **count**.

The following statements also behave in the similar fashion.

***ptr→p** Fetches whatever **p** points to.

***ptr→p++** increments **p** after accessing whatever it points to.

(*ptr→p)++ increments whatever **p** points to.

***ptr++→p** increments **ptr** after accessing whatever it points to.

In the previous chapter, we discussed about passing of a structure as an argument to a function. We also saw an example where a function receives a copy of an entire structure and returns it after working on it. As we mentioned earlier, this method is inefficient in terms of both, the execution speed and memory. We can overcome this drawback by passing a pointer to the structure and then using this pointer to work on the structure members. Consider the following function:

```
print_invent(struct invent *item)  
{  
printf("name: %s\n", item→name);  
printf("price: %f\n", item→price);  
}
```

This function can be called by

```
print_invent(&product);
```

The formal argument item receives the address of the structure product and therefore it must be declared as a pointer of type **struct invent**, which represents the structure of **product**.

11.17 TROUBLES WITH POINTERS

Pointers give us tremendous power and flexibility. However, they could become a nightmare when they are not used correctly. The major problem with wrong use of pointers is that the compiler may not detect the error in most cases and therefore the program is likely to produce unexpected results. The output may not give us any clue regarding the use of a bad pointer. Debugging therefore becomes a difficult task.

We list here some pointer errors that are more commonly committed by the programmers.

- Assigning values to uninitialized pointers

```
int * p, m = 100;
*p = m;           /*error*/
```

- Assigning value to a pointer variable

```
int *p, m = 100;
p = m;           /* error*/
```

- Not dereferencing a pointer when required

```
int *p, x = 100;
p = &x;
printf("%d", p); /* error*/
```

- Assigning the address of an uninitialized variables

```
int m, *p
p = &m;           /* error */
```

- Comparing pointers that point to different objects

```
char name1 [ 20 ], name2 [ 30 ];  
  
char *p1 = name1;  
  
char *p2 = name2;  
  
if(p1 > p2)..... /*error*/
```

We must be careful in declaring and assigning values to pointers correctly before using them. We must also make sure that we apply the address operator & and referencing operator * correctly to the pointers. That will save us from sleepless nights.

3 FILE MANAGEMENT IN C

Key terms

File|Name|ftell |rewind| fseek |command line |argument

3.1 INTRODUCTION

Until now we have been using the functions such as **scanf** and **printf** to read and write data. These are console oriented I/O functions, which always use the terminal (keyword and screen) as the target place. This works fine as long as the data is small. However, many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read wherever necessary, without destroying the data. This method employs the concept of files to store data. A file is a place on the disk where a group of related data is stored. Like most other languages, C supports a number of functions that have the ability to perform basic file operations, which include:

- Naming a file,

- Opening a file,
- Reading data from a file,
- Writing data to a file, and
- Closing a file.

There are two distinct ways to perform file operations in C. The first one is known as the low-level I/O and uses UNIX system calls. The second method is referred to as the high-level I/O operation and uses functions in C's standard I/O library. We shall discuss in this chapter, the important file handling functions that are available in the C library. They are listed in table 12.1.

There are many other functions. Not all of them are supported by all compilers. You should check your C library before using a particular I/O function.

3.2 DEFINING AND OPERNING A FILE

If we want to store data in a file in the secondary memory, we must specify certain things should the file, to the operating system. They include:

Table 3.1 High level I/O Functions

Function name	Operation
fopen()	*Creates a new file for use *Opens an existing file for use.
fclose()	*Closes a file which has been opened for use.
getc()	*Reads a character from a file
putc()	*Writes a character to a file.
fprintf()	*Writes a set of data values to a file.
fscanf()	*Reads a set of data values from a file.
getw()	*Reads an integer from a file.
putw()	*Writes an integer to a file.
fseek()	*Sets the position to a desired point in the
ftell()	file.
rewind()	*Gives the current position in the file. *Sets the position to the beginning of the file.

1. File name.
2. Data structure
3. Purpose.

Filename is a string of characters that make up a valid file name for the operating system. It may contain two parts, a primary name and an optional period with the extension. Example:

input.data

store

PROG.C

Student.c

Text.out

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all file should be declared as type FILE before they are used. **FILE** is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file.

```
FILE *fp;
```

```
fp = fopen("filename", "mode");
```

The first statement declares the variable **fp** as a “pointer to the data type FILE”. As stated earlier, FILE is a structure that is defined in the I/O library. The second statement opens the file named file name and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job. Mode can be one of the following;

- r open the file for reading only.
- w open the file for writing only.
- a open the file for appending(or adding) data to it.

Note that both the filename and mode are specified as string. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen:

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

Consider the following statements:

```
FILE *p1, *p2;  
  
p1 = fopen("data", "r");  
  
p2 = fopen("results", "w");
```

The file data is opened for reading and results is opened for writing. In case, the results file already exists, its contents are deleted and the file is opened as a new file. If data file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They include:

- r+** The existing file is opened to the beginning for both reading and writing.
- w+** Same as **w** except both for reading and writing.

a+ Same as **a** except both for reading and writing.

We can open and use a number of files at a time. This number however depends on the system we use.

3.3 CLOSING A FILE

A file must be closed as soon as all operations on it have been completed.; this ensure that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form:

```
fclose(file_pointer);
```

This would close the file associated with the FILE pointer `file_pointer`. Look at the following segment of a program.

```
.....  
  
.....  
  
FILE *p1, *p2;  
  
p1 = fopen("INPUT", 'W');  
  
P2 = fopen('OUTPUT', 'r');  
  
.....  
  
.....  
  
fclose(p1);  
  
fclose(p2);  
  
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

As a matter of fact all files are closed automatically wherever a program terminates. However, closing a file as soon as you are done with it is a good programming habit.

3.4 INPUT/OUTPUT OPERATIONS ON FILES

Once a file is opened, reading out of or writing to it is accomplished using the standard I/O routines that are listed in Table 3.1

The **getc** and **putc** Functions

The simplest file I/O functions are **getc** and **putc**. These are analogous to **getchar** and **putchar** functions and handle one character at a time. Assume that a file is opened with mode *w* and file pointer *fp1*. Then, the statement

```
putc(c, fp1);
```

Writes the character contained in the character variable *c* to the file associated with FILE pointer **fp1**. Similarly, **getc** is used to read a character from a file that has been opened in read mode. For example, the statement

```
C = getc(fp2);
```

Would read a character from the file whose file pointer is **fp2**.

The file pointer moves by one character position for every operation of **getc** or **putc**. The **getc** will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

Program 3.1 Write a program to read data from the keyboard, write it to a file called **INPUT**, again read the same data from the **INPUT** file, and display it on the screen.

A program and the related input and output data are shown in Fig.3.1. We enter the input data via the keyboard and the program writes it, character by character, to the file **INPUT**. The end of the data is indicated by entering an EOF character, which is control-z in the reference system. (This may be control-D in other systems.) The file **INPUT** is closed at this signal.

Program

```
#include<stdio.h>

main()
{
    FILE *f1;

    char c;

    printf("Data Input\n\n");

    /* Open the file INPUT*/

    f1 = fopen("INPUT", 'W');

    /* Get a character from keyboard */

while ((c = getchar()) != EOF)

    /* Write a character to INPUT*/

    putchar(c,f1);

    /* Close the file INPUT*/

    fclose(f1);

    printf('\n Data output\n\n');

    /* Reopen the file INPUT*/

    f1 = fopen("INPUT", 'r');

    /* Read a character from INPUT*/

While ((c=getc(f1)) != EOF)
```

```

        /* Display a character on screen */

        printf("%c", c);

        /* Close the file INPUT */

        fclose(f1);

    }

```

Output

Data input

This is a program to test the file handling

Features on this system^Z.

Data output

This is a program to test the file handling

Features on this system.

Fig. 3.1 Character oriented read/write operations on a file.

The file INPUT is again reopened for reading. The program then reads its content character by character, and displays it on the screen. Reading is terminated when `getc` encounters the end-of-file mark EOF.

Testing for the end-of-file condition is important. Any attempt to read past the end of the file might either cause the program to terminate with an error or result in an infinite loop situation.

The `getw` and `putw` functions

The `getw` and `putw` are integer-oriented functions. They are similar to the `getc` and `putc` functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of `getw` and `putw` are

```

putw(integer, fp);

```

getw(fp);

Program 3.2 illustrates the use of **putw** and **getw** functions

Program 3.2 A file named DATA contains a series of integer numbers. Code a program to read these numbers and then write all odd' numbers to a file to be called ODD and all even' numbers to a file to be called EVEN.

The program is shown in Fig. 3.2. it uses three files simultaneously and therefore, we need to define three-file pointer **f1**, **f2**, and **f3**.

putw(number, f1);

Notice that when we type -1, the reading is terminated and the file is closed. The next step is to open all the three files, **DATA** for reading, **ODD** and **EVEN** for writing. The contents of **DATA** files are read, integer by integer, by the function **getw(f1)** and written to **ODD** or **EVEN** file after an appropriate test.

Note that the statement

(number = getw(f1)) != EOF

Reads a value, assigns the same to number, and then tests for the end-of-file mark.

Finally, the program displays the contents of odd and even files. It is important to note that the files **ODD** and **EVEN** opened for writing are closed before they are reopened for reading.

Program

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

```
main()
```

```
{
```

```
FILE *f1, *f2, *f3;
```

```
int number, I;
```



```

printf("contents of DATA file\n\n");

f1 = fopen("DATA", "w");      /* Create DATA file*/

for(i=1; i<= 30; i++)

{

scanf("%d", &numbr);

if(number== -1) break;

putw(number, f1);

}

fclose(f1);

f1 = fopen("DATA","r");

f2 = fopen("ODD", "w");

f3 = fopen("EVEN", "w");

/* Read form Data file*/

While ((number = getw(f1)) != EOF)

{

if(number %2 == 0)

putw(number, f3); /* Write to Even file*/

else

putw(number, f2); /* write to odd file */

}

fclose(f1);

```

```

fclose(f2);

fclose(f3);

f2 = fopen("ODD", "r");

f3 = fopen("EVEN", "r");

printf("\n\nContents of Odd file \n\n");

while((number = getw(f2)) != EOF)

printf("%4d", number);

printf("\n\nContents of EVEN file \n\n");

while((number = getw(f3)) != EOF)

printf("%4d", number);

fclose(f2);

fclose(f3);

}

```

Output

Contents of DATA file

```

111  222  333  444  555  666  777  888  999  000 121 232 343 454 565
-1

```

Contents of ODD file

```

111  333  555  777  999  121  343  565

```

Contents of EVEN file

```

222  444  666  888  0  232  454

```

Fig. 3.2 Operations on integer data

The **fprintf** and **fscanf** Functions

So far, we have seen functions that can handle only one character or integer at a time. Most compilers support two other functions, namely **fprintf** and **fscanf** that can handle a group of mixed data simultaneously.

The functions **fprintf** and **fscanf** perform I/O operations that are identical to the familiar **printf** and **scanf** functions, except of course that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of **fprintf** is

```
fprintf(fp, "control string", list);
```

where **fp** is a file pointer associated either a file that has been opened for writing. The control string contains output specifications for the items in the list. The list may include variables, constants and strings. Example:

```
fprintf(f1, "%s %d %f", name, age, 7.5);
```

here, **name** is an array variable of type **char** and **age** is an **int** variable.

The general format of **fscanf** is

```
fscanf(fp, "control string", list);
```

This statement would cause the reading of the items in the list from the file specified by **fp**, according to the specifications contained in the control string. Example:

```
fscanf(f2, "%s %d", item, &quantity);
```

This statement would cause the reading of the items in the list from the file specified by **fp**, according to the specifications contained in the control string. Example:

```
fscanf(f2, "%s %d", item, &quantity);
```

Like **scanf**, **fscanf** also returns the number of items that are successfully read. When the end of the file is reached, it returns the value EOF.

Program 3.3 Write a program to open a file named **INVENTORY** and store in it the following data:

Item name	Number	Price	quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

The program is given in Fig.3.2 The filename INVENTORY is supplied through the keyboard. Data is read using the function **fscanf** from the file **stdin**, which refers to the terminal and it is then written to the file that is being pointed to by the file pointer **fp**. Remember that the file pointer **fp** points to the file INVENTORY.

After closing the file INVENTORY, it is again reopened for reading. The data from the file, along with the item values are written to the file **stdout**, which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file....e.

Program

```
#include<stdio.h>

main()
{
    FILE *fp;

    int number, quantity, I;

    float price, value;

    char item[10], filename[10];

    printf("Input file name\n");
```

```

scanf("%s", filename);

fp = fopen(filename, "w");

printf("Input inventory data \n\n");

printf("Item name Number Price Quantity \n");

for(i=1; i<=3; i++)

{

    fscanf(stdin, "%s %d %f %d", item, &number, &price, &quantity);

    fprintf(fp, "%s %d %.2f %d", item, number, price, quantity);

}

fclose(fp);

fprintf(stdout, "\n\n");

fp = fopen(filename, "r");

printf("Item name Number Price quantity Value\n");

for(I = 1; I <= 3; i++)

{

    fscanf(fp, "%s %d %f %d", item, &number, &price, &quantity);

    value = price * quantity;

    fprintf(stdout, "%-8s %7d %8.2f %8d %11.2f\n", item, number, price,
    quantity, value);
}

```

```

    }
    fclose(fp);
}

```

Output

Input file name

INVENTORY

Input inventory data

Item name	Number	Price	quantity	
AAA-1	111	17.50	115	
BBB-2	125	36.00	75	
C-3	247	31.75	104	
Item name	Number	Price	quantity	Value
AAA-1	111	17.50	115	2012.50
BBB-2	125	36.00	75	2700.00
C-3	247	31.75	104	3302.00

Fig. 3.3 Operations on mixed data types

3.5 ERROR HANDLING DURING I/O OPERATORS

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.

4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. Fortunately, we have two status-inquiry library functions; **feof** and **ferror** that can help us detect I/O errors in the files.

The **feof** function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If **fp** is pointer to file that has just been opened for reading, then the statement

```
if(feof(fp))  
  
printf("End of data.\n");
```

would display the message "End of data" on reaching the end of file condition.

The **ferror** function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. it returns zero otherwise. The statement

```
if(ferror(fp) != 0)  
  
printf("An error has occurred.\n");
```

would print the error message, if the reading is not successful.

We know that whenever a file is opened using **fopen** function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not. Example:

```
if(fp == NULL)
```

```
printf("File could not be opened.\n");
```

Program 3.4 Write a program to illustrate error handling in file operations.

The program shown in Fig. 3.4 illustrates the use of the **NULL** pointer test and **feof** function. When we input filename as TETS, the function call

```
fopen("TETS", "r");
```

returns a **NULL** pointer because the file TETS does not exist and therefore the message "Cannot open the file" is printed out.

Similarly, the call **feof(fp2)** returns a non-zero integer when the entire data has been read, and hence the program prints the message "Ran out of data" and terminates further reading.

Program

```
#include<stdio.h>

main()
{

    char *filename;

    FILE *fp1, *fp2;

    int I, number;

    fp1 = fopen("TEST", "w");

    for(i = 10; I <= 100; I += 10)

        putw(i, fp1);

    fclose(fp1);

    printf("\n Input filename\n");

    open_file;
```



```
scanf("%s", filename);

if((fp2 = fopen(filename, "r")) == NULL)
{
printf("Cannot open the file.\n");
printf("Type file again.\n\n");
goto open_file;
}

else
for(I = 1; i <= 20; i++)
{
number = getw(fp2);
if(feo(fp2))
{
printf("\n Ran out of data.\n");
break;
}

else
printf("%d\n", number);
}

fclose(fp2);
}
```

Output

```
Input filename
TETS
Cannot open the file.
Type filename again.
TEST
10
20
30
40
60
70
80
90
100
Ran out of data.
```

Fig. 3.4 Illustration of error handling in file operations

3.6 RANDOM ACCESS TO FILES

So far we have discussed file functions that are useful for reading and writing data sequentially. There are occasions, however, when we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the functions **fseek**, **ftell**, and **rewind** available in the I/O library.

ftell takes a file pointer and return a number of type long, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:

```
n = ftell(fp);
```

n would give the relative offset (in bytes) of the current position. This means that **n** bytes have already been read (or written).

rewind takes a file pointer and resets the position to the start of the file. For example, the statement

```
rewind(fp);
```

```
n = ftell(fp);
```

would assign 0 to **n** because the file position has been set to the start of the file by **rewind**. Remember, the first byte in the file is numbered as 0, second as 1 and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a **rewind** is done implicitly.

fseek function is used to move the file position to a desired location within the file. It takes the following form:

```
fseek(file_ptr, offset, position);
```

file_ptr is a pointer to the file concerned, *offset* is a number or variable of type long, and *position* is an integer number. The offset specifies the number of positions (bytes) to be moved from the location specified by *position*. The *position* can take one of the following three values:

value	meaning
0	Beginning of file
1	Current position
2	End of file

The offset may be positive, meaning move forwards, or negative, meaning move backwards. Examples in Table 3.2 illustrate the operations of the **fseek** function:

Table 3.2 Operations of **fseek** Function

Statement	Meaning
<code>fseek(fp,0l,0);</code>	Go to the Beginning (Similar to rewind)
<code>fseek(fp,0l,1);</code>	Stay at the current position. (Rarely used)
<code>fseek(fp,0l,2);</code>	Go to the end of the file, past the last character of the file.
<code>fseek(fp,m,0);</code>	Move to (m+1)th byte in the file.
<code>fseek(fp,m,1);</code>	Go forward by m bytes.
<code>fseek(fp,-m,1);</code>	Go backward by m bytes from the current position.
<code>fseek(fp,-m,2);</code>	Go backward by m bytes from the end.(Positions the file to the mth character from the end.)

When the operation is successful, `fseek` returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and `fseek` returns -1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

Program 3.5 Write a program that uses the functions **ftell** and **fseek**

A program employing **ftell** and **fseek** functions is shown in Fig.3.5. We have created a file **RANDOM** with the following contents:

```

Positions    →    0    1    2    .....    25

Character

stored      →    A    B    C            Z

```

We are reading the file twice. First, we are reading the content of every fifth position and printing its value along with its position on the screen. The second time, we are reading contents of the file from the end and printing the same on the screen.

During the first reading, the file pointer crosses the end-of-file mark when the parameter `n` of `fseek(fp,n,0)` becomes 30. Therefore, after printing the content of position 30, the loop is terminated.

For reading the file from the end, we use the statement

```
fseek(fp,-1L,2);
```

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This is achieved by the function

```
fseek(fp, -2L, 1);
```

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

Program

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

```
main()
```

```
{
```

```
FILE *fp;
```

```
long n;
```

```
char c;
```

```
fp = fopen("RANDOM", "w");
```

```
while((c = getchar()) != EOF)
```

```

        putc(c,fp);

printf("No. of characters entered = %d\n", ftell(fp));

fclose(fp);

fp = fopen("RANDOM", "r");

n = 0L;

while(feof(fp) == 0)
{
    fseek(fp, n,0); /* Position to (n+1)th character*/

    printf("Position of %c is %d\n", getc(fp),ftell(fp));

    n = n+5L;
}

    putchar('\n');

    fseek(fp, -=1L,2); /* Position to the last Character */

do
{
    putchar(getc(fp));
}

    while(!fseek(fp, -2L,1));

    fclose(fp);
}

```

Output

ABCDEFGHIJKLMNOPQRSTUVWXYZ

No. of characters entered = 26

Position of A is 0

Position of F is 5

Position of K is 10

Position of P is 15

Position of U is 20

Position of Z is 25

ZYXWVUTSRQPONMLKJIHGFEDCBA

Fig. 3.5 Illustration **fseek** and **ftell** functions

Program 3.6 Write a program to append additional items to the file INVENTORY created in Program 3.3 and print total contents of the file.

The program is shown in Fig 3.6 It uses a structure definition to describe each item and a function append () to add an item to the file.

On execution, the program requests for the filename to which data is to be appended. After appending the items, the position of the last character in the file is assigned to n and then the file is closed.

The file is reopened for reading and its contents are displayed. Note that reading and displaying are done under the control of a while loop. The loop tests the current file position against n and is terminated when they become equal.

Program

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

```

struct invent_record
{
    char name[10];

    int number;

    float price;

    int quantity;
};

main()
{
    struct invent_record item;

    char filename[10];

    int response;

    FILE *fp;

    long n;

    void append(struct invent_record *x, file *y);

    printf("Type filename:");

    scanf("%s", filename);

    fp = fopen(filename, "a+");

do
{
    append(&item, fp);

```



```

    printf("\nItem %s appended.\n", item.name);

    printf("\n Do you want to add another item\ (1 for YES /0 for NO)?");

    scanf("%d", &response);

}

while (response == 1);

n = ftell(fp);      /* position of last character */

fclose(fp);

fp = fopen(filename, "r");

while(ftell(fp) < n)

{

    fscanf(fp, "%s %d %f %d",

    item.name, &item.number, &item.price, &item.quantity);

    fprintf(stdout, "%-8s %7d %8.2f %8d\n",

    item.name, item.number, item.price, item.quantity);

}

fclose(fp);

}

void append(struct invent_record *product, File *ptr)

{

    printf("Item name:");

    scanf("%s", product → name);

```

```

printf("Item number:");

scanf("%d", &product→number);

printf("Item price:");

scanf("%f", &product→price);

printf("Quantity:");

scanf("%d", &product→quantity);

fprintf(ptr, "%s %d %.2f %d",

product → name

product→number

product→price

product→quantity);

}

```

Output

Type file name: INVENTORY

Item name: XXX

Item number : 444

Item Price : 40.50

Quantity : 34

Item XXX appended.

Do you want to add another item (1 for YES /0 for NO)?1

Item name: YYY

Item number : 555

Item Price : 50.50

Quantity : 45

Item YYY appended

Do you want to add another item (1 for YES /0 for NO)?0

AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104
XXX	444	40.50	34
YYY	555	50.50	45

Program 3.7 Write a C program to reverse the first n character in a file. The file name and the value of n are specified on the command line. Incorporate validation of arguments, that is, the program should check that the number of arguments passed and the value of n that are meaningful.

Program

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

void main(int argc, char *argv[])

{

FILE *fs;

char str[100];
```

```

int i, n, j;

if (argc != 3)          /*checking the numbe of arguments given at command line*/
{

puts(“İmproper number of arguments.”);

exit(0);

}

n = atoi(argv[2]);

fs = fopen(argv[1], “r”);    /* opening the source file in read mode*/

if(fs==NULL)
{

printf(“Soruce file cannot be opened.”);

exit(0);

}

I = 0;

while(1)
{

if(str[i] = fgetc(fs) != EOF) /* Reading contents of file character by character */

j = i+1;

else

break;

}

```

```

fclose(fs);

fs = fopen(argv[1], "w");           /*opening the file in write mode*/

if(n<0 | n>strlen(str))

{

printf("Incorrect value of n. Program will terminate ....\n\n");

getch();

exit(1);

}

j = strlen(str);

for(i= 1; i<= n; i++)

{

fputc(str[j],fs);

j-;

}

fclose(fs);

printf("\n%d characters of the file successfully printed in reverse order", n);

getch();

}

```

Output

D:\TC\BIN\program source.txt 5

5 characters of the file successfully printed in reverse order

Fig. 3.7 Program to reverse n characters in a file.

3.7 COMMAND LINE ARGUMENTS

What is a command line argument? It is a parameter supplied to a program when the program is invoked. This parameter may represent a file name the program should process. For example, if we want execute a program to copy the contents of a file named X_FILE to another one named Y_FILE, then may use a command line like

```
C > PROGRAM X_FILE Y_FILE
```

are PROGRAM is the file name where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the filenames during execution. How do these parameters get into the program?

We know that every C program should have one main function and that it marks the beginning the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact main can take two arguments called argc and argv and the information contained in the command line is passed on to the program through these arguments, when main is called up by the system.

The variable **argc** is an argument counter that counts the number of arguments on the command line. The **argv** is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line given above, **argc** is three and **argv** is an array of three pointers to strings as shown below:

```
argv[0]→PROGRAM
```

```
argv[1] →X_FILE
```

```
argv[2]→Y_FILE
```

In order to access the command line arguments, we must declare the main function and its parameters as follows:

```
main(int arge, char *argv[])
```

```
{
```

```
.....  
.....  
}
```

The first parameter in the command line is always the program name and therefore **argv[0]** always represents the program name.

Program 3.8 Write a program that will receive a filename and a line of text as command line arguments and write the text to the file.

Fig 3.8 shows the use of command line arguments. The command line is

```
F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEE EFFFFFF GGGGGG
```

Each word in the command line is an argument to the main and therefore the total number of arguments is 9.

The argument vector **argv[1]** points to the string **TEXT** and therefore the statement

```
fp = fopen(argv[1], "w");
```

opens a file with the name **TEXT**. The for loop that follows immediately writes the remaining 7 arguments to the file **TEXT**.

Program

```
#include<stdio.h>  
  
main(int arge, char *argv[])  
  
{  
  
    FILE *fp;  
  
    int I;  
  
    char word[15];
```

```

fp = fopen(argv[1], "w"); /* open file with name argv[1] */

printf("\n No. of arguments in command line = %d \n\n", argc);

for(i = 2; i < argc; i++)

fprintf(fp, "%s", argv[i]); /* write to file argv[1] */

fclose(fp);

/* Writing content of the file to screen */

printf("Contents of %s file\n\n", argv[1]);

fp = fopen(argv[1], "r");

for(i = 2; i < argc; i++)

{

fscanf(fp, "%s", word);

printf("%s", word);

}

fclose(fp);

printf("\n\n");

/*Writing the arguments from memory */

for ( i= 0; i < argc; i++)

printf("%*s \n", i*5, argv[i]);

}

```

Output

F12_7 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEE EFFFFFF GGGGGG

No. of arguments in Command Line = 9

Contents of TEXT file

AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFFFF GGGGGG

C:\CF12_7.EXE

TEXT

AAAAAA

BBBBBB

CCCCCC

DDDDDD

EEEEEE

FFFFFF

GGGGGG

Fig. 3.8 Use of Command line arguments.

4 DYNAMIC MEMORY ALLOCATION AND LINKED LISTS

KEY TERMS

Dynamic memory allocation | stack, Heap | Linked list | Size of operators | malloc function | calloc function| realloc function | Null pointer

4.1 INTRODUCTION

Most often we face situations in programming where the data is dynamic in nature. That is, the number of data items keep changing during execution of the program. For example, consider a program for processing the list of customers of a corporation. The list grows when names are added and shrinks when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items. Situations can be handled more

easily and effectively by using what is known as *dynamic data structures* in conjunction with dynamic memory management techniques.

Dynamic data structures provide flexibility in adding, deleting or rearranging data items at run time. Dynamic memory management techniques permit us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space. This chapter discusses the concept of linked lists, one of the basic types of dynamic data structures. Before we take up linked lists, we shall briefly introduce the dynamic storage management functions that are available in C. These functions would be extensively used in processing linked lists.

4.2 DYNAMIC MEMORY ALLOCATION

C Language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgment of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory at run time is known as dynamic memory allocation. Although C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution. They are listed in Table 4.1. These functions help us build complex application programs that use the available memory intelligently.

Table 4.1 Memory Allocation Functions

Function	Task
malloc	Allocates request size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes And then returns a pointer to the memory.
free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

MEMORY ALLOCATION PROCESS

Before we discuss these functions, let us look at the memory allocation process associated with a C program. Fig. 4.1 shows the conceptual view of storage of a C program in memory.

Local variables
Free memory
Global variables
C program instructions

The program instructions and global and static variables are stored in a region known as permanent storage area and the local variables are stored in another area called stack. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called the heap. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory “overflow” during dynamic allocation process. In such situations, the memory allocation functions mentioned above return a NULL pointer (when they fail to locate enough memory requested).

4.3 ALLOCATING A BLOCK OF MEMORY: MALLOC

A block of memory may be allocated using the function **malloc**. The **malloc** function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr = (cast-type *) malloc(byte-size);
```

ptr is a pointer of type cast-type. The **malloc** returns a pointer (or cast-type) to an area of memory with size byte-size.

Example:

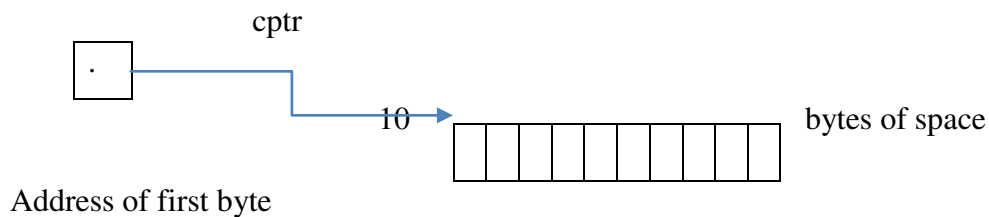
```
x = (int *) malloc (100 *sizeof(int));
```

On successful execution of this statement, a memory space equivalent to “100 times the size of an **int**” bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer **x** of type of **int**.

Similarly, the statement

```
cptr = (char*) malloc(10);
```

allocates 10 bytes of space for the pointer **cptr** of type **char**. This is illustrated as:



Note that the storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.

We may also use **malloc** to allocate space for complex data types such as structures. Example:

```
st_var = (struct store *) malloc(sizeof(struct store));
```

where, **st_var** is a pointer of type **struct store**.

Remember, the **malloc** allocates a block of contiguous bytes. The allocation can fail if the space in the heap is not sufficient to satisfy the request. If it fails, it returns a **NULL**. We should therefore check whether the allocation is successful before using the memory pointer. This is illustrated in the program in Fig. 4.2.

Program 4.1 Write a program that uses a table of integers whose size will be specified interactively at run time.

The program is given in Fig.4.2. It tests for availability of memory space of required size. If it is available then the required space is allocated and the address of the first byte of the space allocated is displayed. The program also illustrates the use of pointer variable for storing and accessing the table values.

Program

```
#include<stdio.h>

#include<stdlib.h>

#define NULL 0

main()
{
    int *p, *table;

    int size;

    printf("\n What is the size of table?");

    scanf("%d", size);

    printf("\n")

    /*-----Memory allocation-----*/

    if(table = (int*)malloc(size *sizeof(int))) == NULL)
    {
        printf("No space available \n");

        exit(1);
    }

    printf("\n Address of the first byte is %u\n", table);

    /*Reading table values*/

    printf("\n Input table values \n");

    for(p = table; p<table + size; p++)

        scanf("%d", p);
```

```

        /*printing table values in reverse order */
        for(p = table + size -1; p>= table; p--)
            printf("%d is stored at address %u \n", *p, p);
    }

```

Output

What is the size of the table? 5

Address of the first byte is 2262

Input table values

11 12 13 14 15

15 is stored at address 2270.

14 is stored at address 2268.

13 is stored at address 2266.

12 is stored at address 2264.

11 is stored at address 2262.

Fig. 4.2 Memory allocation with **malloc**

4.4 ALLOCATING MULTIPLE BLOCKS OF MEMORY: CALLOC

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of **calloc** is:

```
ptr = (cast-type *) calloc (n, elem-size);
```

The above statement allocates contiguous space for n blocks, each size **elem-size** bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

The following segment of a program allocates space for a structure variable:

```
.....  
  
.....  
  
struct student  
  
{  
  
char name[25];  
  
float age;  
  
long int id_num;  
  
};  
  
typedef struct student record;  
  
record *st_ptr;  
  
int class_size = 30;  
  
st_ptr = (record *) calloc(class_size, sizeof(record));  
  
.....  
  
.....
```

record is of type **struct** student having three members. **name** **age** and **id_num**. The **calloc** allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the **st_ptr**. This may be done as follows:

```
if(st_ptr == NULL)
```

```
{  
  
    printf("Avaialable memory not sufficient");  
  
    exit(1);  
  
}
```

4.5 RELEASING THE USED SPACE: FREE

Compile-time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited.

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function:

```
free (ptr);
```

ptr is a pointer to a memory block, which has already been created by **malloc** or **calloc**. Use of an invalid pointer in the call may create problems and cause system crash. We should remember two things here:

1. It is not the pointer that is being released but rather what it points to.
2. To release an array of memory that was allocated by **calloc** we need only to release the pointer once. It is an error to attempt to release elements individually.

The use of **free** function has been illustrated in program 4.2.

4.6 ALTERING THE SIZE OF A BLOCK: REALLOC

It is likely that we discover later, the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it. In both the cases, we can change the memory size already allocated with the help of the function **realloc**. This process is called the reallocation of memory. For example, if the original allocation is done by the statement


```
ptr = malloc(size);
```

then reallocation of space may be done by the statement

```
ptr = realloc(ptr, newsize);
```

This function allocates a new memory space of size `newsize` to the pointer variable `ptr` and returns a pointer to the first byte of the new memory block. The `newsize` may be larger or smaller than the `size`. Remember, the new memory block may or may not begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.

If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed (lost). This implies that it is necessary to test the success of operation before proceeding further. This is illustrated in the program of program 4.2.

Program 4.2 Write a program to store a character string in a block of memory space created by `malloc` and then modify the same to store a larger string.

The program is shown in Fig. 4.3. The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remains same even after modification of the original size.

Program

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

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

```
#define NULL 0
```

```
main()
```

```
{
```

```
    char *buffer;
```

```

    /* Allocating memory */

    if((buffer = (char *)malloc(10)) == NULL)

    {

    printf("malloc failed.\n");

    exit(1);

    }

    printf("Buffer of size %d created \n", _msize(buffer));

    strcpy(buffer, "HYDERABAD");

    printf("\nBuffer contains: %s \n", buffer);

    /* Reallocation */

    if((buffer = (char *)realloc(buffer, 15)) == NULL)

    {

    printf("Reallocation failed.\n");

    exit(1);

    }

    printf("\n Buffer size modified.\n");

    printf("\nBuffer still contains: %s \n", buffer);

    strcpy(buffer, "SECUNDERABAD");

    printf("\n Buffer now contains: %s \n", buffer);

    /* Freeing memory */

}

```

Output

Buffer of size 10 created.

Buffer contains: HYDERABAD

Buffer size modified

Buffer still contains: HYDERABAD

Buffer now contains: SECUNDERABAD

Fig.4.3 Reallocation and release of memory space.

4.7 CONCEPTS OF LINKED LISTS

We know that a list refers to a set of items organized sequentially. An array is an example of list. In an array, the sequential organization is provided implicitly by its index. We use the index for accessing and manipulation of array elements. One major problem with the arrays is that the size of an array must be specified precisely at the beginning. As pointer out earlier, this may be a difficult task in many real-life applications.

A completely different way to represent a list is to make each item in the list part of a structure that also contains a “link” to the structure containing the next item, as shown in Fig. 4.4. This type of list is called a *linked list* because it is a list whose order is given by links from one item to the next.

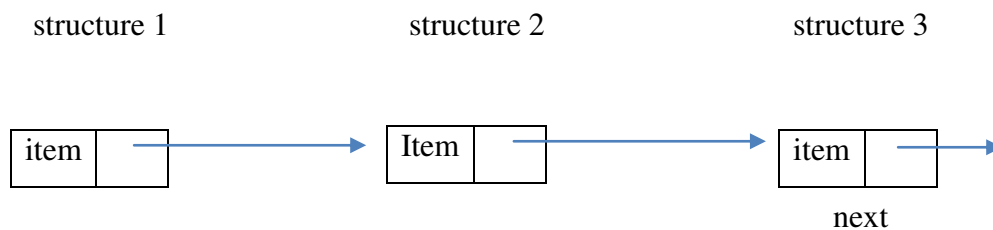


Fig. 4.4 A linked list

Each structure of the list is called a node and consists of two fields, one containing the item, and the other containing the address of the next item (a pointer to the next item) in the list. A linked list is therefore a collection of structures ordered not by their physical placement in

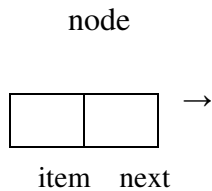
memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows:

```
struct node
{
int item;

struct node *next;

};
```

The first member is an integer item and the second a pointer to the next node in the list as shown below. Remember, the item is an integer here only for simplicity, and could be any complex data type.



Such structures, which contain a member field that points to the same structure type are called self-referential structures.

A node may be represented in general form as follows:

```
struct tag-name
{
type member1;

type member2;

.....

.....
```

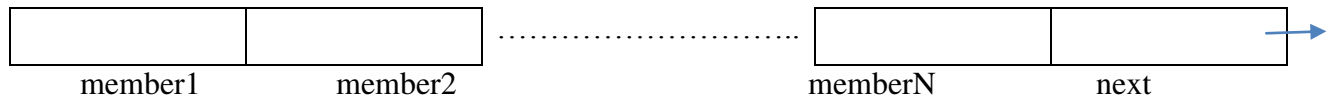
```

struct tag-name *next;

};

```

The structure may contain more than one item with different data types. However, one of the items must be a pointer of the type tag-name.



Let us consider a simple example to illustrate the concept of linking. Suppose we define a structure as follows:

```

struct link_list
{
float age;

struct link_list *next;

};

```

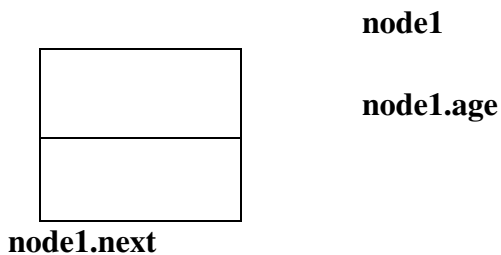
For simplicity, let us assume that the list contains two nodes node1 and node2. They are of type **struct link_list** and are defined as follows:

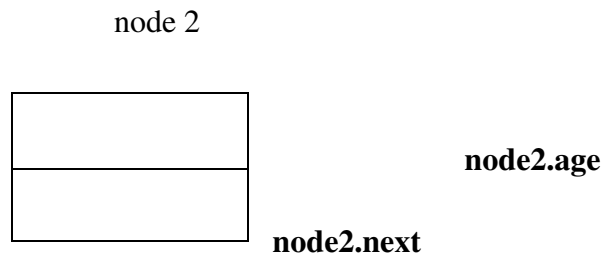
```

struct link_list node1, node 2;

```

This statement creates space for two nodes each containing two empty fields as shown:

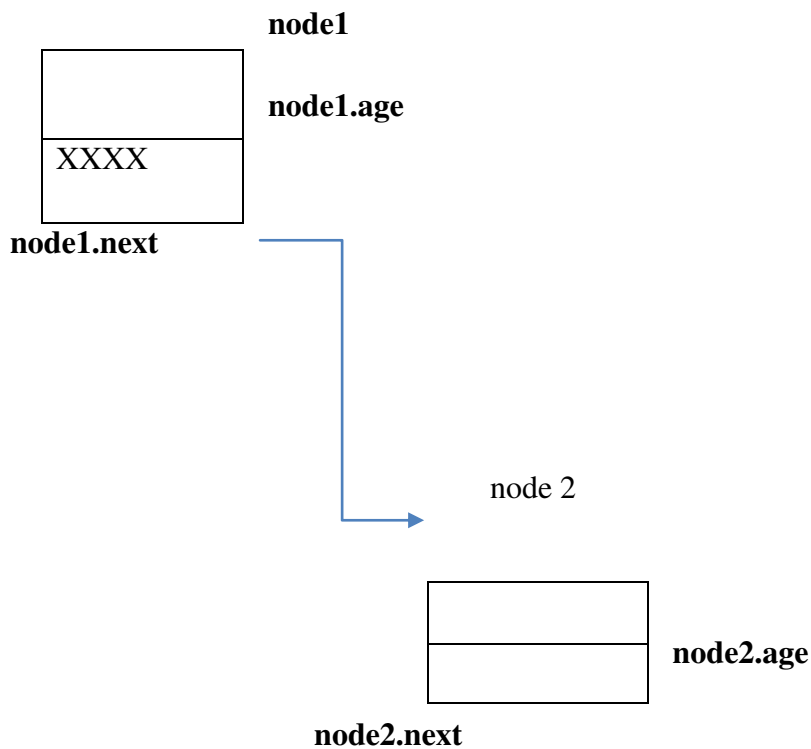




The **next** pointer of **node1** can be made to point to **node2** by the statement

node1.next = &node2;

This statement stores the address of **node2** into the field **node1.next** and thus establishes a “link” between **node1** and node2 as shown:

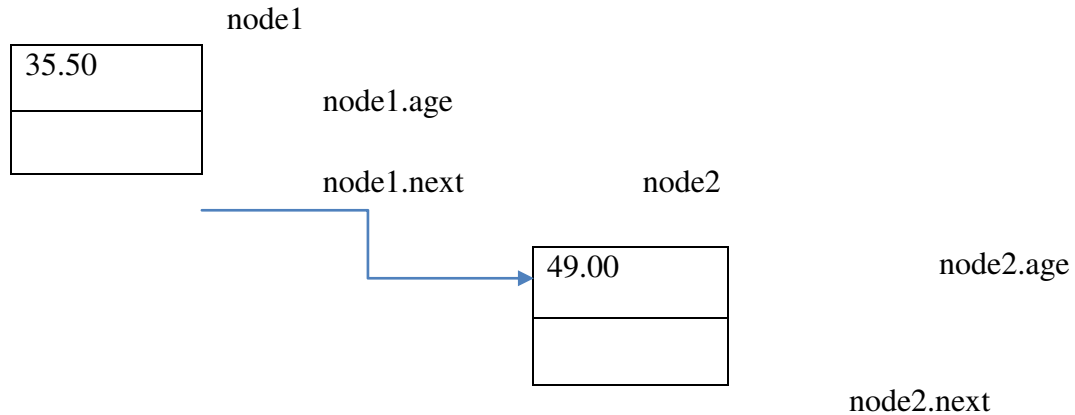


“XXXX” is the address of node2 where the value of the variable node2.age will be stored. Now let us assign values to the field age.

node1.age = 35.50;

node2.age = 49.00

The result is as follows:



We may continue this process to create a linked list of any number of values.

For example:

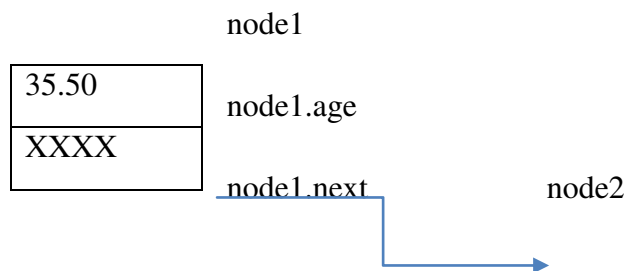
node2.next = &node3;

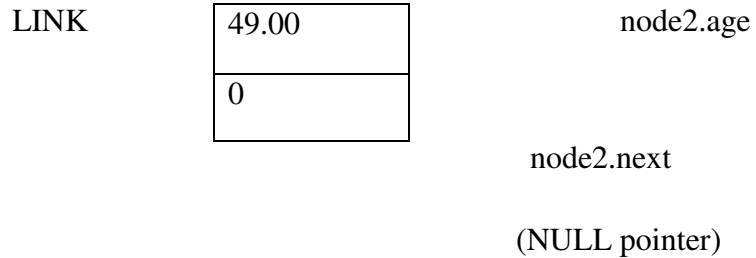
would add another link provided node3 has been declared as a variable of type **struct link list**.

Note list goes on forever. Every list must have an end. We must therefore indicate the end of a linked list. This is necessary for processing the list. C has a special pointer value called **null** that can be store in the next field of the last node. In our two-node list, the end of the list is marked as follows:

node2.next = 0;

The final linked list containing two nodes is as shown:





The value of the age member of **node2** can be accessed using the **next** member of **node1** as follows:

```
printf("%f\n", node.next->age);
```

4.8 ADVANTAGES OF LINKED LISTS

A Linked list is dynamic data structure. Therefore, the primary advantage of linked lists over arrays is that linked lists can grow or shrink in size during the execution of a program. A linked list can be made just as long as required.

Another advantage is that a linked list does not waste memory space. It uses the memory that is just needed for the list at any point of time. This is because it is not necessary to specify the number of nodes to be used in the list.

The third, and the most important advantage is that the linked lists provide flexibility is allowing the items to be rearranged efficiently. It is easier to insert or delete items by rearranging the links. This is shown in Fig. 4.5.

The major limitation of linked lists is that the access to any arbitrary item is little cumbersome and time consuming. Whenever we deal with affixed length list, it would be better to use an array rather than a linked list. We must also note that a linked list will use more storage than an array with the same number of items. This is because each item has an additional link field.

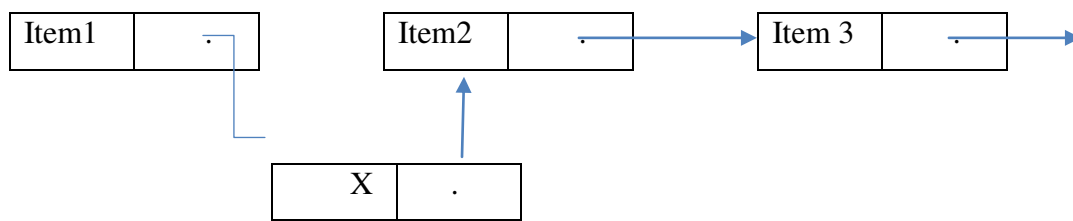
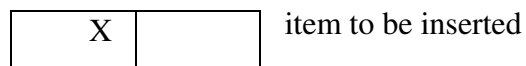
4.9 TYPES OF LINKED LISTS

There are different types of lined lists. The one we discussed so far is known as *linear singly* linked list. *The other* linked lists are:

- Circular linked lists.

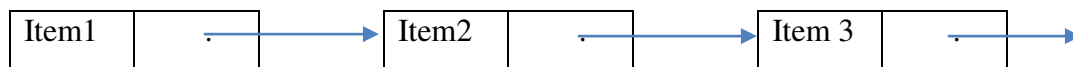
- Two-way or doubly linked lists.
- Circular doubly linked lists.

The circular linked lists have no beginning and no end. The last item points back to the first item. The doubly linked list uses double set of pointers, one pointing to the next item and other pointing to the preceding item. This allows us to traverse the list in either direction. Circular doubly linked lists employs both the forward pointer and backward pointer in circular form. Fig 4.6 illustrates various kinds of linked lists.

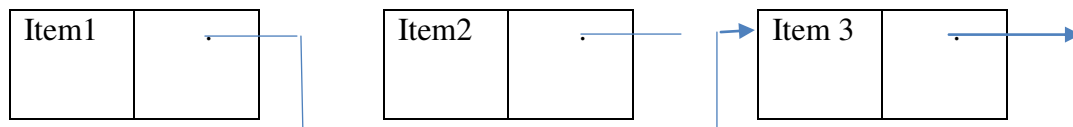


(a) Insertion

(A record is created holding the new item and its next pointer is set to link it to the item, which is to follow it in the list. The next pointer of the item which is to precede it must be modified to point to the new item.)



item to deleted



(b) Deletion

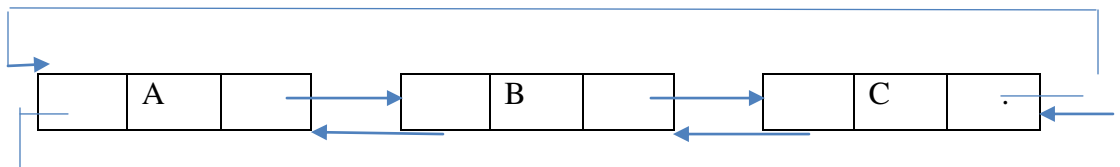
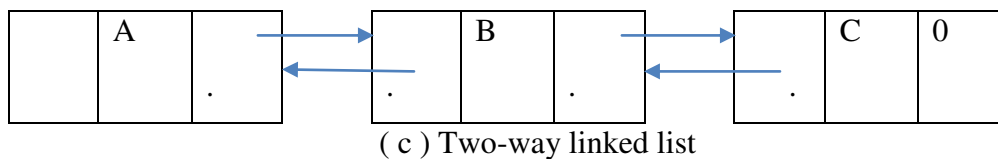
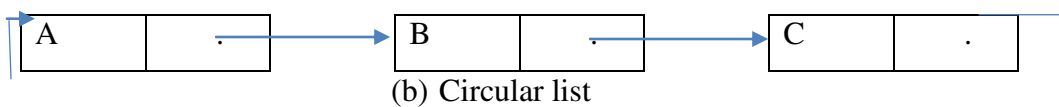
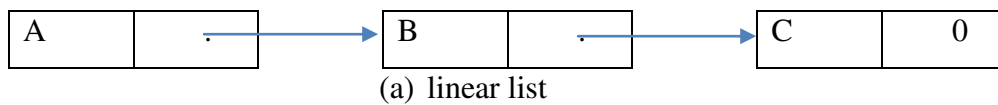
(The next pointer of the item immediately preceding the one to be deleted is attend and made to point to the item following the deleted item.)

Fig 4.5 Insertion into and deletion from a linked list

4.10 POINTERS REVISITED

The pointers are used extensively in processing of the linked lists, we shall briefly review some of their properties that are directly relevant to the processing of lists.

We know that variables can be declared as pointers, specifying the type of data item they can point to. In effect, the pointer will hold the address of the data item and can be used to access its value. In processing linked lists, we mostly use pointers of type structures.



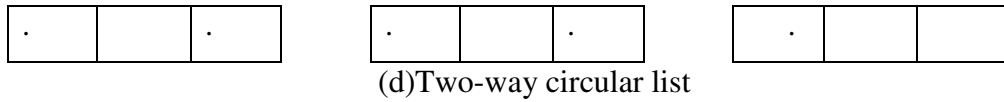
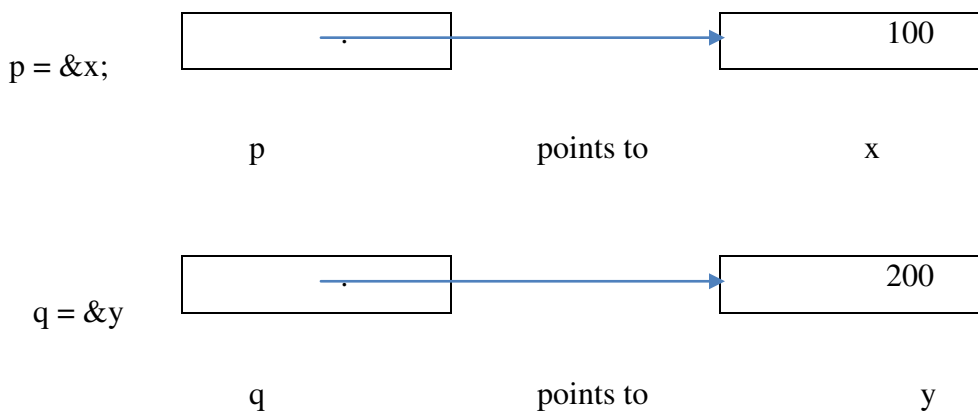


Fig.4.6 Different types of linked lists

It is most important to remember the distinction between the pointer variable **ptr**, which contain the address of a variable, and the referenced variable ***ptr**, which denotes the value of variable to which **ptr**'s value points. The following examples illustrate this distinction. In these illustrations, we assume that the pointers **p** and **q** and the variables **x** and **y** are declared to be of same type.

(a) Initialization

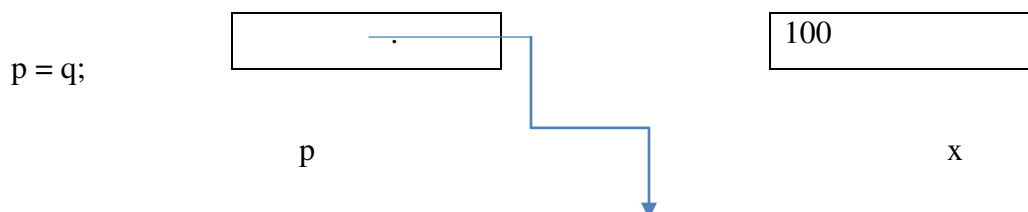


The pointer **p** contains the address of **x** and **q** contains the address of **y**.

***p = 100 and *q = 200 and p <> q**

(b) Assignment $p = q$

The assignment $p = q$ assigns the address of the variable **y** to the pointer variable **p** and therefore **p** now points to the variable **y**.



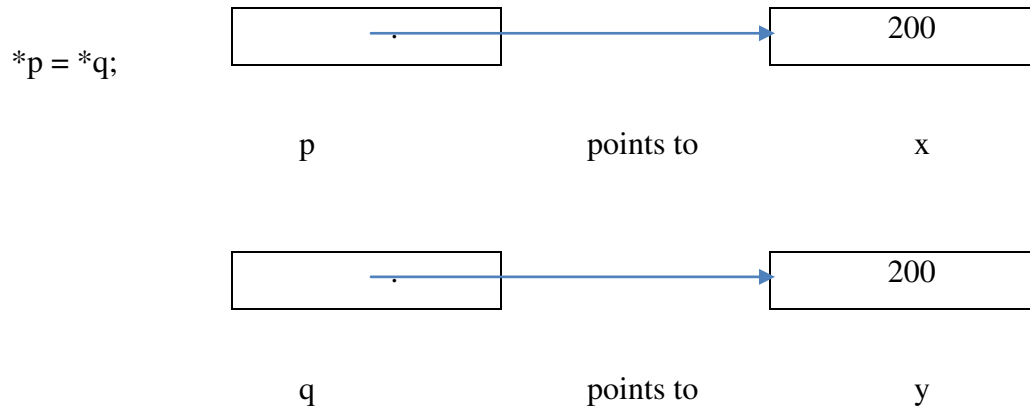


Both the pointer variables point to the same variable.

***p = *q = 200 but x <> y**

(c) Assignment *p = *q

This assignment statement puts the value of the variable pointed to by **q** in the location of the variable pointed to by **p**.



The pointer **p** still points to the same variable **x** but the old value of **x** is replaced by 200 (which is a pointed to by **q**).

(d) Null pointers

A special constant known as NULL pointer (0) is available in C to initialize pointers that point to nothing. That is the statements

p = 0; (or p = NULL;) p 0 →
 q = 0; (q = NULL;) q 0 →

make the pointers **p** and **q** point to nothing. They can be later used to point any values.

We know that a pointer must be initialized by assigning a memory address before using it. There are two ways of assigning memory address to a pointer.

1. Assigning an existing variable address (static assignment)

```
ptr = &count;
```

2. Using a memory allocation function (dynamic assignment)

```
ptr = (int *) malloc(sizeof(int));
```

4.11 CREATING A LINKED LIST

We can treat a linked list as an abstract data type and perform the following basic operations:

1. Creating a list.
2. Traversing the list.
3. Counting the items in the list.
4. Printing the list (or sub list)
5. Looking up an item for editing or printing.
6. Inserting an item.
7. Deleting an item.
8. Concatenating two lists.

In section 4.7 we created a two-element linked list using the structure variable names **node1** and **node2**. We also used the address operator **&** and member operators, and **→** for creating and accessing individual items. The very idea of using a linked list is to avoid any reference to specific number of items in the list so that we can insert or delete items as and when necessary. This can be achieved by using “anonymous” locations to store nodes. Such locations are accessed not by name, but by means of pointers, which refer to them. (For example, we must avoid using references like **node1.age** and **node1.next →age**.)

Anonymous locations are created using pointers and dynamic memory allocation functions such as **malloc**. We use a pointer head to create and access anonymous nodes. Consider the following:

```

struct linked_list
{
int number;

struct linked_list *next;

};

typedef struct linked_list node;

nod *head;

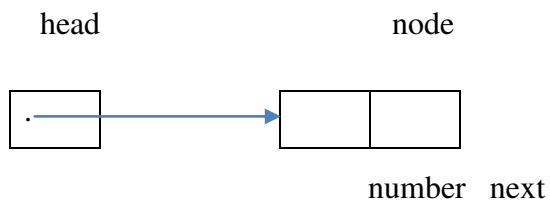
head = (node *) malloc(sizeof(node));

```

The **struct** declaration merely describes the format of the nodes and does not allocate storage. Storage space for a node is created only when the function **malloc** is called in the statement

head = (node *) malloc(sizeof(node));

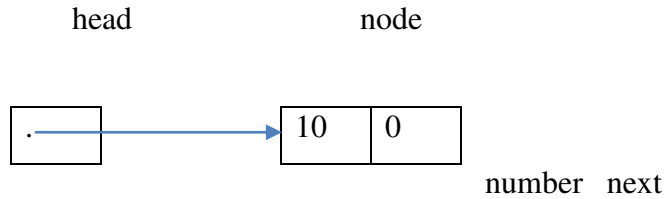
This statement obtains a piece of memory that is sufficient to store a node and assigns its address to the pointer variable **head**. This pointer indicates the beginning of the linked list.



The following statements store values in the member fields:

```
head ->number = 10;
```

```
head ->next = NULL;
```



The second node can be added as follows:

```
head ->number = (node *) malloc(sizeof(node));
head ->next ->number = 20;
head ->next ->next = NULL;
```

Although this process can be continued to create any number of nodes, it becomes cumbersome and clumsy if nodes are more than two. The above process may be easily implemented using both recursion and iteration techniques. The pointer can be move from the current node to the next5 node by a self-replacement statement such as:

```
head = head -> next;
```

The program 4.3 shows creation of a complete linked list and printing of its contents using recursion.

Program 4.3 Write a program to create a linear linked list interactively and print out the list and the total number of items in the list.

The program shown in Fig 4.7 first allocates a block of memory dynamically for the first node using the statement

```
head = (node *) malloc(sizeof(node));
```

which returns a pointer to a structure of a type node that has been type defined earlier. The linked list is then created by the function **create**. The function requests for the number to be placed in the current node that has been created. If the value assigned to the current node is -999, then null is assigned to the pointer variable next and the list ends. Otherwise, memory space is allocated to the next node using again the **malloc** function and the next value is placed

into it. Not that the function **create** calls itself recursively and the process will continue until we enter the number -999.

The items stored in the linked list are printed using the function **print**, which accept a pointer to the current node as an argument. It is a recursive function and stops when it receives a NULL pointer Printing algorithm is as follows:

1. Start with first node.
2. while there are valid nodes left to print
 - (a) print the current item; and
 - (b) advance to next node.

Similarly, the function **count** counts the number of items in the list recursively and return the total number of items to the main function. Note that the counting does not include the item -999(contained in the dummy node).

Program

```
#include<stdio.h>

#include<stdlib.h>

#define NULL 0

struct linked_list

{

    int number;

    struct linked_list *next;

};

typedef struct linked_list node;          /* node type defined */

main()

{
```



```

node *head;

void create(node *p);

int count(node *p);

void pint(node *p);

head = (node *)malloc(sizeof(node));

create(head);

printf("\n");

printf("head");

printf("\n");

printf("\nNumber of items = %d \n", count(head));

}

void create(node *list)

{

printf("Input a number \n");

printf("(type -999 at end): ");

scanf("%d", &list → number );           /* create current node*/

if(list→number == -999)

{

list→next = NULL;

}

```

```

else          /*create next node*/

{

list→next = (node *)malloc(sizeof(node));

create(list→next);

}

return;

}

void print(node *list)

{

if(list→next != NULL)

{

printf(“%d→”, list→number);          /* print current item*/

if(list→nextnext == NULL)

printf(“%d”, list→next→number);

printf(list→next);          /* move to next item*/

}

return;

int count (node *list)

{

```

```
        if(list→next == NULL)
            return(0);
        else
            return(1+ count(list→next));
    }
```

Output

Input a number

(type -999 to end); 60

Input a number

(type -999 to end); 20

Input a number

(type -999 to end); 10

Input a number

(type -999 to end); 40

Input a number

(type -999 to end); 30

Input a number

(type -999 to end); 50

Input a number

(type -999 to end); -999

60→20→10→40→30→50→-999

Number of items = 6

Fig 4.7 Creating a linear linked list

4.12 INSERTING AN ITEM

One of the advantage of linked lists is the comparative ease with which new nodes can be inserted. It requires merely resetting of two pointers (rather than having to move around a list of data as would be the case with arrays.)

Inserting a new item, say X, into the list has three situations:

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

The process of insertion precedes a search for the place of insertion. The search involves in locating a node after which the new item is to be inserted.

A general algorithm for insertion is as follows:

Begin

```
if the list is empty or
the new node comes before the head node then,
insert the new node as the head node,
else
if the new node comes after the last node, then
insert the new node as the end node,
else
insert the new node in body of the list.
```

End

Algorithm for placing the new item at the beginning of a linked list:

1. Obtain space for new node.
2. Assign data to the item field of new node.
3. Set the next field of the new node to point to the start of the list.
4. Change the head pointer to point to the new node.

Algorithm for inserting the new node between two existing nodes, say, N1, and N2

1. Set space for new node .
2. Assign value to the item field of.
3. Set the next field of X to point to node N2.
4. Set the next field of N1 to point to X.

Algorithm for inserting an item at the end of the list is similar to the one for inserting in the middle, except the next field of the new node is set to NULL (or set to point to a dummy or sentinel node, if it exists).

Program 13.4 Write a function to insert a given item before a specified node known as key node.

The function insert shown in Fig. 13.8 requests for the item to inserted as well as the “Key node”. If the insertion happens to be at the beginning, then memory space is created for the new node, the value of new item is assigned to it and the pointer **head** is assigned to the next member. The pointer **new**, which indicates the beginning of the new node is assigned to **head**. Note the following statements:

```
new→number = x;
```

```
new→next = head;
```

```
head = new;
```

```
node *insert(node *head)
```

```
{
```

```

node *find(node *p, int a);

nod *new;          /* pointer to new node*?

node *n1;          /* pointer to node preceding key node */

int key;

int x; /* new item(number) to be inserted*/

printf("Value of new item?");

scanf("%d", &x);

printf("Value of key item? (type -999 if last) ");

scanf("%d", &key);

if(head->number == key)          /* new node is first */
{

    new = (node *)malloc(sizeof(node));

    new->number = x;

    new->next = head;

    head = new;

}

else          /* find key node and insert new node */
{          /* before the key node*/

    n1 = find(head, key);          /* find key node*/

    if(n1 == NULL)

```

```

        printf("\n key is not found \n");

        else                                /* insert new node*/

        {

            new = (node *)malloc(sizeof(node));

            new->number = x;

            new->next = n1->next;

            n1->next = new;

        }

    }

    return(head);
}

node *find(node *lists, int key)

{

    if(list->next->number == key)            /* key found */

        return(list);

    else

        if(list->next->number == NULL)      /* end */

            return(NULL);

    else

        find(list->next, key);

}

```

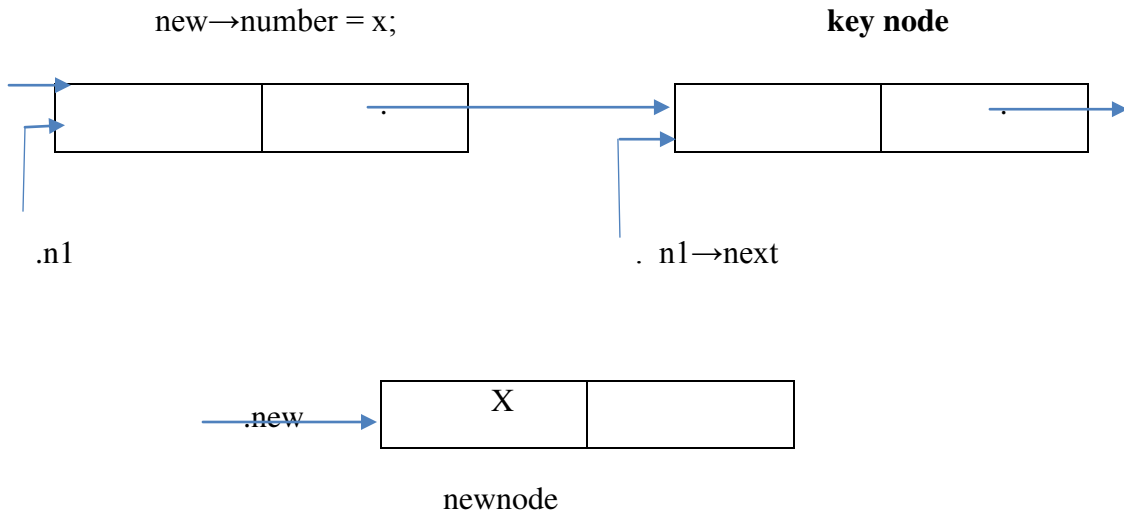
Fig 4.8 A function for inserting an item into linked list

However, if the new item is to be inserted after an existing node, then use the function **find** recursively to locate the 'key node'. The new item is inserted before the key node using the algorithm discussed above. This is illustrated as:

Before insertion

```
new = (node *)malloc(sizeof(node));
```

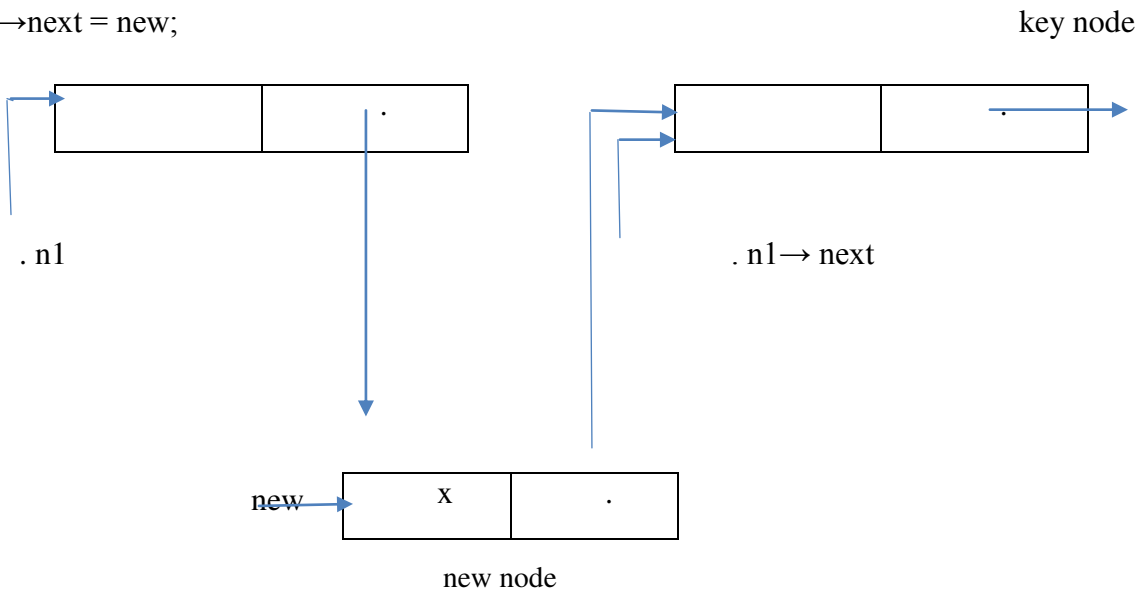
```
new->number = x;
```



After insertion

```
new->next = n1->next;
```

```
n1->next = new;
```



4.13 DELETING AN ITEM

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed. Here again we have three situations.

1. Deleting the first item.
2. Deleting the last item.
3. Deleting between two nodes in the middle of the list.

In the first case, the head pointer is altered to point to the second item in the list. In other two cases, the pointer of the item immediately preceding the one to be deleted is altered to point to the item following the deleted item. The general algorithm for deletion is as follows:

In the first case, the head pointer is altered to point to the second item in the list. In other two cases, the pointer of the item immediately preceding the one to be deleted is altered to point to the item following the deleted item. The general algorithm for deletion is as follows:

begin

if the list is empty, then,

node cannot be deleted

else

if node to be deleted is the first node, then,

make the head to point to second node,

else

delete the node from the body of the list.

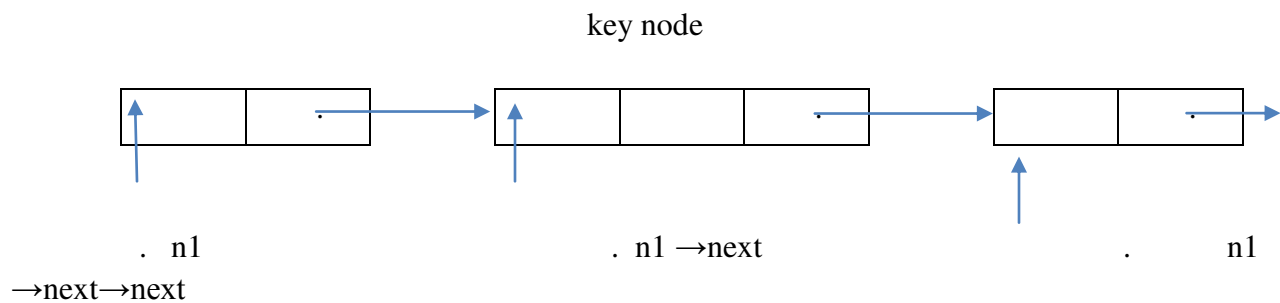
End

The memory space of deleted node may be released for re-use. As in the case of insertion, the process of deletion also involves search for the item to be deleted.

Program 4.5 Write a function to delete a specified node.

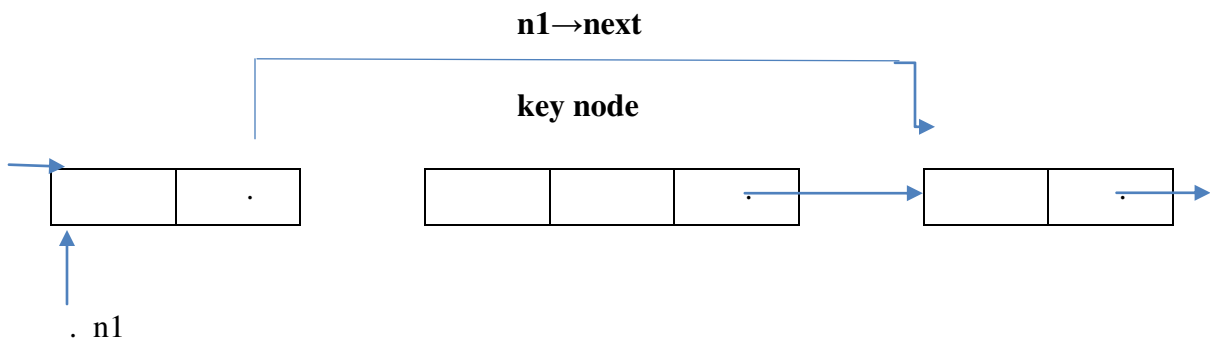
A function to delete a specified node is given in Fig. 13.9. The function first checks whether the specified item belongs to the first node. If yes, then the pointer to the second node is temporarily assigned the pointer variable *p*, the memory space occupied by the first node is freed and the location of the second node is assigned to **head**. Thus the previous second node becomes the first node of the new list.

If the item to be deleted is not the first one, then we use the **find** function to locate the position of “key node” containing the item to be deleted. The pointers are interchanged with the help of a temporary pointer variable making the pointer in the preceding node to point to the node following the key node. The memory space of key node that has been deleted if freed. The figure below shows the relative position of the key node.



The execution of the following code deletes the key node.

```
p = n1 ->next->next;
free(n1 ->next);
n1 ->next = p;
```



```
node *delete(node *head)
```

{

```
node *find(node *p, int a);

int key;          /* item to be deleted */

node *n1;        /* pointer to node preceding keynode */

node *p;         /* temporary pointer */

printf("\n what is the item (number) to be deleted ?");

scanf("%d", &key);

if(head→number == key) /* first node to be deleted*/
{
    p = head →next;      /* pointer to 2nd node in list */
    free(head);          /* release space of key node */
    head = p;           /* make head to point to list node */
}
else
{
    n1 = find(head, key);

    if(n1 == NULL)

        printf("\n key not found \n");

    else                /* delete key node */

    {

        p = n1→next→next); /* pointer to node following the key node*/
```

```

        free(n1→next);                /* free key ;node */

        n1→next = p;                  /* establish link */
    }

}

return(head);

}                                     /* USE FUNCTION find() HERE*/

```

Fig 4.9 A function for deleting an item from linked list

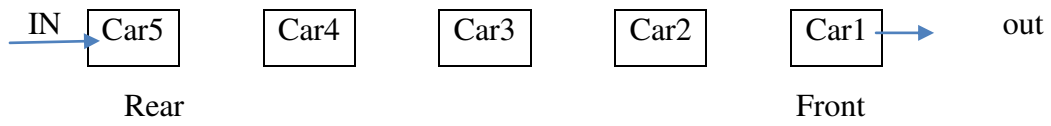
4.14 APPLICATION OF LINKED LISTS

Linked lists concepts are useful to model many different abstract data types such as queues, stacks and trees.

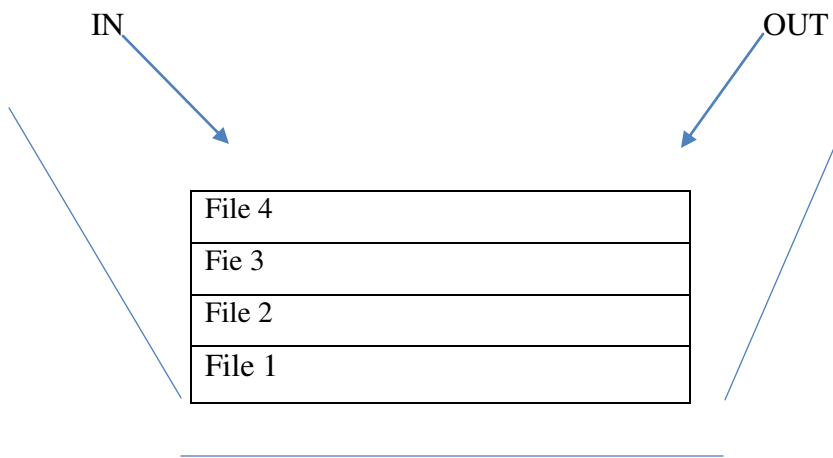
If we restrict the process of insertion to one end of the list and deletions to the other end, then we have a model of a queue. That is, we can insert an item at the rear and remove an item at the front (see Fig 4.10a). This obeys the discipline of “first in, first out” (FIFO). There are many examples of queues in real-life applications.

If we restrict insertions and deletions to occur only at the beginning of list, then we model another data structure known as stack. Stacks are also referred to as push-down lists. An example of a stack is the “in” tray of a busy executive. The files pile up in the tray, and whenever the executive has time to clear the files, he takes it off from the top. That is, files are added at the top and removed from the top (see Fig 4.10b). Stacks are sometimes referred to as “last in, first out” (LIFO) structure.

Lists, queues and stacks are all inherently one-dimensional. A tree represents a two-dimensional linked list. Trees are frequently encountered in everyday life. One example is the organizational chart of a large company. Another example is the chart of sports tournaments.



(a) Queue (repair shop)



(b) Stack (executive tray)

Fig. 4.10 Application of linked lists

UNIT - III

1 Introduction

In this chapter, we consider the problem of approximating a given function by a class of simpler functions, mainly polynomials. There are two main uses of interpolation or interpolating polynomials. The first use is in reconstructing the function $f(x)$ when it is not given explicitly and only the values of $f(x)$ and/or its certain derivatives at a set of points, called **nodes**, **tabular points** or **arguments** are known. The second use is to replace the function $f(x)$ by an interpolating polynomial $P(x)$ so that many common operations such as determination of roots, differentiations and integrations etc. which are intended for the function $f(x)$ may be performed using $P(x)$. In approximations, we measure the deviation of the given function $f(x)$ from the approximating polynomial $P(x)$ for all values of x over a given interval $[a, b]$. We first discuss the methods to construct the interpolating polynomials $P(x)$ to a given function $f(x)$.

Definition 1.1. A polynomial $P(x)$ is called an **interpolating polynomial** if the values of $P(x)$ and/or its certain order derivatives coincide with those of $f(x)$ and/or its same order derivatives at one or more tabular points.

Taylor Series: If the polynomial $P(x)$ is written as the Taylor's expansion, for the function $f(x)$ about a point x_0 , $x_0 \in [a, b]$, in the form

$$P(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2!}(x - x_0)^2 f''(x_0) + \cdots + \frac{1}{n!}(x - x_0)^n f^n(x_0) \quad (1.1)$$

then, $P(x)$ may be regarded as an interpolating polynomial of degree n , satisfying the conditions

$$p^k(x_0) = f^k(x_0), k = 1, 2, \dots, n. \quad (1.2)$$

The term

$$R_n = \frac{1}{(n+1)!} (x-x_0)^{n+1} f^{(n+1)}(\zeta), \quad x_0 < \zeta < x \quad (1.3)$$

which has been neglected in 1.1, is called the *remainder* or the *truncation error*.

The number of terms to be included in 1.1 may be determined by the acceptable error. If this error is $\epsilon > 0$ and the series is truncated at the term $f^n(x_0)$, then

$$\frac{1}{(n+1)!} |x-x_0|^{n+1} |f^{(n+1)}(x)| \leq \epsilon$$

or

$$\frac{1}{(n+1)!} |x-x_0|^{n+1} M_{n+1} \leq \epsilon \quad (1.4)$$

where

$$M_{n+1} = \max_{a \leq x \leq b} |f^{(n+1)}(x)|.$$

Assume that the value of M_{n+1} or its estimate is available.

For a given ϵ and x , 1.4 will determine n , and if n and x are prescribed, it will determine ϵ . When both n and x are given, it will give an upper bound on $(x-x_0)$, that is, it will give an interval about x_0 in which this Taylor's polynomial approximates $f(x)$ to the prescribed accuracy.

Example 1.2. Obtain the Taylor series approximation about $x = 1$, upto second degree terms for the function $f(x) = 1/(1+x)^2$.

We have, $f(x) = \frac{1}{(1+x)^2}$, $f'(x) = -\frac{2x}{(1+x)^2}$, $f''(x) = -\frac{2(1-3x^2)}{(1+x)^3}$, $f'''(x) = -\frac{24x(x^2-1)}{(1+x)^4}$ and $f(1) = 1/2, f'(1) = -1/2, f''(1) = 1/2$. The Taylor series approximation is given by

$$f(x) = f(1) + (x-1)f'(1) + \frac{1}{2}(x-1)^2 f''(1)$$

$$= \frac{1}{2} - \frac{1}{2}(x-1) + \frac{1}{4}(x-1)^2.$$

The error bound is given by

$$|R_2| \leq \frac{(x-1)^3}{3!} M_3, \text{ where } M_3 = \max_{1 \leq x \leq 1.4} |f'''(x)|.$$

We obtain $M_3 = \frac{24(1.4)(0.96)}{16} = 2.016$.

Therefore, $|R_2| \leq \frac{(x-1)^3}{6}(2.016) = 0.336(x-1)^3$.

Maximum absolute error occurs at $x = 1.4$ and this value is 0.0215.

Example 1.3. Obtain polynomial approximation $P(x)$ to $f(x) = e^{-x}$ using the Taylor's expansion about $x_0 = 0$ and determine

(i) x when the error in $P(x)$ obtained from the first four terms only is to be less than 10^{-6} after rounding

(ii) the number of terms in the approximation to find results correct to 10^{-10} for $0 \leq x \leq 1$.

Solution: (i) From $f(x) = e^{-x}$, we have

$f^{(r)}(x) = (-1)^r e^{-x}$ and $f^{(r)}(0) = (-1)^r, r = 0, 1, \dots$. Therefore, we get from 1.1, $P(x) = 1 - x + \frac{x^2}{2} - \frac{x^3}{6}$ and from 1.4, $x^4 M_4 < 24 \times 5 \times 10^{-7}$,

where $M_4 = \max_{0 \leq x \leq 1} |f^{(4)}(x)| = \max_{0 \leq x \leq 1} |e^{-x}| = 1$. Hence, we get $x^4 < 120 \times 10^{-7}$ or $x < 0.06$.

(ii) From 1.4, we obtain $\frac{1}{(n+1)!} < 5 \times 10^{-11}$ which gives $n \geq 14$.

2 Interpolation

In general, if there are $n + 1$ distinct points $a \leq x_0 < x_1 < x_2 < \dots < x_n \leq b$, then the problem of interpolation is to obtain $P(x)$ satisfying the conditions

$$(i) \quad P(x_i) = f(x_i), i = 1, 2, \dots, n. \quad (2.1)$$

or

$$(ii) \quad P(x_i) = f(x_i)$$

$$P'(x_i) = f'(x_i), i = 1, 2, \dots, n. \tag{2.2}$$

The derivative conditions in equation 2.2 may be replaced by more general derivative conditions involving higher order derivatives. The conditions equation 2.1 give rise to *Lagrange/Newton interpolating polynomial* and equation 2.2 give rise to *Hermite interpolating polynomial*.

2.1 Lagrange and Newton Interpolation

Existence

We assume that we are given an interval $[a, b]$ and a function $f(x)$ which is continuous on $[a, b]$. Further, we assume that we have $n + 1$ distinct points $a \leq x_0 < x_1 < x_2 < \dots < x_{n+1} < x_n \leq b$ of $[a, b]$ and that the values of a function $f(x)$ are known at these points. We seek to find the polynomial

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n \tag{2.3}$$

which satisfies the conditions 2.1, that is

$$P(x_i) = f(x_i), \quad i = 1, 2, \dots, n.$$

Substituting the conditions, we obtain the system of equations

$$\begin{aligned} a_0 + a_1x_0 + a_2x_0^2 + \dots + a_nx_0^n &= f(x_0) \\ a_0 + a_1x_1 + a_2x_1^2 + \dots + a_nx_1^n &= f(x_1) \\ \dots & \\ \dots & \\ a_0 + a_1x_n + a_2x_n^2 + \dots + a_nx_n^n &= f(x_n) \end{aligned}$$

This system of equations has a unique solution, or the polynomial $P(x)$ exists if the *Vandermonde's determinant*

$$V(x_0, x_1, \dots, x_n) = \begin{vmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{vmatrix} \neq 0.$$

Let

$$V(x_0, x_1, \dots, x_{n-1}, x) = \begin{vmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^n \\ 1 & x & x^2 & \dots & x^n \end{vmatrix}$$

By the properties of the determinants, we get

$$V(x_0, x_1, \dots, x_{n-1}, x) = (x - x_0)(x - x_1) \dots (x - x_{n-1})A$$

where A is a constant. Comparing the coefficients of x^n , we get

$$A = \begin{vmatrix} 1 & x_0 & \dots & x_0^{n-1} \\ 1 & x_1 & \dots & x_1^{n-1} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ 1 & x_{n-1} & \dots & x_{n-1}^{n-1} \end{vmatrix} = V(x_0, x_1, \dots, x_{n-1}).$$

Therefore,

$$V(x_0, x_1, \dots, x_{n-1}, x) = (x - x_0)(x - x_1) \dots (x - x_{n-1})V(x_0, x_1, \dots, x_{n-1}).$$

Substituting $x = x_n$, we obtain

$$V(x_0, x_1, \dots, x_n) = V(x_0, x_1, \dots, x_{n-1}) \prod_{i=0}^{n-1} (x_n - x_i).$$

Applying recursively, we get

$$V(x_0, x_1, \dots, x_n) = \prod_{i,j=0; i>j}^n (x_i - x_j) \neq 0$$

since x_i are distinct.

Uniqueness

To prove that the polynomial $P(x)$ obtained above is unique, we assume that there is another polynomial $P^*(x)$ which also satisfies

$$P^*(x_i) = f(x_i), \quad i = 1, 2, \dots, n. \quad (2.4)$$

Consider the polynomial

$$Q(x) = P(x) - P^*(x). \quad (2.5)$$

Since $P(x)$ and $P^*(x)$ are both polynomials of degree $\leq n$, $Q(x)$ is also a polynomial of degree $\leq n$ satisfying the conditions

$$Q(x_i) = P(x_i) - P^*(x_i) = 0, \quad i = 1, 2, \dots, n. \quad (2.6)$$

Therefore, $Q(x)$ is a polynomial of degree $\leq n$ which has $n+1$ distinct roots $x_0, x_1, x_2, \dots, x_n$. This implies that $Q(x) \equiv 0$, because a polynomial $Q(x)$ of degree n has exactly n roots, real or complex. Therefore, $P^*(x) = P(x)$.

Thus, the interpolating polynomials obtained in two different ways may be different in form, but are identical otherwise. Depending on its form the polynomial is called either the *Lagrange interpolating polynomial* or the *Newton divided differences interpolating polynomial*.

We discuss now interpolations of various degrees.

2.2 Linear Interpolations

Here, $n = 1$ and we want to determine a polynomial

$$P(x) = a_1x + a_0 \quad (2.7)$$

where a_0 and a_1 are arbitrary constants, which satisfies the interpolating conditions $f(x_0) = P(x_0)$ and $f(x_1) = P(x_1)$. We have

$$\begin{aligned}
 f(x_0) &= P(x_0) = a_1x_0 + a_0 \\
 f(x_1) &= P(x_1) = a_1x_1 + a_0
 \end{aligned}
 \tag{2.8}$$

Eliminating a_0 and a_1 from 2.8, we obtain the required linear interpolating polynomial as

$$\begin{vmatrix}
 P(x) & x & 1 \\
 f(x_0) & x_0 & 1 \\
 f(x_1) & x_1 & 1
 \end{vmatrix} = 0
 \tag{2.9}$$

which is shown graphically in Figure 3.1.

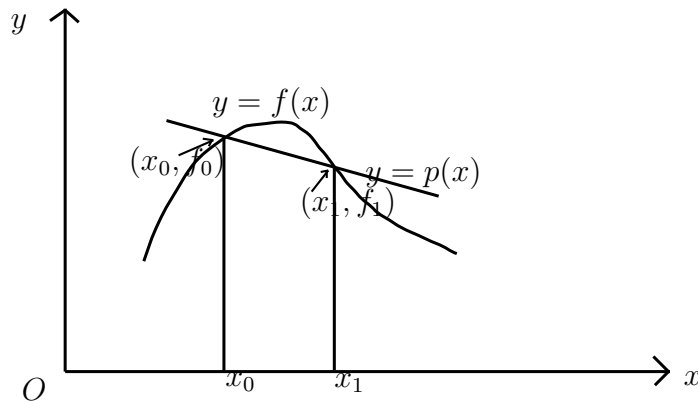


Figure 3.1. Linear Interpolation

2.3 Lagrange Interpolation

We simplify 2.9 in terms of the first column and obtain

$$P(x)(x_0 - x_1) - f(x_0)(x - x_1) + f(x_1)(x - x_0) = 0$$

or

$$\begin{aligned} P(x) &= \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1) \\ &= l_0(x) f(x_0) + l_1(x) f(x_1) \end{aligned} \quad (2.10)$$

where

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}, l_1(x) = \frac{x - x_0}{x_1 - x_0}.$$

The function $l_0(x)$ and $l_1(x)$ are called the *Lagrange fundamental polynomials* and it can be verified that they satisfy the conditions

$$l_0(x) + l_1(x) = 1$$

$$l_0(x_0) = 1, l_0(x_1) = 0$$

$$l_1(x_1) = 0, l_1(x_0) = 1$$

or

$$l_i(x_j) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \quad (2.11)$$

The degree of the polynomials $l_0(x)$ and $l_1(x)$ is one.

The equation (2.10) is the linear Lagrange interpolating polynomial. Thus, to obtain the Lagrange interpolating polynomial, we first determine Lagrange fundamental polynomials, multiply by the corresponding function values and add them together.

2.4 Iterated Linear interpolation

We can write (2.10) as

$$\begin{aligned}
 P(x) &= \frac{1}{x_1 - x_0} [(x_1 - x_0)f(x_0) - (x_1 - x_0)f(x_1)] \\
 &= \frac{1}{x_1 - x_0} \begin{vmatrix} I_0(x) & x_0 - x \\ I_1(x) & x_1 - x \end{vmatrix}
 \end{aligned} \tag{2.12}$$

where $I_0(x) = f(x_0)$ and $I_1(x) = f(x_1)$.

We may regard $I_0(x)$ and $I_1(x)$ as two independent zero degree interpolating polynomials to $f(x)$. It is easily verified that $I_{0,1}(x_0) = f(x_0)$ and $I_{0,1}(x_1) = f(x_1)$. The equation (2.12) is called *Aitken's* or *iterated* linear interpolating polynomials.

2.5 Newton's Divided Difference Interpolation

We now, expand the determinant (2.9) in terms of the first row and get

$$\begin{aligned}
 P(x) &= f(x_0) + (x - x_0) \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\
 &= f(x_0) + (x - x_0) f[x_0, x_1]
 \end{aligned} \tag{2.13}$$

where

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = f[x_0, x_1]. \tag{2.14}$$

The ratio $f[x_0, x_1]$ is called the *first divided difference* of $f(x)$ relative to x_0 and x_1 . We may write (2.13) as

$$\frac{P(x) - f(x_0)}{x - x_0} = f[x_0, x_1]. \tag{2.15}$$

The equation (2.13) or (2.15) is the linear *Newton* interpolating polynomial with divided differences.

Example 2.1. Given $f(2) = 4, f(2.5) = 5.5$, find the linear interpolating polynomial using (i) Lagrange interpolation, (ii) Aitken's iterated interpolating and (iii) Newton's divided difference interpolation. Hence, find an approximate value of $f(2.2)$.

We have $x_0 = 2, x_1 = 2.5, f(x_0) = 4$ and $f(x_1) = 5.5$.

(i) The Lagrange fundamental polynomials are given by

$$l_0(x) = \frac{x-x_1}{x_0-x_1} = \frac{x-2.5}{(-0.5)}, l_1(x) = \frac{x-x_0}{x_1-x_0} = \frac{x-2}{0.5}, \text{ and}$$

$$\begin{aligned} P_1(x) &= l_0(x)f(x_0) + l_1(x)f(x_1) \\ &= -2(x-2.5)(4) + 2(x-2)(5.5) = 3x - 2. \end{aligned}$$

Hence,

$$f(2.2) \approx P_1(2.2) = 3(2.2) - 2 = 4.6.$$

(ii) From (2.12), we obtain

$$P_1(x) = \frac{1}{x_1 - x_0} \begin{vmatrix} I_0(x) & x_0 - x \\ I_1(x) & x_1 - x \end{vmatrix} = \frac{1}{0.5} \begin{vmatrix} 4 & 2 - x \\ 5.5 & 2.5 - x \end{vmatrix} = 3x - 2.$$

$$P_1(2.2) = 4.6.$$

(iii) We have $f[x_0, x_1] = \frac{f(x_0)-f(x_1)}{x_0-x_1} = \frac{5.5-4}{0.5} = 3$, and

$$P_1(x) = f(x_0) + (x - x_0)f[x_0, x_1]$$

$$= 4 + (x - 2)3 = 3x - 2.$$

$$P_1(2.2) = 4.6.$$

Truncation Error Bounds

The polynomial $P(x)$ coincides with the function $f(x)$ at x_0 and x_0 and x_1 , and it deviates at all other points, in the interval (x_0, x_1) as shown in Figure 3.1. This deviation is called the **truncation error** and may be written as

$$E_1(f; x) = f(x) - P(x). \quad (2.16)$$

We will now derive an expression for $E_1(f; x)$ for $x \in [x_0, x_1]$. We use the following result.

Theorem 2.2. (Rolle) *If $g(x)$ is a continuous function on some interval $[a, b]$ and differentiable on (a, b) and if $g(a) = 0, g(b) = 0$, then there is at least one point ξ inside (a, b) for which $g'(\xi) = 0$.*

We notice that if $x = x_0$ or $x = x_1$ then $E_1(f; x) = 0$. If $x \in (x_0, x_1)$, then for this x we define a function $g(t)$ as

$$g(t) = f(t) - P(t) - [f(x) - P(x)] \frac{(t - x_0)(t - x_1)}{(x - x_0)(x - x_1)}. \quad (2.17)$$

It is easy to verify that $g(t) = 0$ at the three distinct points $t = x_0, t = x_1$ and $t = x$. The function $g(t)$ satisfies the conditions of the Rolle's theorem.

Applying the Rolle's theorem on the intervals (x_0, t) and (t, x_1) separately, we get

$$g'(\xi_1) = 0, x_0 < \xi_1 < t \text{ and } g'(\xi_2) = 0, t < \xi_2 < x_1.$$

Now, $g'(t)$ also satisfies the conditions of the Rolle's theorem. Applying Rolle's theorem for $g'(t)$ on the interval (ξ_1, ξ_2) , we obtain $g''(\xi) = 0, \xi_1 < \xi < \xi_2$, or $x_0 < \xi < x_1$. Differentiating (2.17) twice with respect to t , we obtain

$$g''(t) = f''(t) - \frac{2(f(x) - p(x))}{(x - x_0)(x - x_1)}. \quad (2.18)$$

Setting $g''(t) = 0$ and solving (2.18) for $f(x)$ we obtain

$$f(x) = P(x) + \frac{1}{2}(x - x_0)(x - x_1)f''(\xi). \quad (2.19)$$

Therefore, the truncation error in linear interpolation is given by

$$E_1(f; x) = \frac{1}{2}(x - x_0)(x - x_1)f''(\xi). \quad (2.20)$$

If we can determine a bound for $f''(x)$ in $[x_0, x_1]$, i.e.

$$|f''(x)| \leq M_2, x \in [x_0, x_1]$$

then

$$\begin{aligned} |f(x) - P(x)| &= \left| \frac{1}{2}(x - x_0)(x - x_1)f''(\xi) \right| \\ &\leq \frac{1}{2} \max_{x_0 \leq x \leq x_1} |(x - x_0)(x - x_1)f''(\xi)| \\ &\leq \frac{1}{2} \max_{x_0 \leq x \leq x_1} |(x - x_0)(x - x_1)| M_2. \end{aligned} \quad (2.21)$$

Let $w(x) = (x - x_0)(x - x_1)$. Setting $w'(x) = 0$, we obtain the critical point of $w(x)$ as $x = (x_0 + x_1)/2$. Hence, the maximum value of $|(x - x_0)(x - x_1)|$ occurs at $x = (x_0 + x_1)/2$ and 2.21 becomes

$$|f(x) - P(x)| \leq \frac{1}{8}(x_1 - x_0)^2 M_2. \quad (2.22)$$

Equation (2.22) can be used to construct a table of values for a function $f(x)$ for equally spaced nodal points $x_i = a + ih, i = 0, 1, \dots, n, h = (b - a)/n$; so that

the maximum absolute truncation error using the linear interpolating polynomial $P(x)$ is less than a given error tolerance $\epsilon > 0$. Since $x_1 - x_0 = h$, we have from 2.22

$$\frac{h^2}{8} \max |f''(x)| \leq \epsilon. \quad (2.23)$$

Example 2.3. Using the data $\sin(0.1) = 0.09983$ and $\sin(0.2) = 0.19867$, find an approximate value of $\sin(0.15)$ by Lagrange interpolation. Obtain a bound on the truncation error.

We have

$$\begin{aligned} P_1(0.15) &= \frac{0.15 - 0.2}{0.1 - 0.2}(0.09983) + \frac{0.15 - 0.1}{0.2 - 0.1}(0.19867) \\ &= (0.5)(0.09983) + (0.5)(0.19867) = 0.14925 \end{aligned}$$

The truncation error is

$$E_1(f; x) = \frac{(x-0.1)(x-0.2)}{2}(-\sin\xi)$$

where $0.1 < \xi < 0.2$.

The maximum value of $|\sin\xi|$, $\xi \in [0.1, 0.2]$ is $\sin(0.2) = 0.19867$.

$$\begin{aligned} \text{Thus, } |E_1(f; x)| &\leq \left| \frac{(0.15-0.1)(0.15-0.2)}{2} \right| (0.19867) \\ &= (0.19867)(0.00125) \approx 0.00025. \end{aligned}$$

Example 2.4. Determine the stepsize h that can be used in the tabulation of $f(x) = \sin x$ in the interval $[1, 3]$ so that the linear interpolation will be correct to four decimal places after rounding.

We have

$$f(x) = \sin x,$$

$$f'(x) = \cos x,$$

$$f''(x) = -\sin x,$$

and $\max |-\sin x| = 1$. Hence we obtain

$$\frac{h^2}{8} \leq 5 \times 10^{-5}$$

This gives $h \leq 0.02$.

Example 2.5. The function $f(x) = \sin x$ is defined on the interval $[1, 3]$.

(i) Obtain the Lagrange linear interpolating polynomial in this interval and find the bound on the truncation error. Obtain the approximate values of $f(1.5)$ and $f(2.5)$.

(ii) Divide the interval $[1, 3]$ into two subintervals $[1, 2]$ and $[2, 3]$. Obtain the Lagrange linear interpolating polynomial in each subinterval and find the bound on the truncation error. Hence find the approximate values of $f(1.5)$ and $f(2.5)$.

Compare with the exact values.

We have $\sin 1 = 0.8415$, $\sin 2 = 0.9093$, $\sin 3 = 0.1411$ and $f''(x) = -\sin x$. Now, on the interval $[1, 3]$, we obtain

$$\begin{aligned} P_{11}(x) &= \frac{x-3}{1-3} \sin 1 + \frac{x-1}{3-1} \sin 3 \\ &= -\frac{1}{2}(x-3)(0.8415) + \frac{1}{2}(x-1)(0.1411) \\ &= -0.3502x + 1.1917. \end{aligned}$$

$$|E_{11}(x)| \leq \frac{1}{2} \max_{1 \leq x \leq 3} |(x-1)(x-3)| \max_{1 \leq x \leq 3} |\sin x| = 0.5.$$

We find that $f(1.5) \approx P_{11}(1.5) = 0.6664$ and $f(2.5) \approx P_{11}(2.5) = 0.3162$.

On the interval $[1, 2]$, we obtain

$$\begin{aligned}
P_{12}(x) &= \frac{x-2}{1-2} \sin 1 + \frac{x-1}{2-1} \sin 2 \\
&= -(x-2)(0.8415) + (x-1)(0.9093) \\
&= -0.0678x + 0.7737.
\end{aligned}$$

$$|E_{12}(x)| \leq \frac{1}{2} \max_{1 \leq x \leq 2} |(x-1)(x-2)| \max_{1 \leq x \leq 2} |\sin x| = \frac{1}{8} = 0.125.$$

We find that $f(1.5) \approx P_{12}(1.5) = 0.8754$.

On the interval $[2, 3]$, we obtain

$$\begin{aligned}
P_{13}(x) &= \frac{x-3}{2-3} \sin 2 + \frac{x-2}{3-2} \sin 3 \\
&= -(x-3)(0.9093) + (x-2)(0.1411) \\
&= -0.7682x + 2.4457.
\end{aligned}$$

$$|E_{13}(x)| \leq \frac{1}{2} \max_{2 \leq x \leq 3} |(x-2)(x-3)| \max_{2 \leq x \leq 3} |\sin x| = \frac{1}{8} \sin 2 = 0.1137.$$

We find that $f(2.5) \approx P_{13}(2.5) = 0.5252$.

We note that the exact values are

$$f(1.5) = \sin 1.5 = 0.9975 \quad \text{and} \quad f(2.5) = \sin 2.5 = 0.5985.$$

2.6 Quadratic Interpolation

Here, $n = 2$ and we want to determine a polynomial

$$P_2(x) = a_0 + a_1x + a_2x^2$$

where a_0 , a_1 and a_2 are arbitrary constants, which satisfies the interpolatory conditions $f(x_0) = P_2(x_0)$, $f(x_1) = P_2(x_1)$ and $f(x_2) = P_2(x_2)$.

We have

$$f(x_0) = a_0 + a_1x_0 + a_2x_0^2$$

$$f(x_1) = a_0 + a_1x_1 + a_2x_1^2$$

$$f(x_2) = a_0 + a_1x_2 + a_2x_2^2$$

Eliminating a_0 , a_1 and a_2 , we obtain the required quadratic interpolating polynomial as

$$\begin{vmatrix} P_2(x) & 1 & x & x^2 \\ f(x_0) & 1 & x_0 & x_0^2 \\ f(x_1) & 1 & x_1 & x_1^2 \\ f(x_2) & 1 & x_2 & x_2^2 \end{vmatrix} = 0$$

Expanding the determinant, we obtain

$$P_2(x)D_0 - f(x_0)D_1 + f(x_1)D_2 - f(x_2)D_3 = 0$$

where

$$D_0 = \begin{vmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{vmatrix} = (x_0 - x_1)(x_1 - x_2)(x_2 - x_0)$$

$$D_1 = \begin{vmatrix} 1 & x & x^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{vmatrix} = (x - x_1)(x_1 - x_2)(x_2 - x)$$

$$D_2 = \begin{vmatrix} 1 & x & x^2 \\ 1 & x_0 & x_0^2 \\ 1 & x_2 & x_2^2 \end{vmatrix} = (x - x_0)(x_0 - x_2)(x_2 - x)$$

$$D_3 = \begin{vmatrix} 1 & x & x^2 \\ 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \end{vmatrix} = (x - x_0)(x_0 - x_1)(x_1 - x)$$

Therefore, $P_2(x) = \frac{D_1}{D_0}f(x_0) - \frac{D_2}{D_0}f(x_1) + \frac{D_3}{D_0}f(x_2)$

$$= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}f(x_2)$$

$$= l_0(x)f(x_0) + l_1(x)f(x_1) + l_2(x)f(x_2).$$

The truncation error of the Lagrange quadratic interpolating polynomial is given by

$$E_2(f; x) = f(x) - P_2(x).$$

We note that $E_2(f; x) = 0$ for $x = x_0, x_1$ and x_2 . If $x \in (x_0, x_2)$, then for this value of x we define a function $g(t)$ as

$$g(t) = f(t) - P_2(t) - [f(x) - P_2(x)] \frac{(t-x_0)(t-x_1)(t-x_2)}{(x-x_0)(x-x_1)(x-x_2)}$$

It is easy to show that $g(t) = 0$ at four distinct points $t = x_0, x_1, x_2$ and x . The function $g(t)$ satisfies the conditions of Rolle's theorem. Applying the Rolle's theorem repeatedly for $g(t)$, $g'(t)$ and $g''(t)$, we obtain $g'''(\xi) = 0$, where ξ is some point such that $\min(x_0, x_1, x_2, x) < \xi < \max(x_0, x_1, x_2, x)$.

Differentiating $g(t)$ three times with respect to t , we get

$$g'''(t) = f'''(t) - \frac{(3!)[f(x) - P_2(x)]}{(x-x_0)(x-x_1)(x-x_2)}$$

Setting $g'''(\xi) = 0$ and solving for $f(x)$, we get

$$f(x) = P_2(x) + \frac{1}{3!}(x-x_0)(x-x_1)(x-x_2)f'''(\xi).$$

Hence, the truncation error in the Lagrange quadratic interpolation is given by

$$E_2(f; x) = f(x) - P_2(x) = \frac{1}{3!}(x-x_0)(x-x_1)(x-x_2)f'''(\xi).$$

and its bound is

$$|f(x) - P_2(x)| \leq \frac{1}{6}M_3 \left[\max_{x_0 \leq x \leq x_2} |(x-x_0)(x-x_1)(x-x_2)| \right]$$

where $M_3 = \max_{x_0 \leq x \leq x_2} |f'''(x)|$.

2.7 Higher Order Interpolation

The Lagrange fundamental polynomials of degree n based on $n+1$ distinct points $a \leq x_0 < x_1 < x_2 < \dots < x_n \leq b$ and which satisfy (2.11) can be

written in the form

$$l_i(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_0)(x_i - x_1) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)} \quad (2.24)$$

$$i = 0, 1, \dots, n.$$

An alternative form of (2.24) is given by

$$l_i(x) = \frac{w(x)}{(x - x_i)w'(x_i)}$$

where

$$w(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$$

$$w'(x_i) = (x_i - x_0)(x_i - x_1) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)$$

and a prime represents differentiation with respect to x . Thus the polynomial

$$P(x) = \sum_{i=0}^n l_i(x) f(x_i) \quad (2.25)$$

where $l_i(x)$ are given by (2.24), is the Lagrange interpolating polynomial of degree n .

The truncation error in the Lagrange interpolation is given by

$$E_n(f; x) = f(x) - P(x)$$

. Since $E_n(f; x) = 0$ at $x = x_i$, $i = 0, 1, \dots, n$, then for $x \in [a, b]$ and $x \neq x_i$, we define a function $g(t)$ as

$$g(t) = f(t) - P(t) - [f(x) - P(x)] \frac{(t - x_0)(t - x_1) \cdots (t - x_n)}{(x - x_0)(x - x_1) \cdots (x - x_n)} \quad (2.26)$$

We observe that $g(t) = 0$ at $t = x$ and $t = x_i$, $i = 0, 1, \dots, n$.

Applying the Rolle's theorem repeatedly for $g(t)$, $g'(t), \dots$ and $g^{(n)}(t)$, we obtain $g^{(n+1)}(\xi) = 0$ where ξ is some point such that $\min(x_0, x_1, \dots, x_n, x) < \xi < \max(x_0, x_1, \dots, x_n, x)$.

Differentiating (2.26) $n + 1$ times with respect to t , we get

$$g^{(n+1)}(t) = f^{(n+1)}(t) - \frac{(n+1)! [f(x) - P(x)]}{(x-x_0)(x-x_1)\cdots(x-x_n)} \quad (2.27)$$

Setting $g^{(n+1)}(\xi) = 0$ and solving (2.27) for $f(x)$, we get

$$f(x) = P(x) + \frac{w(x)}{(n+1)!} f^{(n+1)}(\xi).$$

Hence the truncation error in Lagrange interpolation is given by

$$E_n(f; x) = \frac{w(x)}{(n+1)!} f^{(n+1)}(\xi). \quad (2.28)$$

2.8 Iterated Interpolation

The iterated form of the Lagrange interpolation can be written as

$$I_{0,1,\dots,n}(x) = \frac{1}{x_n - x_{n-1}} \begin{vmatrix} I_{0,1,\dots,n-1}(x) & x_{n-1} - x \\ I_{0,1,\dots,n-2,n}(x) & x_n - x \end{vmatrix} \quad (2.29)$$

The interpolating polynomials appearing on the right side of (2.29) are any two independent $(n-1)$ th degree polynomials which could be constructed in a number of ways. In the *Aitken* method, we construct the successive iterated interpolations as given in Table 1.1.

Table 1.1 Iterated Interpolation

x_0	$x_0 - x$	$I_0(x)$			
x_1	$x_1 - x$	$I_1(x)$	$I_{0,1}(x)$		
x_2	$x_2 - x$	$I_2(x)$	$I_{0,2}(x)$	$I_{0,1,2}(x)$	
.	
.	
.	
x_{n-1}	$x_{n-1} - x$	$I_{n-1}(x)$	$I_{0,n-1}(x)$	$I_{0,1,n-1}(x)$	
x_n	$x_n - x$	$I_n(x)$	$I_{0,n}(x)$	$I_{0,1,n}(x) \dots$	$I_{0,1,2,\dots,n}(x)$

where $I_i(x) = f(x_i)$ and

$$I_{0,1,n}(x) = \frac{1}{x_n - x_1} \begin{vmatrix} I_{0,1}(x) & x_1 - x \\ I_{0,n}(x) & x_n - x \end{vmatrix}$$

The bound on the error is given by

$$|E_n(f; x)| = \frac{1}{(n+1)!} |w(x)| |f^{(n+1)}(\xi)|.$$

$$\leq \frac{1}{(n+1)!} M_{n+1} \left[\max_{x_0 \leq x \leq x_n} |w(x)| \right]$$

where

$$M_{n+1} = \max_{x_0 \leq x \leq x_n} |f^{(n+1)}(x)| \quad (2.30)$$

Example 2.6. Given that $f(0) = 1$, $f(1) = 3$, $f(3) = 55$, find the unique polynomial of degree 2 or less, which fits the given data. Find the bound on the error.

We have $x_0 = 0$, $x_1 = 1$, $x_2 = 3$, $f_0 = 1$, $f_1 = 3$ and $f_2 = 55$. The Lagrange fundamental polynomials are given by

$$l_0(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{(x-1)(x-3)}{(-1)(-3)} = \frac{1}{3}(x^2 - 4x + 3)$$

$$l_1(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = \frac{x(x-3)}{(1)(-2)} = \frac{1}{2}(3x - x^2)$$

$$l_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{x(x-1)}{3(2)} = \frac{1}{6}(x^2 - x).$$

Hence, the Lagrange quadratic interpolating polynomial is given by

$$\begin{aligned} P_2(x) &= l_0(x)f_0 + l_1(x)f_1 + l_2(x)f_2 \\ &= \frac{1}{3}(x^2 - 4x + 3) + \frac{3}{2}(3x - x^2) + \frac{55}{6}(x^2 - x). \\ &= 8x^2 - 6x + 1. \end{aligned}$$

We have,

$$\begin{aligned} |E_2(f; x)| &\leq \frac{1}{6} M_3 \left[\max_{0 \leq x \leq 3} |x(x-1)(x-3)| \right] \\ &= \frac{1}{6}(2.1126)M_3 = 0.3521 M_3 \end{aligned}$$

where $M_3 = \max_{0 \leq x \leq 3} |f'''(x)|$ and since the maximum of $|x(x-1)(x-3)|$ occurs at $x = 2.2152$.

Example 2.7. The following values of the function $f(x) = \sin x + \cos x$, are

given

x	10^0	20^0	30^0
$f(x)$	1.1585	1.2817	1.3660

Construct the quadratic interpolating polynomial that fits the data. Hence, find $f(\pi/12)$. Compare with the exact value. Since the values of f at $\pi/12$ radians is required, we convert the data into radian measure. We have $x_0 = 10^0 = \frac{\pi}{18} = 0.1745$, $x_1 = 20^0 = \frac{\pi}{9} = 0.3491$, $x_2 = 30^0 = \frac{\pi}{6} = 0.5236$.

The Lagrange fundamental polynomials are given by

$$\begin{aligned}
 l_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{(x-0.3491)(x-0.5236)}{(-0.1746)(-0.3491)} \\
 &= 16.4061(x^2 - 0.8727x + 0.1828) \\
 l_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = \frac{(x-0.1745)(x-0.5236)}{(0.1746)(-0.1745)} \\
 &= -32.8616(x^2 - 0.6981x + 0.0914) \\
 l_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} = \frac{(x-0.1745)(x-0.3491)}{(0.3491)(0.1745)} \\
 &= 16.4155(x^2 - 0.5236x + 0.0609).
 \end{aligned}$$

Hence, the Lagrange quadratic interpolating polynomial is given by

$$\begin{aligned}
 P_2(x) &= 16.4061(x^2 - 0.8727x + 0.1828)(1.1585) \\
 &\quad - 32.8616(x^2 - 0.6981x + 0.0914)(1.2817) \\
 &\quad + 16.4155(x^2 - 0.5236x + 0.0609)(1.3660) \\
 &= -0.6887x^2 + 1.0751x + 0.9903.
 \end{aligned}$$

Hence, $f(\pi/12) = f(0.2618) = 1.2246$. The exact value is $f(0.2618) =$

$$\sin (0.2618) + \cos (0.2618) = 1.2247 .$$

2.9 Newton's Divided Difference Interpolation

The linear Newton divided difference interpolation (2.13) is easy to generalize.

We define the higher order divided differences as

$$\begin{aligned} f[x_0, x_1, x_2] &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} \\ &= \frac{1}{(x_2 - x_0)} \left[\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0} \right] \\ &= \frac{f(x_2)}{(x_2 - x_0)(x_2 - x_1)} - \frac{f(x_1)}{(x_2 - x_0)} \left[\frac{1}{x_2 - x_1} + \frac{1}{x_1 - x_0} \right] + \frac{f(x_0)}{(x_2 - x_0)}(x_1 - x_0) \\ &= \frac{f(x_0)}{(x_0 - x_1)(x_0 - x_2)} + \frac{f(x_1)}{(x_1 - x_0)(x_1 - x_2)} + \frac{f(x_2)}{(x_2 - x_0)(x_2 - x_1)} \\ f[x_0, x_1, x_2, \dots, x_{k-1}, x_k] &= \frac{f[x_1, x_2, \dots, x_k] - f[x_0, x_1, \dots, x_{k-1}]}{x_k - x_0} \\ k &= 3, 4, \dots, n . \end{aligned}$$

In terms of function values, the n th divided difference can be written as

$$f[x_0, x_1, x_2, \dots, x_n] = \sum_{i=0}^n \frac{f(x_i)}{\prod_{j=0, i \neq j}^n (x_i - x_j)} \quad (2.31)$$

The divided differences may be calculated with the help of Table 3.2.

Table 1.2 Divided Difference (d.d) Table

	first d.d	second d.d	third d.d
x_0	$f[x_0]$		
x_1	$f[x_1]$	$f[x_0, x_1]$	
x_2	$f[x_2]$	$f[x_1, x_2]$	$f[x_0, x_1, x_2]$
x_3	$f[x_3]$	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$
			$f[x_0, x_1, x_2, x_3]$

Note that

$$f[x_0, x_1] = f[x_1, x_0]$$

$$f[x_0, x_1, x_2] = f[x_2, x_1, x_0] \text{ etc.}$$

The interpolating polynomial $P_n(x)$, interpolating at the $n+1$ distinct points x_0, x_1, \dots, x_n can also be written as

$$P_n(x) = a_0 + (x-x_0)a_1 + (x-x_0)(x-x_1)a_2 + \dots + (x-x_0)\dots(x-x_{n-1})a_n. \quad (2.32)$$

Substituting successively $x = x_0, x = x_1, \dots, x = x_n$, we obtain

$$P_n(x_0) = f[x_0] = a_0,$$

$$P_n(x_1) = f[x_1] = a_0 + (x_1 - x_0)a_1 = f[x_0] + (x_1 - x_0)a_1,$$

or
$$a_1 = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = f[x_0, x_1],$$

$$P_n(x_2) = f[x_2] = a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2$$

or
$$a_2 = \frac{1}{(x_2 - x_0)(x_2 - x_1)} \left[f[x_2] - f[x_0] - (x_2 - x_0) \left\{ \frac{f[x_1] - f[x_0]}{x_1 - x_0} \right\} \right]$$

$$\begin{aligned}
&= \frac{f[x_0]}{(x_0-x_1)(x_0-x_2)} + \frac{f[x_1]}{(x_1-x_0)(x_1-x_2)} + \frac{f[x_2]}{(x_2-x_0)(x_2-x_1)} \\
&= f[x_0, x_1, x_2].
\end{aligned}$$

Using induction, we can prove that

$$a_n = f[x_0, x_1, \dots, x_n]. \quad (2.33)$$

The divided difference interpolating polynomial becomes

$$P_n(x) = f[x_0] + (x-x_0)f[x_0, x_1] + \dots + (x-x_0)\dots(x-x_{n-1})f[x_0, x_1, \dots, x_n]. \quad (2.34)$$

Note that, since the interpolating polynomial is unique, Lagrange and divided difference polynomials are two different forms of the same polynomial.

Example 2.8. Find the unique polynomial of degree 2 or less, such that $f(0) = 1$, $f(1) = 3$, $f(3) = 55$, using

- (i) the iterated interpolation
- (ii) the Newton divided difference interpolation.

(i) the iterated interpolating polynomial becomes

$$I_{0,1}(x) = \frac{1}{(1-0)} \begin{vmatrix} 1 & 0-x \\ 3 & 1-x \end{vmatrix} = 1 + 2x$$

$$I_{0,2}(x) = \frac{1}{(3-0)} \begin{vmatrix} 1 & 0-x \\ 55 & 3-x \end{vmatrix} = 1 + 18x$$

$$\begin{aligned}
I_{0,1,2}(x) &= \frac{1}{(3-1)} \begin{vmatrix} I_{0,1}(x) & 1-x \\ I_{0,2}(x) & 3-x \end{vmatrix} \\
&= \frac{1}{3} [(1+2x)(3-x) - (1-x)(1+18x)] \\
&= 8x^2 - 6x + 1.
\end{aligned}$$

(ii) The divided differences are given by

$$f[0, 1] = \frac{3-1}{1-0} = 2, f[1, 3] = \frac{55-3}{3-1} = 26, f[0, 1, 3] = \frac{26-2}{3-0} = 8.$$

The Newton divided difference interpolating polynomial becomes

$$\begin{aligned}
P_2(x) &= f[0] + (x-0)f[0, 1] + (x-0)(x-1)f[0, 1, 3] \\
&= 1 + 2x + 8x(x-1) = 8x^2 - 6x + 1.
\end{aligned}$$

Example 2.9. Construct the divided difference table for the data

x	0.5	1.5	3.0	5.0	6.5	8.0
$f(x)$	1.625	5.875	31.0	131.0	282.125	521.0

Hence, find the interpolating polynomial and an approximation to the value of $f(7)$.

We have the following divided difference table

x	$f(x)$	<i>first order d.d</i>	<i>second order d.d</i>	<i>third order d.d</i>	<i>fourth order d.d</i>
0.5	1.625				
		4.25			
1.5	5.875		5.0		
		16.76		1.0	
3.0	31.000		9.5		0
		50.00		1.0	
5.0	131.000		14.5		0
		100.75		1.0	
6.5	282.125		19.5		
		159.25			
8.0	521.000				

We write the divided difference interpolating polynomial as

$$f(x) = f[x_0] + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2]$$

$$\begin{aligned}
& +(x-x_0)(x-x_1)(x-x_2)f[x_0, x_1, x_2, x_3] \\
& = 1.625+(x-0.5)(4.25)+5(x-0.5)(x-1.5)+(x-0.5)(x-1.5)(x-3.0) \\
& = (1.625-2.125+3.75-2.25)+x(4.25-10.0+6.75)+x^2(5-5)+x^3 \\
& = x^3+x+1.
\end{aligned}$$

Hence, $f(7.0) = 351$.

Example 2.10. Calculate the n^{th} divided difference of $1/x$, based on the points $x_0, x_1, x_2, \dots, x_n$.

$$\text{We have } f[x_0, x_1] = \frac{(1/x_1)-(1/x_0)}{x_1-x_0} = -\frac{1}{x_0x_1}.$$

$$f[x_0, x_1, x_2] = \frac{[-1/(x_1x_2)+1/x_0x_1]}{x_2-x_0} = -\frac{x_2-x_0}{x_0x_1x_2(x_2-x_0)} = \frac{(-1)^2}{x_0x_1x_2}$$

$$\text{We shall show that } f[x_0, x_1, \dots, x_n] = \frac{(-1)^n}{x_0x_1\dots x_n}.$$

Let the result be true for x_0, x_1, \dots, x_k . That is

$$f[x_0, x_1, \dots, x_k] = \frac{(-1)^k}{x_0x_1\dots x_k}.$$

Then,

$$\begin{aligned}
f[x_0, x_1, \dots, x_{k+1}] & = \frac{1}{(x_{k+1}-x_0)} \{f[x_1, x_2, \dots, x_{k+1}] - f[x_0, x_1, \dots, x_k]\} \\
& = \frac{1}{(x_{k+1}-x_0)} \left[\frac{(-1)^k}{x_1x_2\dots x_{k+1}} - \frac{(-1)^k}{x_0x_1\dots x_k} \right] = \frac{(-1)^{k+1}}{x_0x_1\dots x_{k+1}}
\end{aligned}$$

Hence, by induction, we obtain

$$f[x_0, x_1, \dots, x_n] = \frac{(-1)^n}{x_0x_1\dots x_n}.$$

3 Numerical solutions for differential equations

3.1 Taylor Series Method

The fundamental numerical method for the solution of $u' = f(t, u)$, $u(t_0) = \eta_0$, $t \in [t_0, b]$, is the Taylor series method.

We assume that the function $u(t)$ can be expanded in Taylor series about any point t_j , that is,

$$u(t) = u(t_j) + (t - t_j)u'(t_j) + \frac{1}{2!}(t - t_j)^2u''(t_j) + \dots + \frac{1}{p!}(t - t_j)^pu^{(p)}(t_j) + \frac{1}{(p+1)!}(t - t_j)^{p+1}u^{(p+1)}(t_j + \theta h). \quad (3.1)$$

This expansion holds for $t \in [t_0, b]$ and $0 < \theta < 1$.

Substituting $t = t_{j+1}$ in (3.1), we get

$$\begin{aligned} u(t_{j+1}) &= u(t_j) + hu'(t_j) + \frac{h^2}{2!}u''(t_j) + \dots + \frac{1}{p!}h^pu^{(p)}(t_j) \\ &\quad + \frac{1}{(p+1)!}h^{p+1}u^{(p+1)}(t_j + \theta h) \\ &= u(t_j) + h\phi(t_j, u(t_j), h) + \frac{1}{(p+1)!}h^{p+1}u^{(p+1)}(t_j + \theta h) \end{aligned}$$

where $h\phi(t_j, u(t_j), h) = hu'(t_j) + \frac{h^2}{2!}u''(t_j) + \dots + \frac{h^p}{p!}u^{(p)}(t_j)$.

Denote by $h\phi(t_j, u_j, h)$, the value obtained from $h\phi(t_j, u(t_j), h)$ by using an approximate value u_j in place of the exact value $u(t_j)$. Neglecting the error term, we have the method

$$u_{j+1} = u_j + h\phi(t_j, u_j, h), j = 0, 1, \dots, N - 1 \quad (3.2)$$

to approximate $u(t_{j+1})$. The error or the truncation error of the method is given

by

$$T_{j+1} = \frac{1}{(p+1)!} h^{p+1} u^{(p+1)}(t_j + \theta h) \quad (3.3)$$

The method (3.2) is called the *Taylor series method* of order p . Substituting $p = 1$ in (3.2) we get

$$u_{j+1} = u_j + hu'_j = u_j + hf(t_j, u_j).$$

which is the Euler method. Therefore, Euler method can also be called as the Taylor series method of order 1.

To apply (3.2), it is necessary to know $u(t_j)$, $u'(t_j), \dots, u^{(p)}(t_j)$. If t_j and $u(t_j)$ are known, then the derivatives can be calculated as follows:

First, the known values t_j and $u(t_j)$ are substituted into the differential equation to give

$$u'(t_j) = f(t_j, u(t_j)).$$

Next, the differential equation $u' = f(t, u)$ is differentiated to obtain expressions for the higher order derivatives of $u(t)$. Thus, we have

$$u' = f(t, u).$$

$$u'' = f_t + f f_u$$

$$u''' = f_u + 2f f_{tu} + f^2 f_{uu} + f_u(f_t + f f_u)$$

.....

where f_t, f_u, \dots represents the partial derivatives of f with respect to t and u and so on. The values $u''(t_j), u'''(t_j), \dots$ can be computed by substituting $t = t_j$. Therefore, if t_j and $u(t_j)$ are known exactly, then (3.2) can be used to compute u_{j+1} with an error

$$\frac{h^{p+1}}{(p+1)!} u^{(p+1)}(t_j + \theta h).$$

The number of terms to be included in (3.2) is fixed by the permissible error. If this error is ϵ and the series is truncated at the term $u^{(p)}(t_j)$ then

$$h^{p+1} |u^{(p+1)}(t_j + \theta h)| < (p+1)! \epsilon$$

or

$$h^{p+1} |f^{(p)}(t_j + \theta h)| < (p+1)! \epsilon. \quad (3.4)$$

We assume that an estimate of $|f^{(p)}(t_j + \theta h)|$ is known.

For a given h and ϵ , (3.4) will determine p , and if p and ϵ are specified, then it will give an upper bound on h .

Since $t_j + \theta h$ is not known, $|f^{(p)}(t_j + \theta h)|$ in (3.4) is replaced by its maximum value in $[t_0, b]$. A way of determining this value is as follows. Write one more non-vanishing term in the series than is required and then differentiate this series p times. The maximum value of this quantity in $[t_0, b]$ gives a rough required bound.

Example 3.1. Given the initial value problem

$$u' = t^2 + u^2, u(0) = 0$$

determine the first three non-zero terms in the Taylor series for $u(t)$ and hence obtain the value for $u(1)$. Also determine t when the error in $u(t)$ obtained from the first two non-zero terms is to be less than 10^{-6} after rounding.

We have

$$u(0) = 0, u'(0) = 0$$

$$u'' = 2t + 2uu', u''(0) = 0$$

$$u''' = 2 + 2(uu'' + (u')^2), u'''(0) = 2$$

$$u^{(4)} = 2(uu''' + 3u'u''), \quad u^{(4)}(0) = 0$$

$$u^{(5)} = 2[uu^{(4)} + 4u'u''' + 3(u'')^2], \quad u^{(5)}(0) = 0$$

$$u^{(6)} = 2(uu^{(5)} + 5u'u^{(4)} + 10u''u'''), \quad u^{(6)}(0) = 0$$

$$u^{(7)} = 2(uu^{(6)} + 6u'u^{(5)} + 15u''u^{(4)} + 10(u''')^2), \quad u^{(7)}(0) = 80$$

$$u^{(8)} = u^{(9)} = u^{(10)} = 0$$

$$u^{(11)} = 2[uu^{(10)} + 10u'u^{(9)} + 45u''u^{(8)} + 120u'''u^{(7)} + 210u^{(4)}u^{(6)} + 126(u^{(5)})^2]$$

$$u^{(11)}(0) = 38400$$

Thus the Taylor series for $u(t)$ becomes

$$u(t) = \frac{1}{3}t^3 + \frac{1}{63}t^7 + \frac{2}{2079}t^{11}.$$

The approximate value of $u(1)$ is given by

$$u(1) = \frac{1}{3} + \frac{1}{63} + \frac{2}{2079} = 0.350168.$$

If only the first two terms are used, then the value of t is obtained from

$$\left| \frac{2}{2079}t^{11} \right| < 0.5 \times 10^{-7}$$

Solving, we get $t = 0.41$.

Example 3.2. Find the three term Taylor series solution for the third order initial value problem

$$W''' + WW'' = 0, \quad W(0) = 0, \quad W'(0) = 0, \quad W''(0) = 1.$$

Find the bound on the error for $t \in [0, 0.2]$.

We find

$$W''' = -WW'', \quad W'''(0) = 0$$

$$W^{(4)} = -(WW'''' + W'W'''), \quad W^{(4)}(0) = 0$$

$$W^{(5)} = -[WW^{(4)} + 2W'W''' + (W'')^2], \quad W^{(5)}(0) = -1$$

$$W^{(6)}(0) = 0, \quad W^{(7)}(0) = 0, \quad W^{(8)}(0) = 11$$

$$W^{(9)}(0) = W^{(10)}(0) = 0, \quad W^{(11)}(0) = -375.$$

The Taylor series solution is

$$W(t) = \frac{t^2}{2!} - \frac{t^5}{5!} + \frac{11}{8!}t^8 + E_8$$

where $|E_8| \leq \max |W^{(9)}(t)| \frac{t^9}{9!}.$

Writting the next term, we have

$$W(t) = \frac{t^2}{2!} - \frac{t^5}{5!} + \frac{11}{8!}t^8 - \frac{375}{11!}t^{11}$$

We find $W^{(9)}(t) = -\frac{375}{2}t^2$ and $\max_{0 \leq t \leq 0.2} |W^{(9)}(t)| = 7.5.$

Hence, $|E_8| \leq \frac{7.5(0.2)^9}{9!} \leq (1.06)10^{-11}.$

3.2 Picard's Method

Integrating the differential equation $\frac{dy}{dx} = f(x, y)$, $x > x_0$ from x_0 to a general point x and using $y = y_0$, at $x = x_0$, the problem is transformed to an integral equation,

$$y(x) - y(x_0) = \int_{x_0}^x f(x, y)dx. \tag{3.5}$$

Solution to (3.5) is obtained in an iterative manner according to the scheme,

$$y^{(n+1)}(x) = y_0 + \int_{x_0}^x f[x, y^{(n)}(x)]dx, \quad n = 0, 1, 2, \dots \tag{3.6}$$

where $y^{(n)}(x)$ denotes n th iteration and $y_0 = y(x_0)$. To start the process, the initial approximation $y^{(0)}(x)$ may be taken as y_0 .

Since the solution is obtained as a function of x , the difference between two successive approximations will also be a function of x . Hence the accuracy of the solution will be dependent on the value of x , and the solution obtained will be valid for certain range of x , for the prescribed accuracy. It may be mentioned that if the solution is obtained in the form of a series with alternating signs, the first neglected term gives the magnitude of the maximum error when the series converges uniformly in a certain interval.

The iterative process converges to the true solution under certain conditions. The major drawback of the method is that integration has to be performed at each stage which may not be possible when the integrand is complicated. The method is not a numerical method; it may be called semi-analytical or approximate analytical method.

Example 3.3. Find the approximate solution by Picard's method for the differential equation,

$$\frac{dy}{dx} = x^2 - y, \quad y(0) = 1$$

which is correct within an accuracy of 10^{-3} for $0 \leq x \leq 0.2$.

Solution: The iterative scheme for the above problem is,

$$y^{(n+1)} - 1 = \int_0^x (x^2 - y^{(n)}) dx.$$

Taking initial estimate $y^{(0)}$ as $y(0) = 1$,

$$y^{(1)} = 1 + \int_0^x (x^2 - 1) dx = 1 - x + \frac{x^3}{3}.$$

$$y^{(2)} = 1 + \int_0^x \left\{ x^2 - \left(1 - x + \frac{x^3}{3} \right) \right\} dx$$

$$= 1 - x + \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{12}$$

$$\text{error} = y^{(2)} - y^{(1)} = \frac{x^2}{2} - \frac{x^4}{12}$$

max error is given at $x = 0.2$, i.e.,

$$\frac{(0.2)^4}{2} - \frac{(0.2)^4}{12} > 10^{-3}$$

Since the error is more than the prescribed accuracy, we go for the next iteration,

$$\begin{aligned} y^{(3)} &= 1 + \int_0^x \left\{ x^2 - \left(1 - x + \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{12} \right) \right\} dx \\ &= 1 - x + \frac{x^2}{2} + \frac{x^3}{6} - \frac{x^4}{12} + \frac{x^5}{60} \\ \text{error} &= y^{(3)} - y^{(2)} = -\frac{x^3}{6} - \frac{x^5}{60} \end{aligned}$$

at $x = 0.2$, $\text{max error} = -\frac{(0.2)^3}{6} + \frac{(0.2)^5}{60} > 10^{-3}$

Again, we go for the next iteration, giving,

$$\begin{aligned} y^{(4)} &= 1 - x + \frac{x^2}{2} + \frac{x^3}{6} - \frac{x^4}{24} + \frac{x^5}{60} - \frac{x^6}{360} \\ \text{max error} &= y^{(4)} - y^{(3)} = \frac{x^4}{24} - \frac{x^6}{360} < 10^{-3} \end{aligned}$$

Therefore $y^{(4)}$ is the desired solution which will give an error less than 10^{-3} for values of x in the interval $(0, 0.2)$.

(The analytical solution is $y(x) = x^2 - 2x + 3 - 2e^{-x}$)

3.3 Euler's Method

$$u_{j+1} = u_j + hf_j, j = 0, 1, 2, \dots, N - 1 \tag{3.7}$$

where $f_j = f(t_j, u_j)$. This is called the *Euler's Method*. Applying the method at the mesh points $t_j, j = 0, 1, 2, \dots, N - 1$, we obtain the numerical solution of (3.7) as

$$\begin{aligned} u_1 &= u_0 + hf_0 \\ u_2 &= u_1 + hf_1 \\ &\dots\dots\dots \\ u_N &= u_{N-1} + hf_{N-1}. \end{aligned}$$

Choosing a substitute value of h and the initial condition, u_1 is obtained from the first equation, u_2 is obtained from the second equation and so on. The method (3.7) is called an *explicit method*, since using u_j , h and f_j we can calculate u_{j+1} directly. The Euler method is the simplest method to solve (3.7).

The truncation error in the method is given by

$$\begin{aligned} T_{j+1} &= u(t_{j+1}) - u_{j+1} \\ &= u(t_{j+1}) - [u(t_j) + hf(t_j, u_j)] = \frac{h^2}{2}u''(\xi) \end{aligned}$$

where $t_j < \xi < t_{j+1}$. If we denote $\max_{[t_0, b]} |T_{j+1}| = T$ and $\max_{[t_0, b]} |u''(\xi)| = M_2$, then, we have

$$T \leq \frac{h^2}{2}M_2 \quad (3.8)$$

The local truncation error is of $O(h^2)$ as $h \rightarrow 0$.

Example 3.4. Use the Euler method to solve numerically the initial value problem

$$u' = -2tu^2, \quad u(0) = 1$$

with $h = 0.2, 0.1$ and 0.05 on the interval $[0, 1]$. Neglecting the roundoff errors, determine the bound for the error. Apply Richardson's extrapolation to improve the computed value $u(1, 0)$

We have

$$u_{j+1} = u_j - 2ht_j u_j^2; \quad j = 0, 1, 2, 3, 4$$

with $h = 0.2$. The initial condition gives $u_0 = 1$

For $j = 0$; $t_0 = 0$, $u_0 = 1$

$$u(0.2) = u_1 = u_0 - 2ht_0 u_0^2 = 1.0.$$

For $j = 1$; $t_1 = 0.2$, $u_1 = 1$

$$\begin{aligned}
u(0.4) &= u_2 = u_1 - 2ht_1u_1^2 \\
&= 1 - 2(0.2)(0.2)(1)^2 = 0.92.
\end{aligned}$$

For $j = 2$; $t_2 = 0.4$, $u_2 = 0.92$

$$\begin{aligned}
u(0.6) &= u_3 = u_2 - 2ht_2u_2^2. \\
&= 0.92 - 2(0.2)(0.4)(0.92)^2 = 0.78458.
\end{aligned}$$

Similarly, we get

$$u(0.8) = u_4 = 0.63684, \quad u(1) = u_5 = 0.50706.$$

When $h = 0.1$, we get

for $j = 0$: $t_0 = 0$, $u_0 = 1$

$$u(0.1) = u_1 = u_0 - 2ht_0u_0^2 = 1.0.$$

for $j = 1$: $t_1 = 0.1$, $u_1 = 1$

$$\begin{aligned}
u(0.2) &= u_2 = u_1 - 2ht_1u_1^2. \\
&= 1 - 2(0.1)(0.1)(1)^2 = 0.98.
\end{aligned}$$

for $j = 2$: $t_2 = 0.2$, $u_2 = 0.98$

$$\begin{aligned}
u(0.3) &= u_3 = u_2 - 2ht_2u_2^2. \\
&= 0.98 - 2(0.1)(0.2)(0.98)^2 = 0.94158.
\end{aligned}$$

Similarly, we get

$$\begin{aligned}
u(0.4) &= u_4 = 0.88839, \quad u(0.5) = u_5 = 0.82525, \\
u(0.6) &= u_6 = 0.75715, \quad u(0.7) = u_7 = 0.68835, \\
u(0.8) &= u_8 = 0.62202, \quad u(0.9) = u_9 = 0.56011, \\
u(1.0) &= u_{10} = 0.50364.
\end{aligned}$$

For $h = 0.05$, we get

$$\begin{aligned}
u(0.05) &= 1.0, \quad u(0.1) = 0.995, \\
u(0.15) &= 0.9851, \quad u(0.2) = 0.97054, \\
u(0.25) &= 0.9517, \quad u(0.3) = 0.92906, \\
u(0.35) &= 0.90316, \quad u(0.4) = 0.87461, \\
u(0.45) &= 0.84401, \quad u(0.5) = 0.81195, \\
u(0.55) &= 0.77899, \quad u(0.6) = 0.74561, \\
u(0.65) &= 0.71225, \quad u(0.7) = 0.67928, \\
u(0.75) &= 0.64698, \quad u(0.8) = 0.61559, \\
u(0.85) &= 0.58527, \quad u(0.9) = 0.55615, \\
u(0.95) &= 0.52831, \quad u(1.0) = 0.50179.
\end{aligned}$$

The truncation error in the Euler method is given by

$$TE = \frac{h^2}{2} u''(\xi)$$

$$|TE| = \frac{h^2}{2} |u''(\xi)| \leq \frac{h^2}{2} \max_{0 \leq t \leq 1} |u''(t)|.$$

Since the exact solution is $u(t) = 1/(1+t^2)$, we get

$$|TE| = \frac{h^2}{2} \max_{0 \leq t \leq 1} \left| \frac{2(1-3t^2)}{(1+t^2)^3} \right| \leq 2h^2.$$

as the absolute maximum of $(1-3t^2)$ in $[0, 1]$ is 2.

The error in Euler method is of the form

$$u(t_j) - u_j(h) = c_1 h + c_2 h^2 + c_3 h^3 + \dots$$

Richardson's extrapolation gives

$$u^{(k)}(h) = \frac{2^k u_j^{(k-1)}(h/2) - u_j^{(k-1)}(h)}{2^k - 1}.$$

We have the following extrapolated value for $u(1.0)$.

Table 1.3: Extrapolated value for $u(1.0)$

h	$u^{(0)}(h)$	$u^{(1)}(h)$	$u^{(2)}(h)$	$u^{(3)}(h)$
0.20	0.50706			
		0.50022		
0.10	0.50364		0.49985	0.5
		0.49994		
0.05	0.50179			

Example 3.5. Show that in Euler method the bound of the truncation error, when applied to the test equation $u' = \lambda u$, $u(a) = B$, $\lambda > 0$, can be written as

$$|u(t_j) - u(t_j, h)| \leq \frac{hM}{2\lambda} [exp\lambda(t_j - a) - 1]$$

where $M = \max |u''(t)|$. Generalise the result when applied to the problem $u' = f(t, u)$, $u(a) = B$.

Applying the Euler method to the test equation $u' = \lambda u$, we get

$$u_{j+1} = u_j + \lambda h u_j = (1 + \lambda h)u_j$$

The exact solution satisfies the equation

$$u(t_{j+1}) = (1 + \lambda h)u(t_j) + T_{j+1}$$

where T_{j+1} is the truncation error given by $T_{j+1} = [h^2 u''(\xi)]/2$. Subtracting the

two equations and setting $\epsilon_j = u(t_j) - u_j$, we obtain

$$u(t_{j+1}) - u_{j+1} = (1 + \lambda h) [u(t_j) - u_j] + T_{j+1}$$

or
$$\epsilon_{j+1} = (1 + \lambda h)\epsilon_j + T_{j+1}$$

Hence,
$$|\epsilon_{j+1}| \leq |1 + \lambda h| |\epsilon_j| + |T_{j+1}|.$$

Let $A = |1 + \lambda h|$. Now,

$$|T_{j+1}| = \frac{h^2}{2} |u''(\xi)| \leq \frac{h^2}{2} M$$

where $M = \max_{[a,b]} |u''(t)|$. Denote $E_{j+1} = \max |\epsilon_{j+1}|$. Then, we have

$$E_{j+1} \leq AE_j + T, \text{ where } T = h^2M/2.$$

Setting $j = 0, 1, 2, \dots$, we get

$$E_1 \leq AE_0 + T$$

$$E_2 \leq AE_1 + T = A^2E_0 + (1 + A)T$$

$$E_3 \leq AE_2 + T = A^3E_0 + (1 + A + A^2)T$$

.....

$$E_j \leq A^jE_0 + (1 + A + A^2 + \dots + A^{j-1})T$$

$$= A^jE_0 + \left(\frac{A^j-1}{A-1}\right)T, \text{ where } A \neq 1.$$

Let $E_0 = 0$, that is, there is no initial error. Now,

$$(1 + h\lambda)^j < \exp[\lambda jh] < \exp[\lambda(t_j - a)], \lambda > 0.$$

Since $\lambda > 0$, we get

$$E_j \leq \frac{h^2M}{2(h\lambda)} [\exp\{\lambda(t_j - a)\} - 1] = \frac{hM}{2\lambda} [\exp\{\lambda(t_j - a)\} - 1]$$

Since, $\max |\epsilon_j| = E_j$, we get

$$|\epsilon_j| \leq |u(t_j) - u(t_j, h)| \leq \frac{hM}{2\lambda} [\exp\lambda(t_j - a) - 1]$$

Since $\lambda = \partial(\lambda u)/\partial u$, the result can be generalised as

$$|u(t_j) - u(t_j, h)| \leq \frac{hM}{2L} [\exp\lambda(t_j - a) - 1],$$

where $|\partial f/\partial y| \leq L$.

Example 3.6. The system

$$y' = z$$

$$z' = -by - az$$

where $0 < a < 2\sqrt{b}$, $b > 0$ is to be integrated by Euler method with known initial values. What is the largest step length h for which all solutions of the corresponding difference equation are bounded?

Applying the Euler method, we obtain

$$y_{j+1} = y_j + hz_j$$

$$\begin{aligned} z_{j+1} &= z_j + h(-by_j - az_j) \\ &= -bhy_j + (1 - ah)z_j \end{aligned}$$

which may be written as

$$\begin{bmatrix} y_{j+1} \\ z_{j+1} \end{bmatrix} = \begin{bmatrix} 1 & h \\ -bh & 1 - ah \end{bmatrix} \begin{bmatrix} y_j \\ z_j \end{bmatrix}$$

or

$$\mathbf{u}_{j+1} = \mathbf{A}\mathbf{u}_j$$

where

$$\mathbf{u}_j = \begin{bmatrix} y_j & z_j \end{bmatrix}^T \quad \text{and} \quad \mathbf{A} = \begin{bmatrix} 1 & h \\ -bh & 1 - ah \end{bmatrix}$$

The characteristic equation of the matrix \mathbf{A} is given by

$$\xi^2 - (2 - ah)\xi + 1 - ah + bh^2 = 0.$$

Putting $\xi = (1 + z)/(1 - z)$ in the characteristic equation, we get the reduced characteristic equation as

$$(4 - 2ah + bh^2z^2 + 2h(a - bh)z + bh^2 = 0.$$

The roots of the characteristic equation will lie within the unit circle or that of the reduced characteristic equation on the left half plane of the z -plane, if and only if the following conditions will be satisfied:

$$4 - 2ah + bh^2 > 0, \quad a - bh > 0, \quad bh^2 > 0.$$

The first inequality can be written as

$$(2 - \sqrt{bh})^2 + 2h(2\sqrt{b} - a) > 0.$$

The second inequality gives $h < a/b$. The third inequality is satisfied since $b > 0$. Hence, we obtain $h < a/b$

For $h = a/b$, or $a = bh$, we get the characteristic equation as

$$\xi^2 - (2 - bh^2)\xi + 1 = 0.$$

The roots are $\xi = \left[(2 - bh^2) \pm \sqrt{(2 - bh^2)^2 - 4} \right] / 2$.

Since $b > 0$, $(2 - bh^2)^2 - 4 < 0$, the roots are a complex pair and $|\xi| = 1$. Thus, the largest step length is given by $h \leq a/b$.

UNIT - IV

1 Runge - Kutta Fourth Order Method

Consider now, the Runge - Kutta fourth order methods defined by

$$u_{j+1} = u_j + W_1K_1 + W_2K_2 + W_3K_3 + W_4K_4 \quad (1.1)$$

where

$$K_1 = hf(t_j, u_j)$$

$$K_2 = hf(t_j + c_2h, u_j + a_{21}K_1)$$

$$K_3 = hf(t_j + c_3h, u_j + a_{31}K_1 + a_{32}K_2)$$

$$K_4 = hf(t_j + c_4h, u_j + a_{41}K_1 + a_{42}K_2 + a_{43}K_3)$$

along with the conditions

$$c_2 = a_{21}$$

$$c_3 = a_{31} + a_{32}$$

$$c_4 = a_{41} + a_{42} + a_{43}$$

$$W_1 + W_2 + W_3 + W_4 = 1$$

$$W_2c_2 + W_3c_3 + W_4c_4 = \frac{1}{2}$$

$$W_2c_2^2 + W_3c_3^2 + W_4c_4^2 = \frac{1}{3}$$

$$W_3c_2a_{32} + W_4(c_2a_{42} + c_3a_{43}) = \frac{1}{6}$$

$$W_2c_2^3 + W_3c_3^3 + W_4c_4^3 = \frac{1}{4}$$

$$W_3c_2^2a_{32} + W_4(c_2^2a_{42} + c_3^2a_{43}) = \frac{1}{12}$$

$$\begin{aligned}
W_3c_2c_3a_{32} + W_4(c_2a_{42} + c_3a_{43}) &= \frac{1}{8} \\
W_4c_2a_{32}a_{43} &= \frac{1}{24}.
\end{aligned} \tag{1.2}$$

Applying the method 1.1 to the test equation $u' = \lambda u$, we get

$$\begin{aligned}
K_1 &= h\lambda u_j = \bar{h}u_j, \text{ where } \bar{h} = h\lambda, \\
K_2 &= h\lambda(u_j + a_{21}K_1) = \bar{h}[1 + c_2\bar{h}]u_j \\
K_3 &= h\lambda(u_j + a_{31}K_1 + a_{32}K_2) \\
&= \bar{h}[u_j + a_{31}\bar{h}u_j + a_{32}\bar{h}(1 + c_2\bar{h})u_j] \\
&= \bar{h}[1 + a_{31}\bar{h} + a_{32}\bar{h} + a_{32}c_2\bar{h}^2]u_j \\
&= \bar{h}[1 + c_3\bar{h} + a_{32}c_2\bar{h}^2]u_j. \\
K_4 &= h\lambda(u_j + a_{41}K_1 + a_{42}K_2 + a_{43}K_3) \\
&= \bar{h}[u_j + a_{41}\bar{h}u_j + a_{42}\bar{h}(1 + c_2\bar{h})u_j + a_{43}\bar{h}(1 + c_3\bar{h} + a_{32}c_2\bar{h}^2)u_j] \\
&= \bar{h}[1 + (a_{41} + a_{42} + a_{43})\bar{h} + (a_{42}c_2 + a_{43}c_3)\bar{h}^2 + a_{43}a_{32}c_2\bar{h}^3]u_j \\
&= \bar{h}[1 + c_4\bar{h} + (a_{42}c_2 + a_{43}c_3)\bar{h}^2 + a_{43}a_{32}c_2\bar{h}^3]u_j
\end{aligned}$$

and

$$\begin{aligned}
u_{j+1} &= u_j + W_1K_1 + W_2K_2 + W_3K_3 + W_4K_4 \\
&= u_j + W_1\bar{h}u_j + W_2\bar{h}(1 + c_2\bar{h})u_j + W_3\bar{h}[1 + (c_3\bar{h} + a_{32}c_2\bar{h}^2)]u_j + W_4\bar{h}[1 + \\
&\quad c_4\bar{h} + (a_{42}c_2 + a_{43}c_3)\bar{h}^2 + a_{43}a_{32}c_2\bar{h}^3]u_j \\
&= [1 + \bar{h}(W_1 + W_2 + W_3 + W_4) + \bar{h}^2(W_2c_2 + W_3c_3 + W_4c_4) + \bar{h}^3\{W_3a_{32}c_2 + \\
&\quad W_4(a_{42}c_2 + a_{43}c_3)\} + W_4a_{43}a_{32}c_2\bar{h}^4]u_j \\
u_{j+1} &= [1 + \bar{h} + \frac{\bar{h}^2}{2} + \frac{\bar{h}^3}{6} + \frac{\bar{h}^4}{24}]u_j = E(\lambda h)u_j
\end{aligned} \tag{1.3}$$

Therefore, the propagation factor of the fourth order methods is independent of the arbitrary parameters. Hence, the stability intervals or regions of all the fourth order methods is same. For, absolute stability, we require

$$|E(\lambda h)| = |1 + \bar{h} + \frac{\bar{h}^2}{2} + \frac{\bar{h}^3}{6} + \frac{\bar{h}^4}{24}| < 1. \tag{1.4}$$

When λ is real and $\lambda < 0$, we obtain the stability interval as $\lambda h \in (-2.78, 0)$, (see $|E(\lambda h)| = |1 + \lambda h + \frac{1}{2}\lambda^2 h^2 + \frac{1}{6}\lambda^3 h^3 + \frac{1}{24}\lambda^4 h^4| < 1$).

When λh is pure imaginary, set $\lambda = iy$. Then, 1.4 gives

$$|1 + i(yh) - \frac{(yh)^2}{2} - i\frac{(yh)^3}{6} + \frac{(yh)^4}{24}| < 1$$

or $(1 - \frac{t^2}{2} + \frac{t^4}{24}) + (t - \frac{t^3}{6}) < 1$, where $t = yh$

or $1 - \frac{t^4}{72} + \frac{t^8}{576} < 1$.

This equation is satisfied for $|t| < 2\sqrt{2}$. Hence, the stability interval in this case is $0 < |\lambda h| < 2\sqrt{2}$.

When λ is complex, it is difficult to derive the stability region analytically. We set $\lambda = x + iy$ in 1.4 and plot the boundary of the region (plot the real and imaginary parts). Stability region is plotted in Figure 4.1.

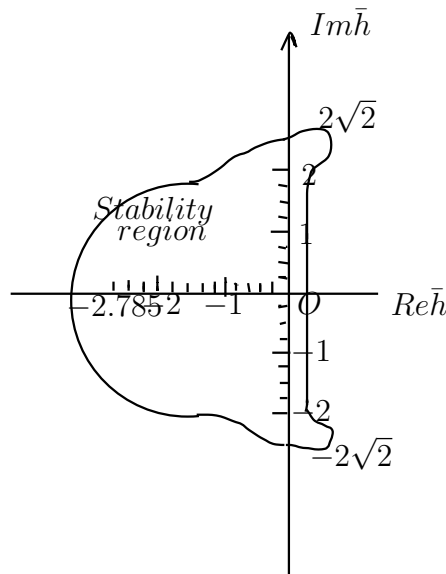


Figure.4.1. Stability region for the fourth order Runge-Kutta method, $\bar{h} = h\lambda$

It is easy to verify that the Euler method , backward Euler method and all Runge-Kutta methods are relatively stable since

$$|E(\lambda h)| \leq e^{\lambda h}, \lambda > 0$$

in all these cases.

Consider now, the stability of the implicit method

$$u_{j+1} = u_j + K_1$$

$$u_1 = hf(t_j + \frac{1}{2}h, u_j + \frac{1}{2}K_1).$$

Applying it on the test equation $u' = \lambda u$, we get

$$K_1 = h\lambda[u_j + \frac{1}{2}K_1, \text{ or } K_1 = \frac{h\lambda}{1-(h\lambda)/2}u_j$$

$$u_{j+1} = [1 + \frac{h\lambda}{1-(h\lambda)/2}]u_j = [\frac{1+(h\lambda)/2}{1-(h\lambda)/2}]u_j$$

Hence, $|E(h\lambda)| = |\frac{1+(h\lambda)/2}{1-(h\lambda)/2}|$.

Since, for real $\lambda < 0$, $|E(h\lambda)| < 1$ always, the method is stable for all $\lambda h \in (-\infty, 0)$. Therefore, the method is also A -stable.

Example 1.1. Find the implicit Runge- Kutta method of the form

$$y_{j+1} = y_j + W_1K_1 + W_2K_2$$

$$K_1 = hf(y_j)$$

$$K_2 = hf(y_j + a(K_1 + K_2))$$

for the solution of the initial value problem $y' = f(y), y(t_0) = y_0$. Obtain the interval of absolute stability when the method is applied on $y' = \lambda y, \lambda < 0$.

We have $K_1 = hf_j$.

Since K_2 can be expanded in powers of h , we write it as

$$K_2 = hA_1 + h^2A_2 + h^3A_3 + \dots$$

where A_i 's are independent of h .

Expanding K_2 in Taylor's series we get

$$K_2 = [hf + ha(K_1 + K_2)f_y + \frac{h}{2}a^2(K_1 + K_2)^2f_{yy} + \frac{h}{6}a^3(K_1 + K_2)^3f_{yyy} + \dots]_{t_j}$$

Substituting for K_1 and K_2 we obtain

$$\begin{aligned} hA_1 + h^2A_2 + h^3A_3 + \dots &= \{hf + ha[hf + hA_1 + h^2A_2 + h^3A_3 + \dots]f_y \\ &\quad + \frac{h}{2}a^2[hf + hA_1 + h^2A_2 + \dots]^2f_{yy} \\ &\quad + \frac{h}{6}a^3[hf + hA_1 + \dots]^3f_{yyy} + \dots\}_{t_j} \end{aligned}$$

Equating the coefficients of various powers of h , we have

$$\begin{aligned} A_1 &= f_j \\ A_2 &= (2aff_y)_{t_j} \\ A_3 &= (aA_2f_y + 2a^2f_{yy}f^2)_{t_j} = (2a^2ff_{yy}^2 + 2a^2f_{yy}f^2)_{t_j} \end{aligned}$$

Hence,

$$\begin{aligned} y_{j+1} &= y_j + hy'_j + \frac{h^2}{2}y''_j + \frac{h^3}{6}y'''_j + \dots \\ &= y_j + [hf + \frac{h^2}{2}ff_y + \frac{h^3}{6}(ff_y^2 + f^2f_{yy}) + \dots]_{t_j} \\ &= y_j + W_1hf_j + W_2[hf + 2ah^2ff_y + h^3(2a^2f^2f_{yy}) + \dots]_{t_j} \end{aligned}$$

Comparing the terms corresponding to various powers of h , we obtain

$$\begin{aligned} W_1 + W_2 &= 1 \\ 2aW_2 &= \frac{1}{2} \\ 2a^2W_2 &= \frac{1}{6}. \end{aligned}$$

whose solution is given by

$$a = 1/3, W_2 = 3/4, W_1 = 1/4.$$

The implicit Runge - Kutta method becomes

$$\begin{aligned} K_1 &= nhf(y_j) \\ K_2 &= hf(y_j + \frac{1}{3}(K_1 + K_2)) \\ y_{j+1} &= y_j + \frac{1}{4}(K_1 + 3K_2). \end{aligned}$$

The order of the method is 3.

We now apply the method to the test equation $y' = \lambda y, \lambda < 0$.

We have

$$\begin{aligned} K_1 &= \bar{h}y_j \\ K_2 &= \bar{h}(y_j + \frac{1}{3}(\bar{h}y_j + K_2)) \end{aligned}$$

$$\begin{aligned}
K_2 &= \frac{1+(1/3)\bar{h}}{1-(1/3)\bar{h}}(\bar{h}y_j) \\
y_{j+1} &= y_j + \frac{1}{4}\bar{h}y_j + \frac{1}{4}\frac{1+(1/3)\bar{h}}{1-(1/3)\bar{h}}(\bar{h}y_j) \\
&= \frac{1+(2/3)\bar{h}+(1/6)\bar{h}^2}{1-(1/3)\bar{h}}y_j
\end{aligned}$$

where $\bar{h} = h\lambda$.

This is a first order difference equation and the characteristic equation is given by

$$\xi = \frac{1+(2/3)\bar{h}+(1/6)\bar{h}^2}{1-(1/3)\bar{h}}.$$

For absolute stability ($\lambda < 0$), we require, $|\xi| \leq 1$. Therefore,

$$-1 \leq \frac{1+(2/3)\bar{h}+(1/6)\bar{h}^2}{1-(\bar{h}/3)} \leq 1$$

or
$$-1 + \frac{\bar{h}}{3} \leq 1 + \frac{2}{3}\bar{h} + \frac{1}{6}\bar{h}^2 \leq 1 - \frac{\bar{h}}{3}.$$

The right inequality gives

$$\bar{h} + \frac{1}{6}\bar{h}^2 = \frac{\bar{h}}{6}(6 + \bar{h}) \leq 0.$$

Since $\bar{h} = \lambda h < 0$, we require $6 + \bar{h} \geq 0$ or $\bar{h} \geq -6$.

The left inequality

$$2 + \bar{h} + \frac{1}{6}\bar{h}^2 \geq 0$$

is satisfied for $\bar{h} \geq -6$. Hence, the stability interval is $(-6, 0)$.

2 PREDICTOR - CORRECTOR METHODS

2.1 $P(EC)^mE$ Method

We shall now discuss the application of the explicit and implicit multistep methods for the initial value problems. We use the explicit (predictor) method for predicting a value $u_{j+1}^{(0)}$ and then use the implicit (corrector) method iteratively

until the convergence is obtained. We consider the predictor - corrector set

$$P : u_{j+1}^{(0)} = \sum_{i=1}^k a_i^{(0)} u_{j-i+1} + h \sum_{i=1}^k b_i^{(0)} f_{j-i+1} \quad (2.1)$$

$$C : u_{j+1}^{s+1} = \sum_{i=1}^k a_i u_{j-i+1} + hb_0 f(t_{j+1}, u_{j+1}^{s+1}) + h \sum_{i=1}^k b_i f_{j-i+1}, i = 0, 1, 2, \dots \quad (2.2)$$

to solve the initial value problem

$$u' = f(t, u), u(t_0) = u_0.$$

The predictor- corrector method may be written as

P : Predict some value $u_{j+1}^{(0)}$

E : Evaluate $f(t_{j+1}, u_{j+1}^{(0)})$

C : Correct

$$u_{j+1}^{(1)} = \sum_{i=1}^k (a_i u_{j-i+1} + hb_i f_{j-i+1}) + hb_0 f(t_{j+1}, u_{j+1}^{(1)})$$

.....

The sequence of operations PECECE... , where corrector C is applied m times, is denoted by $P(EC)^m E$ and is called a **predictor-corrector** method.

We shall now prove a sufficient condition for the convergence of $P(EC)^m E$ scheme.

Theorem 2.1. Let $\{u_{j+1}^{(m)}\}$ be a sequence of approximations of u_{j+1} obtained by a PECE... method. If

$$\left| \frac{\partial f}{\partial u}(t_{j+1}, u) \right| \leq L$$

(for all u near u_{j+1} including $u_{j+1}^{(0)}, u_{j+1}^{(1)}, \dots$) where L satisfies the conditions $L < 1/|hb_0|$, then the sequence $\{u_{j+1}^{(m)}\}$ converges to u_{j+1} .

The numerical solution satisfies the equation

$$u_{j+1} = \sum_{i=1}^k a_i u_{j-i+1} + hb_0 f(t_{j+1}, u_{j+1}) + h \sum_{i=1}^k b_i f_{j-i+1}$$

The corrector satisfies the equation 2.2. Subtracting these two equations we get

$$u_{j+1} - u_{j+1}^{(s+1)} = hb_0 [f(t_{j+1}, u_{j+1}) - f(t_{j+1}, u_{j+1}^{(s)})]$$

Using the Lagrange mean value theorem, we get

$$u_{j+1} - u_{j+1}^{(s+1)} = hb_0 (u_{j+1} - u_{j+1}^{(s)}) \frac{\partial f}{\partial u}(t_{j+1}, u^*)$$

where $u_{j+1}^{(s)} \leq u^* \leq u_{j+1}$. Hence,

$$\begin{aligned} |u_{j+1} - u_{j+1}^{(s+1)}| &\leq |hb_0| |u_{j+1} - u_{j+1}^{(s)}| \left| \frac{\partial f}{\partial u}(t_{j+1}, u) \right| \\ &\leq hL|b_0| |u_{j+1} - u_{j+1}^{(s)}| \\ &\leq hL|b_0|^s |u_{j+1} - u_{j+1}^{(0)}| \end{aligned}$$

Now, $\lim_{s \rightarrow \infty} |u_{j+1} - u_{j+1}^{(s+1)}| \rightarrow 0$, if

$$hL|b_0| < 1 \text{ or } L < \frac{1}{h|b_0|}.$$

We have the following examples of Adams- Moulton methods.

Order 2: $u_{j+1} = u_j + \frac{h}{2}[f_{j+1} + f_j]$
 $b_0 = 1/2, hL < 2.$

Order 3: $u_{j+1} = u_j + \frac{h}{12}[5f_{j+1} + 8f_j - f_{j-1}]$
 $b_0 = 5/12, hL < 12/5.$

Let us illustrate $P(EC)^m E$ method for the equation $u' = \lambda u$ and the $P-C$ set

$$P : u_{j+1} = u_j + hf_j \text{ (Euler method)}$$

$$C : u_{j+1} = u_j + \frac{h}{2}(f_{j+1} + f_j) \text{ (Euler- Cauchy or Heun method)}$$

The $P(EC)^m E$ method may be written as

$$\begin{aligned} u_{j+1}^{(0)} &= u_j + hf_j \\ u_{j+1}^{(s+1)} &= u_j + \frac{h}{2}(f_{j+1}^{(s-1)} + f_j), s = 1, 2, \dots, m. \\ u_{j+1} &= u_{j+1}^{(m)} \end{aligned}$$

$$f_{j+1} = f_{j+1}^{(m)} \tag{2.3}$$

where

$$f_{j+1}^{(s)} = f(t_{j+1}, u_{j+1}^{(s)})$$

The $P(EC)^mE$ method becomes

$$\begin{aligned} u_{j+1}^{(0)} &= (1 + \lambda h)u_j \\ u_{j+1}^{(1)} &= u_j + \frac{h}{2}[\lambda(1 + \lambda h)u_j + \lambda u_j] \\ &= [1 + \lambda h + \frac{1}{2}(\lambda h)^2]u_j \\ u_{j+1}^{(2)} &= u_j + \frac{h}{2}[\lambda(1 + \lambda h + \frac{1}{2}(\lambda h)^2)u_j + \lambda u_j] \\ &= [1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{4}]u_j \\ &\dots\dots\dots \\ u_{j+1}^{(m)} &= [1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{2^2} + \dots + \frac{(\lambda h)^{m+1}}{2^m}]u_j \\ &= [1 + \lambda h + p + p^2 + \dots + p^m]u_j \end{aligned}$$

where $p = \lambda h/2$.

Summing the series, we obtain

$$u_{j+1}^{(m)} = [1 + 2p \frac{1 - p^{m+1}}{1 - p}]u_j = \frac{1}{1 - p}[1 + p - 2p^{m+2}]u_j \tag{2.4}$$

We note that for $\lambda < 0$, $|1 + (\lambda h/2)| < |1 - (\lambda h/2)|$.

If the corrector is iterated to converge, i.e. $m \rightarrow \infty$, the equation 2.4 will converge if $|p| = |\lambda h/2| < 1$ or $|\lambda h| < 2$.

The analytical solution of the test equation may be written as

$$u(t_{j+1}) = e^{\lambda h}u(t_j).$$

The truncation error of $P(EC)^mE$ method given by

$$\begin{aligned} T_{j+1} &= u(t_{j+1}) - u_{j+1} \\ &= (e^{\lambda h} - [\frac{1 + (\lambda h/2) - 2(\lambda h/2)^{m+2}}{1 - (\lambda h/2)}])u(t_j) \end{aligned} \tag{2.5}$$

becomes

$$\frac{1}{2}(\lambda h)^2 + O(|\lambda h|^3) \text{ for 0 corrector}$$

$$\frac{1}{6}(\lambda h)^3 + O(|\lambda h|^4) \text{ for 1 corrector}$$

$$-\frac{1}{12}(\lambda h)^3 + O(|\lambda h|^4) \text{ for 2 correctors}$$

$$-\frac{1}{12}(\lambda h)^3 + O(|\lambda h|^4) \text{ for 3 correctors.}$$

We thus see that the application more than twice does not improve the result because the minimum truncation error is obtained at this stage.

We note the following:

- (i) The order of the predictor is less than or equal to the order of the corrector.
- (ii) We may not restrict to the use of only absolutely stable predictors, that is, we may, if necessary, use unstable predictors.
- (iii) If the order of the predictor is lower than the order of the corrector, then each iteration of the corrector raises the order of the $P-C$ set, until the order of the corrector is reached. If iterated further, then the error constant may reduce

Next we apply the $P-C$ set

$$P : u_{j+1} = 2u_j - u_{j-1} + h^2 f_j.$$

$$C : u_{j+1} = 2u_j - u_{j-1} + \frac{h^2}{12}(f_{j+1} + 10f_j + f_{j-1}).$$

to the second order initial value problem

$$u'' = -\omega^2 u$$

$$u(0) = 1, u'(0) = 0.$$

The $P(EC)^m E$ method may be written as

$$u_{j+1}^{(0)} = 2u_j - u_{j-1} + h^2 f_j$$

$$u_{j+1}^{(s)} = 2u_j - u_{j-1} + \frac{h^2}{12}(f_{j+1}(s-1) + 10f_j + f_{j-1})$$

$$s = 1, 2, \dots, m.$$

$$u_{j+1} = u_{j+1}^{(m)}, f_{j+1} = f_{j+1}^{(m)}$$

where $f_{j+1}^{(s)} = f(t_{j+1}, u_{j+1}^{(s)})$.

We obtain

$$\begin{aligned} u_{j+1}^{(0)} &= 2u_j - u_{j-1} - H^2 u_j \\ &= (2 - H^2)u_j - u_{j-1} \end{aligned}$$

where $H = \omega h$.

$$\begin{aligned} u_{j+1}^{(1)} &= 2u_j - u_{j-1} + \frac{h^2}{12}[-\omega^2(2 - H^{12})u_j - u_{j-1} - 10\omega^2 u_j - \omega^2 u_{j-1}] \\ &= 2u_j - u_{j-1} - \frac{H^2}{12}[(12 - H^2)u_j] \\ &= (2 - H^2 + \frac{H^2}{12})u_j - u_{j-1}. \end{aligned}$$

$$\begin{aligned} u_{j+1}^{(2)} &= 2u_j - u_{j-1} - \frac{H^2}{12}[u_{j+1}^{(2)} + 10u_j + u_{j-1}] \\ &= 2u_j - u_{j-1} - \frac{H^2}{12}[(2 - H^2 + \frac{H^4}{12})u_j - u_{j-1} + 10u_j + u_{j-1}] \\ &= [2 - H^2 + \frac{(H^2)^2}{12} - \frac{(H^2)^3}{12^2}]u_j - u_{j-1}. \end{aligned}$$

After m correctors we obtain

$$u_{j+1} - 2Bu_j + u_{j-1} = 0$$

where

$$B = \frac{1 - (5H^2/12) + 6(-1)^{m-1}(H^2/12)^{m+2}}{1 + (H^2/12)}$$

The characteristic equation of this difference equation is

$$\xi^2 - 2B\xi + 1 = 0.$$

The solution of the difference equation converges, if the roots of the characteristic equation lie inside the unit circle or if they lie on the unit circle, they are simple.

Now, since the product of the roots is equal to 1, the roots lie on the unit circle $|\xi| = 1$, only if they are a complex pair of magnitude 1. If discriminant $4(B^2 - 1) < 0$, that is $|B| < 1$, then the roots are

$$\xi = [B \pm (\sqrt{1 - B^2})i] \text{ and } |\xi| = 1.$$

For convergence, we require $|B| < 1$ as $m \rightarrow \infty$. This condition is satisfied if

$$\frac{H^2}{12} < 1 \text{ and } \left| \frac{1 - (5H^2/12)}{1 + (H^2/12)} \right| < 1.$$

The second inequality gives

$$-[1 + \frac{H^2}{12}] < 1 - \frac{H^2}{12} < 1 + \frac{H^2}{12}.$$

The right part gives $H^2 > 0$ which is true, and the left part gives $H^2 < 6$.

Hence, the condition for convergence is $H^2 < 6$.

The exact solution of the differential equation is given by $u = \cos\omega t$. We have the truncation error as

$$\begin{aligned} T_{j+1} &= u(t_{j+1}) - \bar{u}_{j+1} \\ &= u(t_{j+1}) - 2[2Bu(t_j) - u(t_{j-1})] \\ &= \cos[\omega(t_j + h)] + \cos\omega(t_j - h) - 2Bu(t_j) \\ &= 2\cos(\omega t_j)\cos(\omega h) - 2Bu(t_j) \\ &= 2\cos H - Bu(t_j). \end{aligned} \tag{2.6}$$

Now,

$$\begin{aligned} B &= [1 - \frac{5H^2}{12} + 6(-1)^{m-1}(\frac{H^2}{12})]^{m+2}[1 + \frac{H^2}{12}]^{-1} \\ &= [1 - \frac{5H^2}{12} + 6(-1)^{m-1}(\frac{H^2}{12})]^{m+2}[1 - \frac{H^2}{12} + \frac{H^4}{144} - \frac{H^6}{1728} + \dots] \\ &= [1 - \frac{H^2}{2} + \frac{H^4}{24} - \frac{H^6}{288} + \dots] + 6(-1)^{m-1}(\frac{H^2}{12})^{m+2}[1 - \frac{H^2}{12} + \dots] \end{aligned}$$

and $\cos H = 1 - \frac{H^2}{2} + \frac{H^4}{24} - \frac{H^6}{720} + \dots$

We have

$$\begin{aligned} \text{for } m = 0 : 2(\cos H - B) &= (\frac{H^4}{12}) + \dots \\ \text{for } m = 1 : 2(\cos H - B) &= -(\frac{H^6}{360}) + \dots \\ \text{for } m = 2 : 2(\cos H - B) &= (\frac{H^6}{240}) + \dots \\ \text{for } m = 3 : 2(\cos H - B) &= (\frac{H^6}{240}) + \dots \end{aligned}$$

Therefore, truncation error becomes

$$\begin{aligned} &(H^4/12) + O(H^6) \text{ for 0 corrector} \\ &-(H^6/360) + O(H^8) \text{ for 1 corrector} \\ &(H^6/240) + O(H^8) \text{ for 2 corrector} \\ &(H^6/240) + O(H^8) \text{ for 3 corrector} \end{aligned}$$

Hence, application of the corrector more than twice does not improve the result.

2.2 PM_pCM_c Method

We can use the estimate of the truncation error to modify the predicted and corrected values. Thus, we may write this procedure as PM_pCM_c . This is called the **modified predictor-corrector method**. The estimates of the local truncation error may be obtained as follows. Let the predictor 2.1 and the corrector 2.2 both have the order p . Thus we have

$$u(t_{j+1}) - u_{j+1}^{(m)} = c_{j+1}^* h^{p+1} u^{(p+1)}(t_j) + O(h^{p+2}) \quad (2.7)$$

$$u(t_{j+1}) - u_{j+1}^{(c)} = c_{j+1} h^{p+1} u^{(p+1)}(t_j) + O(h^{p+2}) \quad (2.8)$$

where $u_{j+1}^{(p)}$ and $u_{j+1}^{(c)}$ represent the solution values obtained by using the predictor and corrector respectively.

Subtracting 2.7 and 2.8 we get

$$u_{j+1}^{(p)} - u_{j+1}^{(c)} = (c_{j+1} - c_{j+1}^*) h^{p+1} u^{(p+1)}(t_j) + O(h^{p+2}). \quad (2.9)$$

Substituting the value of $h^{p+1} u^{(p+1)}(t_j)$ from 2.9 into 2.7 and 2.8 we obtain the modified predicted and corrected values m_{j+1} and u_{j+1} respectively as

$$m_{j+1} = p_{j+1} + c_{j+1}^* (c_{j+1} - c_{j+1}^*)^{-1} (p_{j+1} - C_{j+1})$$

$$u_{j+1} = C_{j+1} + c_{j+1} (c_{j+1} - c_{j+1}^*)^{-1} (p_{j+1} - C_{j+1})$$

where p_{j+1} and c_{j+1} are the predicted and corrected values respectively.

Thus the modified $P - C$ method becomes

$$\text{Predicted value : } p_{j+1} = \sum_{i=1}^k (a_i^{(0)} u_{j-i+1} + hb_i^{(0)} f_{j-i+1})$$

Modified value :

$$m_{j+1} = p_{j+1} + c_{j+1}^* (c_{j+1} - c_{j+1}^*)^{-1} (p_i - C_j) \quad (2.10)$$

Corrector value : $C_{j+1} = \sum_{i=1}^k (a_i u_{j-i+1} + hb_i u'_{j-i+1}) + hb_0 m'_{j+1}$

Final value : $u_{j+1} = C_{j+1} + c_{j+1}(c_{j+1} - c_{j+1}^*)^{-1}(p_{j+1} - C_{j+1})$.

The quantity $p_1 - c_1$ required for the modification of the first step is generally taken as

$$p_1 - C_1 = 0. \tag{2.11}$$

For example, consider the $P - C$ method

$$P : u_{j+1} = u_j + \frac{h}{2}(3u'_j - u'_{j-1})$$

$$C : u_{j+1} = u_j + \frac{h}{2}(u'_{j+1} - u'_j)$$

Thus,

$$u(t_{j+1}) - u_{j+1}^{(p)} = \frac{5}{12}h^3 u'''(t_j) + O(h^4)$$

$$u(t_{j+1}) - u_{j+1}^{(c)} = \frac{5}{12}h^3 u'''(t_j) + O(h^4)$$

Comparing with 2.7 and 2.8, we get $c_{j+1}^* = 5/12$, $c_{j+1} = -1/12$ and $c_{j+1} - c_{j+1}^* = 1/2$. Using 2.10, the modified predicted - corrected method can be written as

$$p_{j+1} = u_j + \frac{h}{2}(3u'_j - u'_{j-1})$$

$$m_{j+1} = p_{j+1} - \frac{5}{6}(p_j - C_j)$$

$$C_{j+1} = u_j + \frac{h}{2}(m'_{j+1} + u'_j)$$

$$u_{j+1} = C_{j+1} + \frac{1}{2}(p_{j+1} - C_{j+1}), j = 1, 2, \dots$$

Example 2.2. Solve the initial value problem

$$u' = -2tu^2, u(0) = 1$$

with $h = 0.2$ on the interval $[0, 0.4]$, using the $P - C$ method

$$P : u_{j+1} = u_j + \frac{h}{2}(3u'_j - u'_{j+1})$$

$$C : u_{j+1} = u_j + \frac{h}{2}(u'_{j+1} + u'_j)$$

as (i) $P(EC)^m E, m = 2$, (ii) $PM_p CM_c$.

To use the predictor, we need the values of $u(t)$ and $u'(t)$ at $t = 0.2$. The values

obtained from the exact solution $u(t) = 1.0/(1 + t^2)$ are

$$u(0.2) = u_1 = 0.09615385, u'(0.2) = u'_1 = -0.3698225.$$

We may also use the Taylor series method of second order or any Runge-Kutta method of second order to obtain $u(0.2)$ and $u'(0.2)$. We obtain the following results.

For $j = 1 : t_0 = 0, t_1 = 0.2, t_2 = 0.4$.

$$\begin{aligned} P : u_2^{(0)} &= u_1 + \frac{h}{2}(3u'_1 - u'_0) \\ &= 0.9615386 + 0.1(-3 \times 0.3698225 - 0) = 0.8505918 \end{aligned}$$

$$E : f(t_2, u_2^{(0)}) = -0.5788051.$$

$$\begin{aligned} C : u_2^{(1)} &= u_1 + \frac{h}{2}(u_2^{(0)'} + u'_1) \\ &= 0.9615385 + 0.1(-0.5788051 - 0.3698225) = 0.866675 \end{aligned}$$

$$E : f(t_2, u_2^{(1)}) = -0.6009015$$

$$\begin{aligned} C : u_2^{(1)} &= u_1 + \frac{h}{2}(u_2^{(1)'} + u'_1) \\ &= 0.9615385 + 0.1(-0.6009015 - 0.3698225) \\ &= 0.8644661 \approx u(0.4). \end{aligned}$$

For PM_pCM_c method, we have

$$\begin{aligned} p_{j+1} &= u_j + \frac{h}{2}(3u'_j - u'_{j-1}) \\ m_{j+1} &= p_{j+1} - \frac{5}{6}(p_j - C_j) \\ C_{j+1} &= u_j + \frac{h}{2}(m'_{j+1} + u'_j) \\ u_{j+1} &= C_{j+1} + \frac{1}{2}(p_{j+1} - C_{j+1}), j = 1, 2, \dots \end{aligned}$$

To start the method, we need the values of $u(t)$ and $u'(t)$ at $t = 0.2$. The exact values are

$$t_1 = 0.2, u_1 = 0.09615385, u'_1 = -0.3698225.$$

For $j = 1$, we get

$$\begin{aligned} p_2 &= u_1 + \frac{h}{2}(3u'_1 - u'_0) \\ &= 0.9615385 + 0.1(-3 \times 0.3698225 - 0) = 0.8505918. \\ m_2 &= p_2 - \frac{5}{6}(p_1 - C_1). \end{aligned}$$

Taking $p_1 - C_1 = 0$, we obtain

$$\begin{aligned} m_2 &= 0.8505918 \\ m'_2 &= -2t_2 m_2^2 = -2(0.4)(0.8505918)^2 = -0.5788051 \\ C_2 &= u_1 + \frac{h}{2}(m'_2 + u'_1) \\ &= 0.9615385 + 0.1(-0.5788051 - 0.3698225) = 0.8666757 \\ p_2 - C_2 &= 0.8505918 - 0.8666757 = -0.0160839 \\ u(0.4) &= u_2 = C_2 + \frac{1}{6}(p_2 - C_2) \\ &= 0.8666757 + \frac{1}{6}(-0.0160839) = 0.8639951. \end{aligned}$$

The exact solution is $u(0.4) = 0.86207$.

3 Milne's Method

The open-type quadrature formula is

$$\int_{x_0}^{x_4} y(x) dx = \frac{4h}{3}(2y_1 - y_2 + 2y_3) + \frac{14}{45}h^5 y^{iv}(\xi).$$

From the above we can write,

$$\int_{x_{n-3}}^{x_{n+1}} y(x) dx = \frac{4h}{3}(2y_{n-2} - y_{n-1} + 2y_n) + \frac{14}{45}h^5 y^{iv}(\xi). \quad (3.1)$$

Applying (3.1) on $y' = \frac{dy}{dx}$, we get

$$y_{n+1} = y^{n-3} + \frac{4h}{3}(2y'_{n-2} - y'_{n-1} + 2y'_n) + \frac{14}{45}h^5 y^v(\xi).$$

When the above formula is used in context of the differential equation $y' = f(x, y)$

it gives,

$$y_{n+1} = y^{n-3} + \frac{4h}{3}(2f_{n-2} - f_{n-1} + 2f_n) + \frac{14}{45}h^5y^v(\xi). \quad (3.2)$$

Similarly, from the Simpson's quadrature formula(closed type), we get,

$$y_{n+1} = y^{n-1} + \frac{h}{3}(f_{n-1} + 4f_n + f_{n+1}) - \frac{h^5}{90}y^v(\xi). \quad (3.3)$$

It may be noted that the error in (3.2) is 28 times that of formula (3.3). Therefore, using (3.2) as predictor and (3.3) as corrector formula, the Milne's method may be written as,

$$y_{n+1}^p = y^{n-3} + \frac{4h}{3}(2f_{n-2} - f_{n-1} + 2f_n). \quad (3.4)$$

$$y_{n+1}^c = y^{n-1} + \frac{h}{3}(f_{n-1} + 4f_n + f_{n+1}). \quad (3.5)$$

The value of f_{n+1} is computed using (6), i.e.,

$$f_{n+1} = f(x_{n+1}, y_{n+1}^p). \quad (3.6)$$

The P-C method given by (3.4), (3.5) is also known as Milne-Simpson method.

It may be noted from (3.4) that in order to compute y_{n+1}^p , the values of y at four previous points, viz., y_{n-3} , y_{n-2} , y_{n-1} and y_n are ought to be known while in all previous methods the value of y was required at one previous point, $x = x_n$ only. Thus only two values were involved in the earlier methods, one at the current level and the other at the previous level. The methods which require information at more than one previous points in order to compute the value at the current level are known as 'multi-step' methods. Milne-Simpson's is a multi-step method.

On account of the reason stated above, the Milne's method is not 'self-starting'. That is, the first three values y_1 , y_2 and y_3 can not be computed by this method.

These values have to be computed by any one of the methods discussed earlier. After that this method may be applied for $n = 3$ onwards.

We now discuss the stability of the corrector formula (3.5). As before, let y_n denote the computed value and $y(x_n)$, the exact value of y at $x = x_n$, and let ϵ_n be the error in y_n such that $y(x_n) = y_n + \epsilon_n$. Then,

$$\begin{aligned} f\{x_n, y(x_n)\} &= f\{x_n, y_n + \epsilon_n\} \\ &= f(x_n, y_n) + \epsilon_n f_y(x_n, y_n) + \text{higher powers of } \epsilon_n^2 \end{aligned}$$

or
$$f\{x_n, y(x_n)\} - f(x_n, y_n) = \epsilon_n \theta_n, \text{ where } \theta_n = f_y(x_n, y_n)$$

Similar relations can be written for x_{n-1} and x_{n+1} , giving

$$f\{x_{n-1}, y(x_{n-1})\} - f(x_{n-1}, y_{n-1}) = \epsilon_{n-1} \theta_{n-1}, \text{ where } \theta_{n-1} = f_y(x_{n-1}, y_{n-1})$$

and
$$f\{x_{n+1}, y(x_{n+1})\} - f(x_{n+1}, y_{n+1}) = \epsilon_{n+1} \theta_{n+1}, \text{ where } \theta_{n+1} = f_y(x_{n+1}, y_{n+1})$$

From the corrector formula (3.5), the approximate value of y at $x = x_{n+1} = x_n + h$ is given by,

$$y_{n+1} = y_{n-1} + \frac{h}{3}(f_{n-1} + 4f_n + f_{n+1}),$$

while the exact value would be,

$$y(x_{n+1}) = y(x_{n-1}) + \frac{h}{3}[f(x_{n-1}, y_{n-1} + \epsilon_{n-1}) + 4f(x_n, y_n + \epsilon_n) + f(x_{n+1}, y_{n+1} + \epsilon_{n+1})].$$

Subtracting the approximate value from the exact and using above relations we get,

$$\epsilon_{n+1} = \epsilon_{n-1} + \frac{h}{3}(\theta_{n-1}\epsilon_{n-1} + 4\theta_n\epsilon_n + \theta_{n+1}\epsilon_{n+1})$$

or
$$(1 - \frac{h}{3}\theta_{n+1})\epsilon_{n+1} - \frac{4h}{3}\theta_n\epsilon_n - (1 + \frac{h}{3}\theta_{n-1})\epsilon_{n-1} = 0.$$

Assuming h to be small such that value of $\theta = f_y$ does not vary much in $x_{n-1} \leq x \leq x_{n+1}$, the above can be written as,

$$(1 - \frac{h}{3}\theta)\epsilon_{n+1} - \frac{4h}{3}\theta\epsilon_n - (1 + \frac{h}{3}\theta)\epsilon_{n-1} = 0$$

In order to solve the above difference equation, let us assume the solution as $\epsilon_n = \alpha^n$. Substituting this value, we get the auxiliary/characteristic equation,

$$\left(1 - \frac{h\theta}{3}\right) \alpha^2 - \frac{4h\theta}{3} \alpha - \left(1 + \frac{h\theta}{3}\right) = 0.$$

If α_1 and α_2 are the roots of this equation, then

$$\alpha_1 + \alpha_2 = \frac{4h\theta}{3-h\theta}; \quad \alpha_1 \cdot \alpha_2 = -\frac{3+h\theta}{3-h\theta},$$

and the solution will be given by,

$$\epsilon_n = c_1(\alpha_1)^n + c_2(\alpha_2)^n,$$

where c_1 and c_2 are arbitrary constants.

For $h\theta$ small, the product of the roots $\alpha_1\alpha_2$ is nearly equal to unity in absolute value which means the roots α_1 and α_2 are almost reciprocal to each other numerically. Thus the solution can be expressed as,

$$\epsilon_n \approx c_1(\alpha_1)^n + c_2\left(\frac{1}{\alpha_1}\right)^n, \text{ since } \alpha_2 = -\frac{1}{\alpha_1}.$$

The above solution has two components which behave reciprocally. That is, if one component decreases with increasing n , the other increases. Thus, in any case, the error grows exponentially with number of steps. Hence, the formula is unstable. That means, the formula should not be used for large number of steps and nor for iteration in the scheme P-C-C-C.

It may also be stated that such a situation is very likely to occur whenever a lower order differential equation is approximated by a higher order difference formula.

UNIT - V

1 Introduction

Consider the two point boundary value problem

$$u'' = f(x, u, u'), x \in (a, b) \quad (1.1)$$

where a prime denotes differentiation with respect to x , with one of the following three boundary conditions.

Boundary conditions of the first kind:

$$u(a) = \gamma_1, u(b) = \gamma_2. \quad (1.2)$$

Boundary conditions of the second kind:

$$u'(a) = \gamma_1, u'(b) = \gamma_2. \quad (1.3)$$

Boundary conditions of the third kind (or mixed kind):

$$a_0u(a) - a_1u'(a) = \gamma_1b_0u(b) + b_1u'(b) = \gamma_2 \quad (1.4)$$

where $a_0, b_0, a_1, b_1, \gamma_1$ and γ_2 are constants such that

$$\begin{aligned} a_0a_1 &\geq 0, |a_0| + |a_1| \neq 0 \\ b_0b_1 &\geq 0, |b_0| + |b_1| \neq 0 \text{ and } |a_0| + |b_0| \neq 0 \end{aligned}$$

A homogeneous boundary value problem possesses only a trivial solution $y(x) = 0$. We, therefore, consider those boundary value problems in which a parameter λ occurs either in the differential equation or in the boundary conditions, and we determine values of λ , called **eigenvalues**, for which the boundary value

problem has a nontrivial solution. Such a solution is called an eigenfunction and the entire problem is called an eigenvalue or a characteristic value problem. In general, a boundary value problem does not always have a unique solution. However, we shall assume that the boundary value problem under consideration has a unique solution.

The solution of the boundary value problem 1.1 exists and is unique if the following conditions are satisfied:

Let $u' = z$ and $-\infty < u, z < \infty$

(i) $f(x, u, z)$ is continuous.

(ii) $\frac{\partial f}{\partial u}$ and $\frac{\partial f}{\partial z}$ exist and are continuous.

(iii) $\frac{\partial f}{\partial u} > 0$ and $|\frac{\partial f}{\partial z}| \leq w$. In what follows, we shall assume that the boundary value problem has a unique solution and we shall attempt to determine it. The numerical methods for solving the boundary value problems may broadly be classified into the following three types:

(i) **Shooting methods** These are initial value problem methods. Here, we add sufficient number of conditions at one end point and adjust these conditions until the required conditions are satisfied at the other end.

(ii) **Difference methods** The differential equation is replaced by a set of difference equations which are solved by direct or iterative methods.

(ii) **Finite element methods** The differential equation is discretized by using approximate methods with the piecewise polynomial solution.

We shall now discuss in detail the shooting methods, difference methods and finite element methods for solving numerically both the linear and nonlinear second order boundary value problems.

2 Linear Second Order Differential Equations

Consider the boundary value problem 1.1 (BVP) subject to the given boundary conditions.

Since the differential equation is of second order, we require two linearly independent conditions to solve the boundary value problem. One of the ways of solving the boundary value problem is the following.

Boundary conditions of the first kind:

Here, we are given $u(a) = \gamma_1$. In order that an initial value method can be used, we guess the value of the slope at $x = a$ as $u'(a) = s$.

Boundary conditions of the second kind:

Here, we are given $u'(a) = \gamma_1$. In order that an initial value method can be used, we guess the value of $u(x)$ at $x = a$ as $u(a) = s$.

Boundary conditions of the third kind:

Here, we guess the value $u(a)$ or $u'(a)$. If we assume that $u'(a) = s$, then from 1.4, we get

$$u(a) = (a_1 s + \gamma_1) / a_0.$$

The related initial value problem is solved upto $x = b$, by using a singlestep or a multistep method. If the problem is solved directly, then we use the methods for second order initial value problems. If the differential equation is reduced to a system of two first order equations, then we use the Runge-Kutta methods or the multistep methods for a system of first order equations.

If the solution at $x = b$ does not satisfy the given boundary condition at the other end $x = a$, then we take another guess value of $u(a)$ or $u'(a)$ and solve

the initial value problem again upto $x = b$. These two solutions at $x = b$, of the initial value problems are used to obtain a better estimate of $u(a)$ or $u'(a)$. A sequence of such problems are solved, if necessary, to obtain the solution of the given boundary value problem. *For, a linear, nonhomogeneous boundary value problem, it is sufficient to solve two initial value problems with to linearly independent guess initial conditions.*

This technique of solving the boundary value problems by using the methods for solving the initial value problems is called the **shooting method**.

2.1 Linear Second Order Differential Equations

Consider the numerical solution of the differential equation

$$-u'' + p(x)u' + q(x)u = r(x), a < x < b \quad (2.1)$$

subject to the boundary conditions. We assume that the functions $p(x), q(x) > 0$, and $r(x)$ are continuous on $[a, b]$, so that the boundary value problem 2.1 has a unique solution.

The general solution of 2.1 can be written as

$$u(x) = u_0(x) + \mu_1 u_1(x) + \mu_2 u_2(x) \quad (2.2)$$

where

(i) $u_0(x)$ is a particular solution of the nonhomogeneous equation 2.1, that is

$$-u_0'' + p(x)u_0' + q(x)u_0 = r(x) \quad (2.3)$$

(ii) $u_1(x)$ and $u_2(x)$ are two linearly independent, complementary solutions of the corresponding homogeneous equation of 2.1, that is

$$-u_1'' + p(x)u_1' + q(x)u_1 = 0 \quad (2.4)$$

$$-u_2'' + p(x)u_2' + q(x)u_2 = 0 \quad (2.5)$$

We choose the initial conditions as follows:

Boundary conditions of the first kind:

Since $u(a) = \gamma_1$ is given, we take a guess value for $u'(a)$. We have the following two cases.

Case 1 $\gamma_1 \neq 0$. We choose $u_0(a) = u_1(a) = u_2(a) = \gamma_1$

$$u_0'(a) = \eta_0^*(a), u_1'(a) = \eta_1^*, u_2'(a) = \eta_2^* \quad (2.6)$$

where $\eta_0^*, \eta_1^*, \eta_2^*$ are arbitrary. Since $u_1(x)$ and $u_2(x)$ are linearly independent solutions, a suitable choice of the initial conditions is

$$\eta_0^* = 0, \eta_1^* = 1, \eta_2^* = 0. \quad (2.7)$$

Other choices of linearly independent values can also be considered.

We now solve the differential equations (2.3)-(2.5) along with the corresponding initial conditions, using the initial value methods with the same step lengths, and obtain $u_0(b), u_1(b)$ and $u_2(b)$.

Now since the solution (2.2) satisfies the boundary conditions at $x = a$ and $x = b$, we obtain, at

$$x = a : u_0(a) + \mu_1 u_1(a) + \mu_2 u_2(a) = \gamma_1$$

or

$$\gamma_1 + \mu_1 \gamma_1 + \mu_2 \gamma_1 = \gamma_1$$

$$\text{or} \quad \mu_1 + \mu_2 = 0 \quad (2.8)$$

$$x = b : u_0(b) + \mu_1 u_1(b) + \mu_2 u_2(b) = \gamma_2$$

or

$$\mu_2 = \frac{\gamma_2 - u_0(b)}{u_2(b) - u_1(b)}, u_1(b) \neq u_2(b). \quad (2.9)$$

Case 2 $\gamma_1 = 0$. In this case, we cannot use the conditions (2.7), since $[u_1(a), u_1'(a)]^T = [0, 1]^T$ and $[u_2(a), u_2'(a)]^T = [0, 0]^T$ are linearly dependent.

We choose the conditions as

$$\begin{aligned} u_0(a) &= \eta_0, u_1(a) = \eta_1, u_2(a) = \eta_2, \\ u_0'(a) &= \eta_0^*, u_1'(a) = \eta_1^*, u_2'(a) = \eta_2^*. \end{aligned}$$

A suitable set of values is

$$\eta_0 = \gamma_1 = 0, \eta_0^* = 0; \eta_1^* = 0; \eta_2 = 0, \eta_2^* = 1. \quad (2.10)$$

We note that the conditions $[u_1(a), u_1'(a)]^T = [1, 0]$ and $[u_2(a), u_2'(a)]^T = [0, 1]^T$ are linearly independent. Any other linearly independent set of values can be used.

We now solve the corresponding initial value problems upto $x = b$.

Now, since the solution (2.2) satisfies the boundary conditions at $x = a$ and $x = b$, we obtain, at

$$x = a : u_0(a) + \mu_1 u_1(a) + \mu_2 u_2(a) = \gamma_1 = 0$$

or $\eta_0 + \mu_1 \eta_1 + \mu_2 \eta_2 = 0,$

or $\mu_1 = 0$ (using(2.10))

$$x = b : u_0(b) + \mu_1 u_1(b) + \mu_2 u_2(b) = \gamma_2 \quad (2.11)$$

or $\mu_2 = \frac{\gamma_2 - u_0(b)}{u_2(b)}, u_2(b) \neq 0. \quad (2.12)$

We determine μ_1, μ_2 from (2.8) or (2.11) and obtain the solution of the given boundary value problem, using (2.2), at the mesh points used in integrating the initial value problems.

Boundary conditions of the second kind:

Since $u'(a) = \gamma_1$ is given, we guess a value for $u(a)$. Again, we consider the following two cases.

Case 1. $\gamma_1 \neq 0$. We choose $u_0(a) = \eta_0, u_1(a) = \eta_1, u_2(a) = \eta_2$,

$$u'_0(a) = \eta_0, u'_1(a) = \eta_1, u'_2(a) = \gamma_1 \quad (2.13)$$

A suitable set of values is

$$\eta_0 = 0, \eta_1 = 1, \eta_2 = 0 \quad (2.14)$$

Since the initial conditions $[u_1(a), u'_1(a)]^T = [1, \gamma_1]^T, [u_2(a), u'_2(a)]^T = [0, 1]^T$ are linearly independent, we obtain linearly independent solutions $u_1(x)$ and $u_2(x)$. Using these initial conditions, we solve the corresponding initial value problems, with the same step lengths, upto $x = b$.

Now, from (2.2), we get

$$u'(x) = u'_0(x) + \mu_1 u'_1(x) + \mu_2 u'_2(x) \quad (2.15)$$

Using the given conditions (1.3), we get, at

$$x = a : u'_0(a) + \mu_1 u'_1(a) + \mu_2 u'_2(a) = \gamma_1$$

$$\text{or } \gamma_1 + \mu_1 \gamma_1 + \mu_2 \gamma_1 = \gamma_1, \text{ or } \mu_1 + \mu_2 = 0. \quad (2.16)$$

$$x = b : u'_0(b) + \mu_1 u'_1(b) + \mu_2 u'_2(b) = \gamma_2$$

$$\text{or } \mu_2 = \frac{\gamma_2 - u'_0(b)}{u'_2(b)}, u'_1(b) \neq u'_2(b). \quad (2.17)$$

Case 2. $\gamma_1 = 0$. We cannot use the conditions as in case 1, since $[u_1(a), u'_1(a)]^T = [1, 0]^T$ and $[u_2(a), u'_2(a)]^T = [0, 0]^T$ are linearly dependent. In this case, we choose

$$u_0(a) = \eta_0, u_1(a) = \eta_1, u_2(a) = \eta_2$$

$$u'_0(a) = \eta_0^*, u'_1(a) = \eta_1^*, u'_2(a) = \eta_2^* .$$

A suitable set of values is

$$\eta_0 = 0, \eta_0^* = \gamma_1 = 0; \eta_1 = 1, \eta_1^* = 0; \eta_2 = 0, \eta_2^* = 1 \quad (2.18)$$

We note that the conditions $[u_1(a), u'_1(a)]^T = [1, 0]^T$ and $[u_2(a), u'_2(a)]^T = [0, 1]^T$ are linearly independent. Any other set of linearly independent values can be used.

Using (2.2), (2.15) and the boundary conditions (1.3), we get, at

$$x = a : \quad u'_0(a) + \mu_1 u'_1(a) + \mu_2 u'_2(a) = \gamma_1 = 0$$

$$\text{or} \quad \eta_0^* + \mu_1 \eta_1^* + \mu_2 \eta_2^* = 0, \text{ or } \mu_2 = 0. \quad (2.19)$$

$$x = b : \quad u'_0(b) + \mu_1 u'_1(b) + \mu_2 u'_2(b) = \gamma_2$$

$$\text{or} \quad \mu_1 = \frac{\gamma_2 - u'_0(b)}{u'_1(b)}, u'_1(b) \neq 0. \quad (2.20)$$

We determine μ_1, μ_2 from (2.16) or (2.20) and obtain the solution of the boundary value problem, using (2.2), at the mesh points used in integrating the initial value problems.

Boundary conditions of the third kind:

In this case, we assume the arbitrary initial conditions $u_0(a) = \eta_0, u_1(a) = \eta_1, u_2(a) = \eta_2$

$$u'_0(a) = \eta_0^*, u'_1(a) = \eta_1^*, u'_2(a) = \eta_2^*. \quad (2.21)$$

A suitable set of values is

$$\eta_0 = 0, \eta_0^* = 0; \eta_1 = 1, \eta_1^* = 0; \eta_2 = 0, \eta_2^* = 1 \quad (2.22)$$

Again, we note that the conditions $[u_1(a), u'_1(a)]^T = [1, 0]^T$ and $[u_2(a), u'_2(a)]^T = [0, 1]^T$ are linearly independent. Using these initial conditions, we solve the corresponding initial value problems, using the same step lengths, upto $x = b$.

Using (2.2),(2.18) and the boundary conditions (1.3), we get, at

$$x = a : a_0[u_0(a) + \mu_1 u_1(a) + \mu_2 u_2(a)] - a_1[u'_0(a) + \mu_1 u'_1(a) + \mu_2 u'_2(a)] = \gamma_1$$

$$\text{or } a_0[u_0(a) + \mu_1 u_1(a) + \mu_2 u_2(a)] - a_1[u'_0(a) + \mu_1 u'_1(a) + \mu_2 u'_2(a)] = \gamma_1$$

$$\text{or } a_0[\eta_0 + \mu_1 \eta_1 + \mu_2 \eta_2] - a_1[\eta_0^* + \mu_1 \eta_1^* + \mu_2 \eta_2^*] = \gamma_1$$

$$\text{or } a_0 \mu_1 - a_1 \mu_2 = \gamma_1. \quad (2.23)$$

$$x = b : b_0[u_0(b) + \mu_1 u_1(b) + \mu_2 u_2(b)] - a_1[\eta_0^* + \mu_1 \eta_1^* + \mu_2 \eta_2^*] = \gamma_1$$

$$\text{or } \mu_1 [b_0 u_1(b) + b_1 u'_1(b)] + \mu_2 [b_0 u_2(b) + b_1 u'_2(b)] = \gamma_2 - [b_0 u_0(b) + b_1 u'_0(b)]. \quad (2.24)$$

We determine μ_1, μ_2 from (2.23) and obtain the solution of the boundary value problem, using (2.2), at the mesh points used in integrating the initial value problems.

The shooting method requires the solution of the three initial value problems 2.3, 2.5 and 2.8. Denoting $\phi_i(x) = w^{(i+1)}(x)$ and $\phi'_i(x) = v^{(i+1)}(x), i = 0, 1, 2$, the IVPs 2.3-2.8 can be written as the following equivalent first order systems.

2.2 Alternate Method

When the boundary value problem is nonhomogeneous, then it is sufficient to solve the two initial value problems

$$-\phi_1'' + p(x)\phi_1' + q(x)\phi_1 = r(x) \quad (2.25)$$

$$-\phi_2'' + p(x)\phi_2' + q(x)\phi_2 = r(x) \quad (2.26)$$

with suitable initial conditions at $x = a$. We write the general solution of the boundary value problem in the form

$$y(x) = \lambda \phi_1(x) + (1 - \lambda) \phi_2(x) \quad (2.27)$$

and determine λ so that the boundary condition at the other end, that is, at $x = b$ is satisfied. We solve the initial value problems (2.25),(2.26) upto $x = b$ using the initial conditions.

(i) *Boundary conditions of the first kind:*

$$\phi_1(a) = \gamma_1, \phi_1'(a) = 0$$

$$\phi_2(a) = \gamma_1, \phi_2'(a) = 1$$

From 2.27, we obtain

$$y(b) = \gamma_2 = \lambda\phi_1(b) + (1 - \lambda)\phi_2(b),$$

which gives

$$\lambda = \frac{\gamma_2 - \phi_2(b)}{\phi_1(b) - \phi_2(b)}, \phi_1(b) \neq \phi_2(b).$$

(ii) *Boundary conditions of the second kind:*

$$\phi_1(a) = 0, \phi_1'(a) = \gamma_1$$

$$\phi_2(a) = 1, \phi_2'(a) = \gamma_1$$

From 2.27, we obtain

$$y'(b) = \gamma_2 = \lambda\phi_1'(b) + (1 - \lambda)\phi_2'(b),$$

which gives

$$\lambda = \frac{\gamma_2 - \phi_2'(b)}{\phi_1'(b) - \phi_2'(b)}, \phi_1'(b) \neq \phi_2'(b).$$

(iii) *Boundary conditions of the third kind:*

$$\phi_1(a) = 0, \phi_1'(a) = -\frac{\gamma_1}{a_1}$$

$$\phi_2(a) = 1, \phi_2'(a) = -\frac{(a_0 - \gamma_1)}{a_1}$$

From 2.27, we obtain

$$y'(b) = \gamma_2 = \lambda\phi_1'(b) + (1 - \lambda)\phi_2'(b),$$

which gives

$$y(b) = \lambda\phi_1(b) + (1 - \lambda)\phi_2(b),$$

$$y'(b) = \gamma_2 = \lambda\phi_1'(b) + (1 - \lambda)\phi_2'(b),$$

Substituting in the second condition, $b_0y(b) + b_1y'(b) = \gamma_2$, in 1.4, we get

$\gamma_2 = b_0[\lambda\phi_1(b) + (1 - \lambda)\phi_2(b)] + b_1[\lambda\phi_1'(b) + (1 - \lambda)\phi_2'(b)]$ which gives

$$\lambda = \frac{\gamma_2 - [b_0\phi_2(b) + b_1\phi_2'(b)]}{[b_0\phi_1(b) + b_1\phi_1'(b)] - [b_0\phi_2(b) + b_1\phi_2'(b)]} \quad (2.28)$$

The results obtained are identical in both the approaches.

Example 2.1. Using the shooting method, solve the first boundary value problem

$$\begin{aligned} u'' &= u + 1, \quad 0 < x < 1 \\ u(0) &= 0, \quad u(1) = e - 1. \end{aligned}$$

Use the Euler-Cauchy method with $h = 0.25$ to solve the resulting system of first order initial value problems. Compare the solution with the exact solution $u(x) = e^x - 1$.

Since the boundary value problem is linear and nonhomogeneous, we assume the solution in the form

$$u(x) = u_0(x) + \mu_1 u_1(x) + \mu_2 u_2(x) \quad (2.29)$$

where $u_0(x)$ satisfies the nonhomogeneous differential equation and $u_1(x)$, $u_2(x)$ satisfy the homogeneous differential equation. Therefore, we have

$$u_0'' - u_0(x) = 1, \quad u_1''(x) - u_1(x) = 0 \quad \text{and} \quad u_2''(x) - u_2(x) = 0$$

We assume the initial conditions as given in 2.9, that is

$$\begin{aligned} \eta_0 &= \gamma_1 = 0, \quad \eta_0^* = 0; \quad \eta_1 = 1, \quad \eta_1^* = 0; \quad \eta_2 = 0, \quad \eta_2^* = 1, \quad \text{that is} \\ u_0(0) &= 0, \quad u_0'(0) = 0; \quad u_1(0) = 1, \quad u_1'(0) = 0; \quad u_2(0) = 0, \quad u_2'(0) = 1. \end{aligned}$$

For the sake of illustration, we shall follow the steps in the method and obtain the analytical solution also.

Solving the differential equations and using the initial conditions, we obtain

$$\begin{aligned}
u_0(x) &= \left(\frac{1}{2}\right)(e^x + e^{-x}) - 1, \quad u_1(x) = \left(\frac{1}{2}\right)(e^x + e^{-x}), \\
u_2(x) &= \left(\frac{1}{2}\right)(e^x - e^{-x})
\end{aligned} \tag{2.30}$$

Now, using 2.29, we get

$$u(0) = u_0(0) + \mu_1 u_1(0) + \mu_2 u_2(0) = \mu_1 = 0$$

and
$$u(1) = u_0(1) + \mu_1 u_1(1) + \mu_2 u_2(1)$$

$$u(1) = u_0(1) + \mu_2 u_2(1) = e - 1. \tag{2.31}$$

Now, from 2.30, we obtain

$$u_0(1) = \left(\frac{1}{2}\right)(e + e^{-1}) - 1, \text{ and } u_2(1) = \left(\frac{1}{2}\right)(e - e^{-1}).$$

Hence, from 2.31, we get

$$\begin{aligned}
\mu_2 &= \frac{(e-1)-u_0(1)}{u_2(1)} = \frac{2(e-1)-(e+e^{-1}-2)}{(e-e^{-1})} \\
&= \frac{e-e^{-1}}{e-e^{-1}} = 1
\end{aligned}$$

Therefore, the analytical solution of the problem is

$$\begin{aligned}
u(x) &= u_0(x) + \mu_1 u_1(x) + \mu_2 u_2(x) \\
&= \left(\frac{1}{2}\right)(e^x + e^{-x}) - 1 + \left(\frac{1}{2}\right)(e^x - e^{-x}) = e^x - 1.
\end{aligned}$$

This illustrates the general procedure of implementation of the method.

We now determine the solution of the initial value problem, using the Euler-Cauchy method with $h = 0.25$.

We need to solve the following three, second order initial value problems in $0 < x < 1$

$$\begin{aligned}
u_0'' - u_0(x) &= 1, \quad u_0(0) = 0, \quad u_0'(0) = 0. \\
u_1'' - u_1(x) &= 0, \quad u_1(0) = 1, \quad u_1'(0) = 0. \\
u_2'' - u_2(x) &= 0, \quad u_2(0) = 0, \quad u_2'(0) = 1
\end{aligned} \tag{2.32}$$

We write these problems as equivalent first order systems.

$$\begin{aligned} \text{Denote } u_0(x) &= Y_0(x), \quad u'_0(x) = Y'_0(x) = Z_0(x), \\ u_1(x) &= Y_1(x), \quad u'_1(x) = Y'_1(x) = Z_1(x), \\ u_2(x) &= Y_2(x), \quad u'_1(x) = Y'_2(x) = Z_2(x). \end{aligned}$$

Then, we can write 2.32 as the following systems

$$\begin{aligned} \begin{bmatrix} Y_0 \\ Z_0 \end{bmatrix}' &= \begin{bmatrix} Z_0 \\ 1 + Y_0 \end{bmatrix}, & \begin{bmatrix} Y_0(0) \\ Z_0(0) \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} Y_1 \\ Z_1 \end{bmatrix}' &= \begin{bmatrix} Z_1 \\ Y_1 \end{bmatrix}, & \begin{bmatrix} Y_1(0) \\ Z_1(0) \end{bmatrix} &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ \begin{bmatrix} Y_2 \\ Z_2 \end{bmatrix}' &= \begin{bmatrix} Z_2 \\ Y_2 \end{bmatrix}, & \begin{bmatrix} Y_2(0) \\ Z_2(0) \end{bmatrix} &= \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \end{aligned}$$

Applying the Euler-Cauchy method

$$\begin{aligned} \mathbf{u}_{j+1} &= \mathbf{u}_j + \frac{1}{2}(\mathbf{k}_1 + \mathbf{k}_2) \\ \mathbf{k}_1 &= h\mathbf{f}(t_j, \mathbf{u}_j), \quad \mathbf{k}_2 = h\mathbf{f}(t_j + h, \mathbf{u}_j + \mathbf{k}_1) \end{aligned}$$

We obtain the following systems:

System 1 We have $f_1 = Z_0$ and $f_2 = 1 + Y_0$

$$\begin{aligned}
\begin{bmatrix} Y_{0,j+1} \\ Z_{0,j+1} \end{bmatrix} &= \begin{bmatrix} Y_{0,j} \\ Z_{0,j} \end{bmatrix} + \frac{h}{2} \begin{bmatrix} Z_{0,j} \\ 1 + Y_{0,j} \end{bmatrix} + \frac{h}{2} \begin{bmatrix} Z_{0,j} + h(1 + Y_{0,j}) \\ 1 + Y_{0,j} + hZ_{0,j} \end{bmatrix} \\
&= \begin{bmatrix} 1 + (h^2/2) & h \\ h & 1 + (h^2/2) \end{bmatrix} \begin{bmatrix} Y_{0,j} \\ Z_{0,j} \end{bmatrix} + \begin{bmatrix} h^2/2 \\ h \end{bmatrix} \\
&= \mathbf{B}(h) \begin{bmatrix} Y_{0,j} \\ Z_{0,j} \end{bmatrix} + \begin{bmatrix} h^2/2 \\ h \end{bmatrix}
\end{aligned}$$

where $\mathbf{B}(h) = \begin{bmatrix} 1 + (h^2/2) & h \\ h & 1 + (h^2/2) \end{bmatrix}$

The initial conditions are $Y_{0,0} = 0$, $Z_{0,0} = 0$.

The *systems 2 and 3* can be immediately written as

$$\begin{bmatrix} Y_{1,j+1} \\ Z_{1,j+1} \end{bmatrix} = \mathbf{B}(h) \begin{bmatrix} Y_{1,j} \\ Z_{1,j} \end{bmatrix}, \quad Y_{1,0} = 1, \quad Z_{1,0} = 0$$

and

$$\begin{bmatrix} Y_{2,j+1} \\ Z_{2,j+1} \end{bmatrix} = \mathbf{B}(h) \begin{bmatrix} Y_{2,j} \\ Z_{2,j} \end{bmatrix}, \quad Y_{2,0} = 0, \quad Z_{2,0} = 1$$

where $\mathbf{B}(h)$ is same as above.

Using $h = 0.25$. We obtain

$$\begin{bmatrix} Y_{0,j+1} \\ Z_{0,j+1} \end{bmatrix} = \begin{bmatrix} 1.03125 & 0.25 \\ 0.25 & 1.03125 \end{bmatrix} \begin{bmatrix} Y_{0,j} \\ Z_{0,j} \end{bmatrix} + \begin{bmatrix} 0.03125 \\ 0.25 \end{bmatrix}$$

with $Y_{0,0} = 0, Z_{0,0} = 0$. For $j = 0, 1, 2, 3$, we get

$$u_0(0.25) \approx Y_{0,1} = 0.03125, \quad u'_0(0.25) \approx Z_{0,1} = 0.25.$$

$$u_0(0.50) \approx Y_{0,2} = 0.12598, \quad u'_0(0.50) \approx Z_{0,2} = 0.51563.$$

$$u_0(0.75) \approx Y_{0,3} = 0.29007, \quad u'_0(0.75) \approx Z_{0,3} = 0.81324.$$

$$u_0(1.00) \approx Y_{0,4} = 0.53369, \quad u'_0(1.00) \approx Z_{0,4} = 1.16117.$$

We have

$$\begin{bmatrix} Y_{1,j+1} \\ Z_{1,j+1} \end{bmatrix} = \begin{bmatrix} 1.03125 & 0.25 \\ 0.25 & 1.03125 \end{bmatrix} \begin{bmatrix} Y_{1,j} \\ Z_{1,j} \end{bmatrix}, \quad Y_{1,0} = 1, \quad Z_{1,0} = 0$$

$$u_1(0.25) \approx Y_{1,1} = 1.03125, \quad u'_1(0.25) \approx Z_{1,1} = 0.25.$$

$$u_1(0.50) \approx Y_{1,2} = 1.12598, \quad u'_1(0.50) \approx Z_{1,2} = 0.51563.$$

$$u_1(0.75) \approx Y_{1,3} = 1.29007, \quad u'_1(0.75) \approx Z_{1,3} = 0.81324.$$

$$u_1(1.00) \approx Y_{1,4} = 1.53369, \quad u'_1(1.00) \approx Z_{1,4} = 1.16117.$$

$$\begin{bmatrix} Y_{2,j+1} \\ Z_{2,j+1} \end{bmatrix} = \begin{bmatrix} 1.03125 & 0.25 \\ 0.25 & 1.03125 \end{bmatrix} \begin{bmatrix} Y_{2,j} \\ Z_{2,j} \end{bmatrix}, \quad Y_{2,0} = 0, \quad Z_{2,0} = 1$$

$$u_2(0.25) \approx Y_{2,1} = 0.25, \quad u_2'(0.25) \approx Z_{2,1} = 1.03125.$$

$$u_2(0.50) \approx Y_{2,2} = 0.51563, \quad u_2'(0.50) \approx Z_{2,2} = 1.12598.$$

$$u_2(0.75) \approx Y_{2,3} = 0.81324, \quad u_2'(0.75) \approx Z_{2,3} = 1.29007.$$

$$u_2(1.00) \approx Y_{2,4} = 1.16117, \quad u_2'(1.00) \approx Z_{2,4} = 1.53369.$$

From 2.11 and 2.12, we get

$$\mu_1 = 0, \quad \mu_2 = \frac{\gamma_2 - u_0(1)}{u_2(1)} = \frac{e^{-1} - 0.53369}{1.16117} = 1.02017.$$

We obtain the solution of the boundary value problem from

$$u(x) = u_0(x) + 1.02017u_2(x).$$

The solutions at the nodal points are given in Table 5.1. The maximum absolute error which occurs at $x = 0.50$ is given by

$$\text{max.abs.error} = 0.00329$$

Table 5.1 Solution of Example 2.1

x_j	<i>Exact</i> : $u(x_j)$	u_j
0.25	0.28403	0.28629
0.50	0.64872	0.65201
0.75	1.11700	1.11971
1.00	1.71828	1.71828

More accurate results can be obtained by using smaller step length h .

Alternate Method

To apply the alternate method, we solve the two initial value problems

$$u_1'' = u_1 + 1, \quad u_1(0) = 0, \quad u_1'(0) = 0$$

and

$$u_2'' = u_2 + 1, \quad u_2(0) = 0, \quad u_2'(0) = 1.$$

We can also take the initial condition $u_1'(0)$ as $u_1'(0) = \alpha$, $\alpha \neq 0, 1$. Therefore, we obtain the equations (see equations of *systems* 1)

$$\begin{bmatrix} Y_{i,j+1} \\ Z_{i,j+1} \end{bmatrix} = \begin{bmatrix} 1.03125 & 0.25 \\ 0.25 & 1.03125 \end{bmatrix} \begin{bmatrix} Y_{i,j} \\ Z_{i,j} \end{bmatrix} + \begin{bmatrix} 0.03125 \\ 0.25 \end{bmatrix}$$

where $Y_1 = u_1$ and $Z_2 = u_2$.

Using the conditions $Y_{1,0} = 0$, $Z_{1,0} = 0$, we obtain

$$\begin{bmatrix} Y_{1,1} \\ Z_{1,1} \end{bmatrix} = \begin{bmatrix} 0.03125 \\ 0.25 \end{bmatrix}, \quad \begin{bmatrix} Y_{1,2} \\ Z_{1,2} \end{bmatrix} = \begin{bmatrix} 0.12598 \\ 0.51563 \end{bmatrix},$$

$$\begin{bmatrix} Y_{1,3} \\ Z_{1,3} \end{bmatrix} = \begin{bmatrix} 0.29007 \\ 0.81324 \end{bmatrix}, \quad \begin{bmatrix} Y_{1,4} \\ Z_{1,4} \end{bmatrix} = \begin{bmatrix} 0.53369 \\ 1.16117 \end{bmatrix},$$

Using the conditions $Y_{2,0} = 0$, $Z_{2,0} = 1$, we obtain

$$\begin{bmatrix} Y_{2,1} \\ Z_{2,1} \end{bmatrix} = \begin{bmatrix} 0.28125 \\ 1.28125 \end{bmatrix}, \quad \begin{bmatrix} Y_{2,2} \\ Z_{2,2} \end{bmatrix} = \begin{bmatrix} 0.64160 \\ 1.64160 \end{bmatrix},$$

$$\begin{bmatrix} Y_{2,3} \\ Z_{2,3} \end{bmatrix} = \begin{bmatrix} 1.10330 \\ 2.10330 \end{bmatrix}, \quad \begin{bmatrix} Y_{2,4} \\ Z_{2,4} \end{bmatrix} = \begin{bmatrix} 1.69485 \\ 2.69485 \end{bmatrix},$$

From 2.27, we get

$$\lambda = \frac{(e-1)-Y_{2,4}}{Y_{1,4}-Y_{2,4}} = \frac{e-1-1.69485}{0.53369-1.69485} = -0.02019$$

Hence, $u(x) = -0.02019 Y_1(x) + 1.02019 Y_2(x)$.

Substituting $x = 0.25, 0.5, 0.75$ and 1.0 , we get

$$u(0.25) \approx 0.28630, \quad u(0.50) \approx 0.65201, \quad u(0.75) \approx 1.11972, \quad u(1.0) \approx 1.71829.$$

These values are same as given in Table 5.1, except for the round off error in the last digit.

Example 2.2. Use the shooting method to solve the mixed boundary value problem

$$u'' = u - 4xe^x, 0 < x < 1,$$

$$u(0) - u'(0) = -1, u(1) + u'(1) = -e.$$

Use the Taylor series method

$$u_{j+1} = u_j + hu'_j + \frac{h^2}{2}u''_j + \frac{h^3}{6}u'''_j$$

$$u'_{j+1} = u'_j + hu''_j + \frac{h^2}{2}u'''_j$$

to solve the initial value problems. Assume $h = 0.25$. Compute with the exact solution $u(x) = x(1-x)e^x$.

We assume the solution in the form

$$u(x) = u_0(x) + \mu_1 u_1(x) + \mu_2 u_2(x)$$

where $u_0(x), u_1(x)$ and $u_2(x)$ satisfy the differential equations

$$u''_0 - u_0 = -4xe^x, u''_1 - u_1 = 0,$$

$$u''_2 - u_2 = 0.$$

The initial conditions may be assumed as given in 2.9;

$$u_0(0) = 0, u'_0(0) = 0,$$

$$u_1(0) = 1, u'_1(0) = 0,$$

$$u_2(0) = 0, u'_2(0) = 1.$$

To illustrate the solution procedure, we solve analytically the initial value problems. The analytical solutions of the above initial value problems are given by

$$u_0(x) = (1/2)e^{-x} - e^x(x^2 - x + (1/2))$$

$$u_1(x) = (1/2)(e^x + e^{-x}), u_2(x) = (1/2)(e^2 - e^{-x}).$$

We also have

$$u(0) = u_0(0) + \mu_1 u_1(0) + \mu_2 u_2(0) = \mu_1.$$

$$u'(0) = u'_0(0) + \mu_1 u'_1(0) + \mu_2 u'_2(0) = \mu_2.$$

$$\begin{aligned}
u_0(1) &= -(1/2)(e - e^{-1}), u_1(1) = (1/2)(e + e^{-1}), u_2(1) = (1/2)(e - e^{-1}). \\
u'_0(1) &= -(1/2)(3e + e^{-1}), u'_1(1) = (1/2)(e - e^{-1}), u'_2(1) = (1/2)(e + e^{-1}), \\
u(1) &= u_0(1) + \mu u_1(1) + \mu_2 u_2(1) \\
&= (1/2)(e^{-1} - e) + (1/2)\mu_1(e - e^{-1}) + (1/2)\mu_2(e - e^{-1}). \\
u'(1) &= u'_0(1) + \mu_1 u'_1(1) + \mu_2 u'_2(1) \\
&= -(1/2)(3e - e^{-1}) + (1/2)\mu_1(e - e^{-1}) + (1/2)\mu_2(e + e^{-1}).
\end{aligned}$$

Substituting into the boundary conditions, we get the relations

$$\mu_2 - \mu_1 = 1, \mu_2 + \mu_1 = 1, \text{ or } \mu_1 = 0, \mu_2 = 1.$$

Thus, the initial conditions are given by $u(0) = 0, u'(0) = 1$.

We now solve the three, second order initial value problems

$$\begin{aligned}
u''(0) &= u_0 - 4xe^x, u_0(0) = 0, u'_0(0) = 0, \\
u''_1 &= u_1, u_1(0) = 1, u'_1(0) = 0, \\
u''_2 &= u_2, u_2(0) = 0, u'_2(0) = 1.
\end{aligned}$$

by using the given Taylor series method with $h = 0.25$. We have the following results.

(i) $i = 0, u_{0,0} = 0, u'_{0,0} = 0$.

$$u''_{0,j} = u_{0,j} - 4x_j e^{x_j}, u'''_{0,j} = u'_{0,j} - 4(x_j + 1)e^{x_j}, j = 1, 2, 3.$$

Hence,

$$\begin{aligned}
u_{0,j+1} &= u_{0,j} + hu'_{0,j} + \frac{h^2}{2}(u_{0,j} - 4x_j e^{x_j}) + \frac{h^3}{6}[u'_{0,j} - 4(x_j + 1)e^{x_j}] \\
&= (1 + \frac{h^2}{2})u_{0,j} + (h + \frac{h^3}{6})u'_{0,j} - [\frac{2}{3}h^3(1 + x_j) + 2h^2x_j]e^{x_j} \\
&= 1.03125u_{0,j} + 0.25260u'_{0,j} + (0.13542x_j + 0.01042)e^{x_j} \\
u'_{0,j+1} &= u'_{0,j} + h[u_{0,j} - 4x_j e^{x_j}] + \frac{h^2}{2}[u'_{0,j} - 4(x_j + 1)e^{x_j}] \\
&= hu_{0,j} + (1 + \frac{h^2}{2})u'_{0,j} - 2[2hx_j + h^2(1 + x_j)]e^{x_j} \\
&= 0.25u_{0,j} + 1.03125u'_{0,j} - 2(0.5625x_j + 0.0625)e^{x_j}
\end{aligned}$$

Hence,

$$u_0(0.25) \approx u_{0,1} = -0.01042, u'(0.25) \approx u'_{0,1} = -0.12500,$$

$$\begin{aligned}
u_0(0.50) &\approx u_{0,2} = -0.09917, u'(0.50) \approx u'_{0,2} = -0.65315, \\
u_0(0.75) &\approx u_{0,3} = -0.39606, u'(0.75) \approx u'_{0,3} = -1.83185, \\
u_0(1.00) &\approx u_{0,4} = -1.10823, u'(1.00) \approx u'_{0,4} = -4.03895.
\end{aligned}$$

(ii) $i = 1, u_{1,0} = 1, u'_{1,j}, j = 1, 2, 3.$

$$\begin{aligned}
u_{1,j+1} &= u_{1,j} + hu'_{1,j} + \frac{h^2}{2}u_{1,j} + \frac{h^2}{6}u'_{1,j}. \\
&= (1 + \frac{h^2}{2})u_{1,j} + (h + \frac{h^2}{6})u'_{1,j}. \\
&= 1.03125u_{1,j} + 0.25260u'_{1,j}. \\
u'_{1,j+1} &= u'_{1,j} + hu_{1,j} + \frac{h^2}{2}u'_{1,j} \\
&= hu_{1,j} + (1 + \frac{h^2}{2})u'_{1,j} = 0.25u_{1,j} + 1.03125u'_{1,j}.
\end{aligned}$$

Hence

$$\begin{aligned}
u_1(0.25) &\approx u_{1,1} = 1.03125, u'_1(0.25) \approx u'_{1,1} = 0.25, \\
u_1(0.50) &\approx u_{1,2} = 1.12663, u'_1(0.50) \approx u'_{1,2} = 0.51563, \\
u_1(0.75) &\approx u_{1,3} = 1.29209, u'_1(0.75) \approx u'_{1,3} = 0.81340, \\
u_1(1.00) &\approx u_{1,4} = 1.53794, u'_1(1.00) \approx u'_{1,4} = 1.16184.
\end{aligned}$$

(iii) $i = 2, u_{2,0} = 0, u'_{2,0} = 1.$

$$u''_{2,j} = u_{2,j}, u'''_{2,j} = u'_{2,j}, j = 1, 2, 3.$$

Since the differential equation is same as for u_1 , we get

$$\begin{aligned}
u_{2,j+1} &= 1.03125u_{2,j} + 0.25260u'_{2,j} \\
u'_{2,j+1} &= 0.25u_{2,j} + 1.03125u'_{2,j}
\end{aligned}$$

Hence,

$$\begin{aligned}
u_2(0.25) &\approx u_{2,1} = 0.25260, u'_2(0.25) \approx u'_{2,1} = 1.03125, \\
u_2(0.50) &\approx u_{2,2} = 0.52099, u'_2(0.50) \approx u'_{2,2} = 1.12663, \\
u_2(0.75) &\approx u_{2,3} = 0.82186, u'_2(0.75) \approx u'_{2,3} = 1.29208, \\
u_2(1.00) &\approx u_{2,4} = 1.17393, u'_2(1.00) \approx u'_{2,4} = 1.53792.
\end{aligned}$$

From (7.20) and the given boundary conditions, we have

$$a_0 = a_1 = 1, b_0 = b_1 = 1, \gamma_1 = -1, \gamma_2 = -e.$$

$$\mu_1 - \mu_2 = -1$$

$$[u_1(1) + u_1'(1)]\mu_1 + [u_2(1) + u_2'(1)]\mu_2 = -e - [u_0(1) + u_0'(1)]$$

or

$$2.69978\mu_1 + 0.271185\mu_2 = 2.42890.$$

Solving these equations, we obtain $\mu_1 = -0.05229, \mu_2 = 0.94711$. We obtain the solution of the boundary value problem from

$$u(x) = u_0(x) - 0.05229u_1(x) + 0.94771u_2(x).$$

The solution at the nodal points are given in Table 5.2. The maximum absolute error which occurs at $x = 0.75$, is given by

$$\text{max. abs. error} = 0.08168.$$

Table 5.2 Divided Difference (d.d) Table

x_j	Exact : $u(x_j)$	u_j
0.25	0.24075	0.17505
0.50	0.41218	0.33567
0.75	0.39694	0.31526
1.00	0.0	-0.07610

2.3 Nonlinear Second Order Differential Equations

We now consider the nonlinear differential equation

$$y'' = f(x, y, y'), a < x < b$$

subject to one of the boundary conditions (1.2) to (1.4). Since the differential equation is non linear, we cannot write the solution in the form $y(x) = \phi_0(x) + \mu_1\phi_1(x) + \mu_2\phi_2(x)$ or (2.27). Depending on the boundary conditions, we proceed as follows :

Boundary condition of the first kind :

We have the boundary conditions as $y(a) = \gamma_1$ and $y(b) = \gamma_2$. We assume $y'(a) = s$ and solve the initial value problem $y'' = f(x, y, y')$,

$$y(a) = \gamma_1, y'(a) = s \tag{2.33}$$

upto $x = b$ using any numerical method. The solution, $y(b, s)$ of the initial value problem should satisfy the boundary condition at $x = b$. Let

$$\phi(s) = y(b, s) - \gamma_2 \tag{2.34}$$

Hence, the problem is to find s , such that $f(s) = 0$.

Boundary conditions of the second kind:

We have the boundary conditions as $y'(a) = \gamma_1$ and $y'(b) = \gamma_2$. We assume $y(a) = s$ and solve the initial value problem

$$y'' = f(x, y, y') y(a) = s, y'(a) = \gamma_1 \tag{2.35}$$

upto $x = b$ using any numerical method. The solution $y(b, s)$ of the initial value problem should satisfy the boundary condition at $x = b$. Let

$$\phi(s) = y'(b, s) - \gamma_2. \tag{2.36}$$

Hence, the problem is to find s , such that $f(s) = 0$.

Boundary condition of the third kind:

We have the boundary conditions as $a_0y(a) - a_1y'(a) = \gamma_1$, and $b_0y(b) + b_1y'(b) = \gamma_2$. Here, we can assume the value of $y(a)$ or $y'(a)$. Let $y'(a) = s$. Then, from

$$a_0y(a) - a_1y'(a) = \gamma_1,$$

we get $y(a) = \frac{(a_1s + \gamma_1)}{a_0}$.

We now solve the initial value problem $y'' = f(x, y, y')$

$$y(a) = \frac{1}{a_0}(a_1s + \gamma_1), y'(a) = s \quad (2.37)$$

upto $x = b$ using any numerical method. The solution $y(b, s)$ of the initial value problem should satisfy the boundary condition at $x = b$. Let

$$\phi(s) = b_0y(b, s) + b_1y'(b, s) - \gamma_2. \quad (2.38)$$

Hence, the problem is to find s , such that $\phi(s) = 0$. The function $\phi(s)$ in 2.36 or 2.34 or 2.38 is a nonlinear function in s . We solve the equation

$$\phi(s) = 0 \quad (2.39)$$

by using any iterative method discussed in Unit III. $\lambda = \frac{\gamma_2 - \phi_2'(b)}{\phi_1'(b) - \phi_2'(b)}$, $\phi_1'(b) \neq \phi_2'(b)$.

2.4 Secant Method

The iteration method for solving $\phi(s) = 0$ is given by

$$s^{(k+1)} = s^{(k)} - \left[\frac{s^{(k)} - s^{(k-1)}}{\phi(s^{(k)}) - \phi(s^{(k-1)})} \right] \phi(s)^{(k)} \quad (2.40)$$

which $s^{(0)}$ and $s^{(1)}$ are two initial approximations to s . We solve the initial value problem 2.33 or 2.35 or 2.37 with two guess values of sand keep iterating till $|\phi(s^{(k+1)})| < (\text{given error tolerance})$.

2.5 Newton-Raphson Method

The iteration method for solving $\phi(s) = 0$ is given by

$$s^{(k+1)} = s^{(k)} - \frac{\phi(s^{(k)})}{\phi'(s^{(k)})}, k = 0, 1, 2, \dots \quad (2.41)$$

where $s^{(0)}$ is some initial approximation to s . To determine $\phi'(s^{(k)})$, we proceed as follows : Denote $y_s = y(x, s), y'_s = y'(x, s), y''_s = y''(x, s)$. Then, we can write 2.37 as

$$y''_s = f(x, y_s, y'_s) \quad (2.42)$$

$$y_s(a) = \frac{1}{a_0}(a_1 s + \gamma_1), y'_s(a) = s. \quad (2.43)$$

Differentiating 2.42 partially with respect to s , we get

$$\begin{aligned} \frac{\partial}{\partial s}(y''_s) &= \frac{\partial f}{\partial x} \frac{\partial x}{\partial s} + \frac{\partial f}{\partial y_s} \frac{\partial y_s}{\partial s} + \frac{\partial f}{\partial y'_s} \frac{\partial y'_s}{\partial s} \\ &= \frac{\partial f}{\partial y_s} \frac{\partial y_s}{\partial s} + \frac{\partial f}{\partial y'_s} \frac{\partial y'_s}{\partial s} \end{aligned} \quad (2.44)$$

since x is independent of s . Differentiating 2.43 partially with respect to s , we get

$$\frac{\partial}{\partial s}[y_s(a)] = \frac{a_1}{a_0}, \frac{\partial}{\partial s}[y'_s(a)] = 1. \quad (2.45)$$

Let $v = \frac{\partial y_s}{\partial s}$. Then, we have

$$\begin{aligned} v' &= \frac{\partial v}{\partial x} = \frac{\partial}{\partial x} \left(\frac{\partial y_s}{\partial s} \right) = \frac{\partial}{\partial s} \left(\frac{\partial y_s}{\partial x} \right) = \frac{\partial}{\partial s}(y'_s) \\ v'' &= \frac{\partial v'}{\partial x} = \frac{\partial}{\partial x} \left[\frac{\partial}{\partial s} \left(\frac{\partial y_s}{\partial x} \right) \right] = \frac{\partial}{\partial s} \left[\frac{\partial^2 y_s}{\partial x^2} \right] = \frac{\partial}{\partial s}(y''_s) \end{aligned}$$

From 2.43 and 2.44, we obtain

$$v'' = \frac{\partial f}{\partial y_s}(x, y_s, y'_s)v + \frac{\partial f}{\partial y'_s}(x, y_s, y'_s)v' \quad (2.46)$$

$$v(a) = \frac{a_1}{a_0}, v'(a) = 1. \quad (2.47)$$

The differential equation 2.46 is called the first variational equation. It can be solved step by step along with 2.43, that is, 2.43 and 2.46, 2.47 can be solved

together as a single system. When the computation of one cycle is completed, $v(b)$ and $v'(b)$ are available. Now, from 2.38, at $x = b$, we have

$$\frac{d\phi}{ds} = b_0 \frac{\partial y_s}{\partial s} + b_1 \frac{\partial y'_s}{\partial s} = b_0 v(b) + b_1 v'(b). \quad (2.48)$$

Thus, we have the value of $\phi'(s^{(k)})$ to be used in 2.41. If the boundary conditions of the first kind are given, then we have

$$a_0 = 1, a_1 = 0, b_0 = 1, b_1 = 0 \quad \text{and} \quad \phi(s) = y_s(b) - \gamma_2. \quad (2.49)$$

The initial conditions (2.45), on v become

$$v(a) = 0, v'(a) = 1.$$

Then, we have from (2.48)

$$\frac{d\phi}{ds} = v(b). \quad (2.50)$$

Example 2.3. Use the shooting method to solve the boundary value problem

$$\begin{aligned} u'' &= 2uu', 0 < x < 1 \\ u(0) &= 0.5, u(1) = 1. \end{aligned}$$

Use the Taylor series method

$$\begin{aligned} u_{j+1} &= u_j + hu'_j + \frac{h^2}{2}u''_j + \frac{h^3}{6}u'''_j \\ u'_{j+1} &= u'_j + hu''_j + \frac{h^2}{2}u'''_j \end{aligned} \quad (2.51)$$

to solve the corresponding initial value problems and the secant method for the iteration. Iterate until tolerance is less than 0.005. Assume $h = 0.25$ Compare with the exact solution $u(x) = 1/(2-x)$. Let the starting value of the slope at $x = 0$ be taken as $u'(0) = s^{(0)} = 0.5$. Therefore, we need to solve the initial value problem

$$u'' = 2uu'$$

$$u(0) = 0.5, u'(0) = s^{(0)} = 0.5.$$

Using the given Taylor series method and substituting

$u_j'' = 2u_j u_j'$, $u_j''' = 2[(u_j')^2 + u_j u_j'']$ with $h = 0.25$, $u_0 = 0.5$, $u_0' = 0.5$, in (2.51), we obtain

$$\begin{aligned} u_{j+1} &= u_j + hu_j' + \frac{h^2}{2}(2u_j u_j') + \frac{h^3}{3}[(u_j')^2 + u_j u_j''] \\ &= u_j + 0.25u_j' + 0.0625u_j u_j' + 0.00521[(u_j')^2 + u_j u_j''] \end{aligned}$$

$$u_{j+1} = u_j + 0.25u_j' + 0.0625u_j u_j' + 0.00521[(u_j')^2 + 2(u_j)^2 u_j'] \quad (2.52)$$

$$u_{j+1}' = u_j' + h(2u_j u_j') + h^2[(u_j')^2 + u_j(2u_j u_j')]$$

$$u_{j+1}' = u_j' + 0.5u_j u_j' + 0.0625[(u_j')^2 + 2(u_j)^2 u_j'] \quad (2.53)$$

We obtain from (2.52) and (2.53), for $j = 0, 1, 2, 3$

$$u(0.25) \approx u_1 = 0.64323, u'(0.25) \approx u_1' = 0.65625,$$

$$u(0.50) \approx u_2 = 0.83875, u'(0.50) \approx u_2' = 0.92817,$$

$$u(0.75) \approx u_3 = 1.13074, u'(0.75) \approx u_3' = 1.45289,$$

$$u(1.0) \approx u_4 = 1.62699, u'(1.0) \approx u_4' = 2.63844.$$

From (2.49), we get $\phi(s^{(0)}) = u(1, s^{(0)}) - 1.0 = 0.62699$.

We now take another value of the slope at $x = 0$ as $u'(0) = s^{(1)} = 0.1$. Therefore, we need to solve the equations (2.53) with $u_0 = 0.5$ and $u_0' = 0.1$.

We obtain, for $j = 0, 1, 2, 3$.

$$u(0.25) \approx u_1 = 0.52844, u'(0.25) \approx u_1' = 0.12875,$$

$$u(0.50) \approx u_2 = 0.56534, u'(0.50) \approx u_2' = 0.16830,$$

$$u(0.75) \approx u_3 = 0.61407, u'(0.75) \approx u_3' = 0.22437,$$

$$u(1.0) \approx u_4 = 1.67991, u'(1.0) \approx u_4' = 0.30698.$$

From (2.49), we get $\phi(s^{(1)}) = u(1, s^{(1)}) - 1.0 = -0.32009$.

Using the secant method (2.40), we obtain

$$s^{(2)} = s^{(1)} - \left[\frac{s^{(1)} - s^{(0)}}{\phi(s^{(1)}) - \phi(s^{(0)})} \right] \phi(s^{(1)}) = 0.1 - \left[\frac{0.1 - 0.5}{-0.32009 - 0.62699} \right] (-0.32009) = 0.23519 .$$

Now, we solve the equations (2.53) with $u_0 = 0.5$ and $u'_0 = 0.23519$. We obtain, for $j = 0, 1, 2, 3$

$$\begin{aligned} u(0.25) &\approx u_1 = 0.56705, u'(0.25) \approx u'_1 = 0.30479, \\ u(0.50) &\approx u_2 = 0.65555, u'(0.50) \approx u'_2 = 0.40926, \\ u(0.75) &\approx u_3 = 0.77734, u'(0.75) \approx u'_3 = 0.57586, \\ u(1.0) &\approx u_4 = 0.95464, u'(1.0) \approx u'_4 = 0.86390, \end{aligned}$$

From (2.49), we get $\phi(s^{(2)}) = u(1, s^{(2)}) - 1.0 = -0.04536$.

Using the secant method we obtain

$$s^{(3)} = s^{(2)} - \left[\frac{s^{(2)} - s^{(1)}}{\phi(s^{(2)}) - \phi(s^{(1)})} \right] \phi(s^{(2)}) = 0.23519 - \left[\frac{0.23519 - 0.1}{-0.04536 + 0.32009} \right] (-0.04536) = 0.25751 .$$

we solve the equations (2.53) with $u_0 = 0.5$ and $u'_0 = 0.25751$. We obtain, for $j = 0, 1, 2, 3$

$$\begin{aligned} u(0.25) &\approx u_1 = 0.57344, u'(0.25) \approx u'_1 = 0.33408, \\ u(0.50) &\approx u_2 = 0.67066, u'(0.50) \approx u'_2 = 0.45058, \\ u(0.75) &\approx u_3 = 0.80536, u'(0.75) \approx u'_3 = 0.63969, \\ u(1.0) &\approx u_4 = 1.00394, u'(1.0) \approx u'_4 = 0.97472, \end{aligned}$$

From (2.49), we get $\phi(s^{(3)}) = u(1, s^{(3)}) - 1.0 = 0.00394 < 0.005$. The iteration is now stopped.

The numerical and exact solutions are given in Table 5.3 . The maximum absolute error in the solutions occurs at $x = 0.75$ and its value is max. abs. error=0.00536.

Table 1: Solution of Example 2.51

x	<i>Exact</i> : $u(x)$	u_j
0.25	0.57143	0.57344
0.50	0.66667	0.67066
0.75	0.80000	0.80536
1.00	1.00000	1.00394

Example 2.4. Use the shooting method to solve the boundary value problem

$$u'' = 2uu', 0 < x < 1,$$

$$u(0) = 0.5, u(1) = 1,$$

Use the two stage Runge-Kutta method

$$k_1 = \frac{h^2}{2} f(x_j, u_j, u'_j)$$

$$k_2 = \frac{h^2}{2} f(x_j + \frac{2}{3}hu'_j + \frac{2}{3}k_1, u'_j + \frac{4}{3h}k_1)$$

$$u_{j+1} = u_j + hu'_j + \frac{1}{2}(k_1 + k_2)$$

$$u'_{j+1} = u'_j + \frac{1}{2h}(k_1 + 3k_2)$$

with $h = 0.25$, to solve the corresponding initial value problem. Use Newton's method, assuming the starting value of the slope at $x = 0$ as $(s^{(0)}) = u'(0) = 0.3$. Perform two iterations and compare with the exact solution $u(x) = 1/(2 - x)$.

We have $(s^{(0)}) = u'(0) = 0.3, u_0 = 0.5$.

The boundary value problem occurring in the application of the Newton-Raphson

method is given by (see Eqs.(2.46)(2.47)

$$\begin{aligned}v'' &= 2(u'v + uv') \\v(0) &= 0, v'(0) = 1.\end{aligned}\tag{2.54}$$

We solve the boundary value problems for u and v simultaneously. We have

$$\begin{aligned}(i) k_1 &= h^2 u_j u'_j \\k_2 &= h^2 \left(u_j + \frac{2}{3} h u'_j + \frac{2}{3} k_1 \right) \left(u'_j + \frac{4}{3h} k_1 \right) \\u_{j+1} &= u_j + h u'_j + \frac{1}{2} (k_1 + k_2) \\u'_{j+1} &= u'_j + \frac{1}{2h} (k_1 + 3k_2) \\(ii) k_1^* &= h^2 [u'_j v_j + u_j v'_j] \\k_2^* &= h^2 \left[u'_j \left(v_j + \frac{2}{3} h v'_j + \frac{2}{3} k_1^* \right) + u_j \left(v'_j + \frac{4}{3h} k_1^* \right) \right] \\v_{j+1} &= v_j + h v'_j + \frac{1}{2} (k_1^* + k_2^*) \\v'_{j+1} &= v'_j + \frac{1}{2h} (k_1^* + 3k_2^*)\end{aligned}$$

First iteration

We have the following results for $h = 0.25$.

For $j = 0 : u_0 = 0.5, u'_0 = 0.3, v_0 = 0, v'_0 = 1, h = 0.25$.

$$k_1 = 0.009375, k_2 = 0.01217, u_1 = 0.58577, u'_1 = 0.39177.$$

$$k_1^* = 0.03125, k_2^* = 0.03997, v_1 = 0.28561, v'_1 = 1.30232.$$

For $j = 1 : k_1 = 0.001434, k_2 = 0.01933, u_1 = 0.70055, u'_1 = 0.53643$.

$$k_1^* = 0.05467, k_2^* = 0.07155, v_1 = 0.67430, v'_2 = 1.84096.$$

For $j = 2 : k_1 = 0.02349, k_2 = 0.03332, u_3 = 0.86306, u'_3 = 0.78333$.

$$k_1^* = 0.10321, k_2^* = 0.13991, v_3 = 1.25610, v'_3 = 2.88684.$$

For $j = 3 : k_1 = 0.04225, k_2 = 0.06441, u_4 = 1.11222, u'_4 = 1.25429$.

$$k_1^* = 0.21722, k_2^* = 0.31035, v_4 = 2.24160, v'_4 = 5.18338.$$

Hence, we obtain from (2.34) and (2.50)

$$\phi(s^{(0)}) = u(b) - 1.0 = u_4 - 1.0 = 1.11222 - 1.0 = 0.11222$$

and

$$\frac{d\phi(s^{(0)})}{7} ds = v(b) = v_4 = 2.24160.$$

Newton-Raphson method gives the next approximation to the slope at $x = 0$ as

$$\begin{aligned} s^{(1)} &= s^{(0)} - \frac{\phi(s^{(0)})}{\phi'(s^{(0)})} \\ &= 0.3 - \frac{0.11222}{2.24160} = 0.24994. \end{aligned}$$

Second iteration

We have the following results for $h = 0.25$.

For $j = 0 : u_0 = 0.5, u'_0 = 0.24994, v_0 = 0, v'_0 = 1, h = 0.25$.

$$k_1 = 0.00781, k_2 = 0.00997, u_1 = 0.57138, u'_1 = 0.32538.$$

$$k_1^* = 0.03125, k_2^* = 0.03939, v_1 = 0.28532, v'_1 = 1.29884.$$

For $j = 1 : k_1 = 0.01162, k_2 = 0.01533, u_2 = 0.66620, u'_2 = 0.44060$.

$$k_1^* = 0.05219, k_2^* = 0.06724, v_2 = 0.66975, v'_2 = 1.80666.$$

For $j = 2 : k_1 = 0.01385, k_2 = 0.02530, u_3 = 0.79818, u'_3 = 0.62910$.

$$k_1^* = 0.09367, k_2^* = 0.12448, v_3 = 1.23049, v'_3 = 2.74088.$$

For $j = 3 : k_1 = 0.03138, k_2 = 0.04599, u_4 = 0.99414, u'_4 = 0.96780$.

$$k_1^* = 0.18511, k_2^* = 0.25718, v_4 = 2.13686, v'_4 = 4.65418.$$

Hence, we have

$$\phi(s^{(1)}) = u(b) - 1.0 = u_4 - 1.0 = 0.99414 - 1.0 = -0.00586$$

and

$$\frac{d\phi(s^{(1)})}{ds} = v(b) = 2.13686.$$

Newton-Raphson method gives the next approximation to the slope at $x = 0$ as

$$s^{(2)} = s^{(1)} - \frac{\phi(s^{(1)})}{\phi'(s^{(1)})} = 0.24994 + \frac{0.00586}{2.13686} = 0.25268.$$

We now solve only for u in the third iteration.

For $j = 0 : u_0 = 0.5, u'_0 = 0.25268$

$$k_1 = 0.00790, k_2 = 0.01009, u_1 = 0.57217, u'_1 = 0.32902.$$

For $j = 1 : k_1 = 0.01177, k_2 = 0.01555, u_2 = 0.66809, u'_2 = 0.44586$,

For $j = 2 : k_1 = 0.01862, k_2 = 0.02572, u_3 = 0.80173, u'_3 = 0.63742$.

For $j = 3 : k_1 = 0.03194, k_2 = 0.04691, u_4 = 1.00051, .$

Now, error in satisfying the boundary condition at $x = 1$ is

$$\text{error} = u(1) - u_4 = 1.0 - 1.00051 = 0.00051.$$

The solutions at the mesh points are

$$u(0.25) \approx u_1 = 0.57217, u(0.5) \approx u_2 = 0.66809, u(0.75) \approx u_3 = 0.80173.$$

The exact solution is

$$u(0.25) \approx u_1 = 0.57143, u(0.5) = 0.66667, u(0.75) = 0.80000.$$

The maximum absolute error in the solutions which occurs at $x = 0.75$, is $\text{max.abs.error} = 0.00173$.

3 Finite Difference Methods

Let the interval $[a, b]$ be divided into $N + 1$ subintervals, such that

$$x_j = a + jh, j = 1, 2, \dots, N + 1$$

where $x_0 = a, x_{N+1} = b$ and $h = (b-a)/(N + 1)$.

3.1 Linear Second Order Differential Equations

We consider the linear second order differential equation

$$y'' + p(x)y' + q(x)y = r(x) \tag{3.1}$$

subject to the boundary conditions of the first kind

$$y(a) = \gamma_1, y(b) = \gamma_2. \tag{3.2}$$

Using the second order finite difference approximations

$$\begin{aligned} y'(x_j) &\cong \frac{1}{2h}[y_{j+1} - y_{j-1}], \\ y''(x_j) &\cong \frac{1}{h^2}[y_{j+1} - 2y_j + y_{j-1}], \text{ at } x = x_j, \text{ we obtain the difference equation} \\ -\frac{1}{h^2}(y_{j+1} - 2y_j + y_{j-1}) + \frac{1}{2h}(y_{j+1} - y_{j-1})p(x_j) + q(x_j)y_j &= r(x_j) \end{aligned} \tag{3.3}$$

$j = 1, 2, \dots, N$.

The boundary conditions 3.2 become $y_0 = \gamma_1, y_{N+1} = \gamma_2$. Multiplying 3.3 by $h^2/2$, we obtain

$$A_j y_{j-1} + B_j y_j + C_j y_{j+1} = \frac{h^2}{2} r(x_j), j = 1, 2, \dots, N \quad (3.4)$$

where $A_j = -\frac{1}{2}(1 + \frac{h}{2}p(x_j))$, $B_j = (1 + \frac{h^2}{2}q(x_j))$, $C_j = -\frac{1}{2}(1 - \frac{h}{2}p(x_j))$.

The system 3.4 in matrix notation, after incorporating the boundary conditions, becomes

$$\mathbf{A}\mathbf{y} = \mathbf{b} \quad (3.5)$$

where $\mathbf{y} = [y_1, y_2, \dots, y_N]^T$

$$\mathbf{b} = \frac{h^2}{2} [r(x_1) - \frac{2A_1\gamma_1}{h^2}, r(x_2), \dots, r(x_{N-1}), r(x_N) - \frac{2C_N\gamma_2}{h^2}]^T$$

$$\mathbf{A} = \begin{bmatrix} B_1 & C_1 & & & \mathbf{0} \\ A_2 & B_2 & C_2 & & \\ \dots & \dots & \dots & \dots & \dots \\ & & A_{N-1} & B_{N-1} & C_{N-1} \\ \mathbf{0} & & & A_N & B_N \end{bmatrix}$$

The solution of this system of lin-

ear equations gives the finite difference solution of the differential equation 3.1 satisfying the boundary conditions 3.2.

3.2 Local Truncation Error

The local truncation error of 3.4 is defined by

$$T_j = A_j y(x_{j1}) + B_j y(x_j) + C_j y(x_{j+1}) - \frac{h^2}{2} r(x_j) \quad (3.6)$$

Expanding each term on the right hand side Taylors series about x_j , we get $T_j = -\frac{h^2}{24}[y^{(4)}(\psi_1) - 2p(x_j)y^{(3)}(\psi_2)], j = 1, 2, \dots, N$ where $\psi_1 \in (x_{j-1}, x_{j+1}), \psi_2 \in (x_{j-1}, x_{j+1})$.

The largest value of p for which the relation

$$T_j = O(h^{p+2}) \quad (3.7)$$

holds is called the order of the difference method. Therefore, the method is of second order.

3.3 Fourth Order Method when y' is Absent in 3.1

Consider the differential equation

$$-y'' + q(x)y = r(x), a < x < b \quad (3.8)$$

subject to the boundary conditions of the first kind uation

$$y(a) = \gamma_1, y(b) = \gamma_2. \quad (3.9)$$

We write the differential equation as

$$y'' = q(x)y - r(x) = f(x, y) \quad (3.10)$$

A fourth order difference approximation for 3.10 is obtained as

$$y_{j1} - 2y_j + y_{j+1} = \frac{h^2}{12}(y''_{j-1} + 10y''_j + y''_{j+1}), j = 1, 2, \dots, N, \quad (3.11)$$

which is also called the Numerov method. We can also write the method as

$$\left[1 - \frac{h^2}{12}q_{j-1}\right]u_{j-1} - \left[2 + \frac{5}{6}h^2q_j\right]u_j + \left[1 - \frac{h^2}{12}q_{j+1}\right]u_{j+1} = -\frac{h^2}{12}[r_{j-1} + 10r_j + r_{j+1}] \quad (3.12)$$

where $r_i = r(x_i)$, $q_i = q(x_i)$, $i = j-1, j, j+1$. The truncation error associated with 3.11 is given by $T_j = -\frac{h^6}{240}y^{(6)}(x_i)$, $x_{j-1} < (\psi)x_{j-1} < x_i < x_{j+1}$,

Example 3.1. Solve the boundary value problem

$$u'' = u + x$$

$$u(0) = 0, \quad u(1) = 0$$

with $h = 1/4$. Use the following methods

(i) the second order method, (ii) the Numerov method.

We divide the interval $[0, 1]$ into four subintervals. The nodal points are $x_j = jh$, $j = 0, 1, 2, 3, 4$ and $h = 1/4$.

(i) The second order method gives the following system of equations

$$\frac{u_{j-1} - 2u_j + u_{j+1}}{h^2} = u_j + x_j, \quad j = 1, 2, 3.$$

Multiplying by $-h^2$ we obtain

$$-u_{j-1} + 2u_j - u_{j+1} = -h^2(u_j + x_j), \quad j = 1, 2, 3.$$

For $h = 1/4$, we get

$$-16u_{j-1} + 33u_j - 16u_{j+1} = -x_j.$$

We have

$$\text{for } j = 1: \quad -16u_0 + 33u_1 - 16u_2 = -0.25.$$

$$\text{for } j = 2: \quad -16u_1 + 33u_2 - 16u_3 = -0.50.$$

$$\text{for } j = 3: \quad -16u_2 + 33u_3 - 16u_4 = -0.75.$$

Using the boundary conditions $u_0 = u_4 = 0$, we get the system of equations

$$\begin{bmatrix} 33 & -16 & 0 \\ -16 & 33 & -16 \\ 0 & -16 & 33 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = - \begin{bmatrix} 0.25 \\ 0.50 \\ 0.75 \end{bmatrix}$$

which gives

$$u_1 = -0.034885, \quad u_2 = -0.056326, \quad u_3 = -0.050037.$$

(ii) The Numerov method(3.11), with $q(x) = 1$, $r(x) = -x$, $h = 1/4$ gives the following system of equations

$$u_{j-1} - 2u_j + u_{j+1} = \frac{1}{192} [(u_{j-1} + x_{j-1}) + 10(u_j + x_j) + (u_{j+1} + x_{j+1})], \quad j = 1, 2, 3.$$

$$\text{or } 191u_{j-1} - 394u_j + 191u_{j+1} = x_{j-1} + 10x_j + x_{j+1}.$$

We have

$$\text{for } j = 1 : 191u_0 - 394u_1 + 191u_2 = 3.$$

$$\text{for } j = 2 : 191u_1 - 394u_2 + 191u_3 = 6.$$

$$\text{for } j = 3 : 191u_2 - 394u_3 + 191u_4 = 9.$$

Using the boundary conditions $u_0 = u_4 = 0$, we get the system of equations

$$-394u_1 + 191u_2 = 3$$

$$191u_1 - 394u_2 + 191u_3 = 6$$

$$-191u_2 - 394u_3 = 9$$

which gives

$$u(0.25) \approx u_1 = -0.0350481, \quad u(0.50) \approx u_2 = -0.0565914, \quad u(0.75) \approx u_3 = -0.0502765.$$

The exact solution is $u(x) = [(\sinh x / \sinh 1) - x]$ and the exact values are

$$u(0.25) = -0.0350476, \quad u(0.50) = -0.0565906, \quad u_{0.75} = -0.0502758.$$

The errors at the nodal points are

Second order method:

$$|\epsilon(0.25)| = 0.000163, \quad |\epsilon(0.50)| = 0.00265, \quad |\epsilon(1.0)| = 0.000239.$$

Numerov method

$$|\epsilon(0.25)| = 0.0000005, \quad |\epsilon(0.50)| = 0.0000008, \quad |\epsilon(1.0)| = 0.0000007.$$

Example 3.2. Solve the boundary value problem

$$u'' = xu$$

$$u(0) + u'(0) = 1, \quad u(1) = 1$$

with $h = 1/3$. Use the second order method

$$u_{j-1} - 2u_j + u_{j+1} = h^2 f_j.$$

With $h = 1/3$, we have four nodal points $x_j = jh$, $j = 0, 1, 2, 3$ that is 0, 1/3, 2/3, 1. The second order method gives the following system of equations

$$u_{j-1} - 2u_j + u_{j+1} = \frac{1}{9}x_j u_j, \quad j = 0, 1, 2, 3.$$

We have

$$\text{for } j = 0: \quad u_{-1} - 2u_0 + u_1 = 0.$$

$$\text{for } j = 1: \quad u_0 - 2u_1 + u_2 = \left(\frac{1}{27}\right)u_1.$$

$$\text{for } j = 2: \quad u_1 - 2u_2 + u_3 = \left(\frac{2}{27}\right)u_2.$$

Since the method is of second order, we may replace $u'(0)$ in the boundary condition by the approximation

$$u'(0) = (u_1 - u_{-1})/(2h)$$

which is also of second order. Thus, the boundary conditions become

$$u_0 + (3/2)(u_1 - u_{-1}) = 1 \quad \text{and} \quad u_3 = 1.$$

Eliminating u_{-1} , we get the equations

$$-2u_0 + 3u_1 = 1$$

$$u_0 - (55/27)u_1 + u_2 = 0$$

$$u_1 - (56/27)u_2 = -1.$$

Solving the system of equations we get

$$u(0) \approx u_0 = -0.9879518, \quad u(1/3) \approx u_1 = -0.3253012,$$

$$u(2/3) \approx u_2 = -0.3253012$$

Example 3.3. Solve the boundary value problem

$$u'' = u - 4xe^x, \quad 0 \leq x \leq 1$$

$$u(0) - u'(0) = -1, \quad u(1) + u'(1) = -e$$

using the Numerov method, with $h = 1/3$. Use suitable fourth order approximations to the boundary conditions. Compare with the exact solution $u(x) = x(1-x)e^x$.

Comparing the given differential equation with 3.10, we get $q(x) = 1$, $d(x) = 4xe^{-x}$. With $h = 1/3$, we have the mesh points as 0, 1/3, 2/3, 1.

At the interior points, we have from 3.12

$$\left(1 - \frac{1}{108}\right) u_{j-1} - \left(2 + \frac{5}{54}\right) u_j + \left(1 - \frac{1}{108}\right) u_{j+1} = -\frac{4}{108} [x_{j-1}e^{x_{j-1}} + 10x_je^{x_j} + x_{j+1}e^{x_{j+1}}]$$

$$\text{or} \quad 107u_{j-1} - 226u_j + 107u_{j+1} = -4 [x_{j-1}e^{x_{j-1}} + 10x_je^{x_j} + x_{j+1}e^{x_{j+1}}], \quad j = 1, 2$$

We now derive the discretization of the boundary condition at $x = 0$. From

$$\frac{1}{h} (u_1 - u_0) - \frac{h}{6} [u_0'' + 2u_{1/2}''] = \frac{1}{a_1} (a_0u_0 - \gamma_1), \quad (3.13)$$

we get

$$\frac{1}{h} (u_1 - u_0) - \frac{h}{6} [u_0'' + 2u_{1/2}''] = \frac{1}{a_1} (a_0u_0 - \gamma_1)$$

$$\text{or} \quad 3(u - u_0) - \frac{1}{18} [u_0'' + 2u_{1/2}''] = 1 + u_0.$$

From $u_{1/2} = u_0 + \frac{h}{2}u_0' + \frac{h^2}{8}u_0''$, we get

$$u_{1/2} = u_0 + \frac{h}{2}u_0' + \frac{h^2}{8}u_0'' = u_0 + \frac{1}{6}u_0' + \frac{1}{72}u_0''$$

$$\text{Now,} \quad u_0'' = f(0, u_0) = u_0 \quad \text{and} \quad u_0' = 1 + u_0$$

Therefore,

$$u_{1/2} = u_0 + \frac{1}{6}(1 + u_0) + \frac{1}{72}u_0'' = \frac{1}{72}(85u_0 + 12)$$

$$u_{1/2}'' = f(x_{1/2}, u_{1/2}) = u_{1/2} - 4x_{1/2}e^{x_{1/2}}$$

$$= u_{1/2} - \frac{2}{3}e^{x_{1/2}} = \frac{1}{72}(85u_0 + 12) - \frac{2}{3}e^{1/6}$$

Hence, we have at $x = 0$

$$3(u_1 - u_0) - \frac{1}{18} \left[u_0 + \frac{1}{36}(85u_0 + 12) - \frac{4}{3}e^{1/6} \right] = 1 + u_0$$

or

$$-4.18673u_0 + 3u_1 = 0.93101.$$

For $j = 1$, we get

$$107u_0 - 226u_1 + 107u_2 = -4 \left[0 + \frac{10}{3}e^{1/3} + \frac{2}{3}e^{2/3} \right] = -23.80212.$$

For $j = 2$, we get

$$107u_1 - 226u_2 + 107u_3 = -4 \left[\frac{1}{3}e^{1/3} + \frac{20}{3}e^{2/3} + e \right] = -64.67351.$$

We now derive the discretization at $x = 1$.

From $\frac{1}{h}(u_{N+1} - u_N) + \frac{h}{6}(2u''_{N+1/2} + u''_{N+1}) = \frac{1}{b_1}(\gamma_2 - b_0u_{N+1})$, we get (with $N=2$)

$$\frac{1}{h}(u_3 - u_2) + \frac{h}{6}(2u''_{5/2} + u''_3) = \frac{1}{b_1}(\gamma_2 - b_0u_3)$$

or

$$3(u_3 - u_2) + \frac{1}{18}(2u''_{5/2} + u''_3) = -e - u_3.$$

Now,

$$u''_3 = f(x_3, u_3) = u_3 - 4x_3e^{x_3} = u_3 - 4e.$$

From $u_{N+1/2} = u_{N+1} - \frac{h}{2}u'_{N+1} + \frac{h^2}{8}u''_{N+1}$, we get (with $N=2$)

$$\begin{aligned} u_{5/2} &= u_3 - \frac{1}{6}u'_3 + \frac{1}{72}u''_3 \\ &= u_3 + \frac{1}{6}(e + u_3) + \frac{1}{72}(u_3 - 4e) \\ &= \frac{1}{72}(85u_3 + 8e) = 1.18056u_3 + 0.30203. \end{aligned}$$

$$\begin{aligned} u''_{5/2} &= f(x_{5/2}, u_{5/2}) = u_{5/2} - 4x_{5/2}e^{x_{5/2}} \\ &= \frac{1}{72}(85u_3 + 8e) - \frac{10}{3}e^{5/6} \\ &= 1.18056u_3 + 7.36789. \end{aligned}$$

Therefore, we get

$$3(u_3 - u_2) + \frac{1}{18} [2(1.18056u_3 - 7.36789) + u_3 - 4e] = -e - u_3$$

or
$$-3u_2 + 4.18673u_3 = -1.29556 .$$

Hence, we have the system of equations

$$\begin{bmatrix} -4.18673 & 3 & 0 & 0 \\ 107 & -226 & 107 & 0 \\ 0 & 107 & -226 & 107 \\ 0 & 0 & -3 & 4.18673 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = - \begin{bmatrix} 0.93101 \\ -23.80212 \\ -64.67351 \\ -1.29556 \end{bmatrix}$$

We solve the system by Gauss-elimination. The solution is obtained as

$$u_0 = 0.00076, \quad u_1 = 0.31140, \quad u_2 = 0.43350, \quad u_3 = 0.00190 .$$

The exact solution is $u(0) = 0$, $u(1/3) = 0.31014$, $u(2/3) = 0.43283$ and $u(1) = 0$. The maximum absolute error is 0.0019.

3.4 Nonlinear Second Order Differential Equations $y'' = f(x, y)$

We consider the nonlinear second order differential equation

$$y'' = f(x, y), \quad a < x < b \tag{3.14}$$

subject to one of the boundary conditions (1.2) to (1.4). Since the differential equation is non linear, we cannot write the solution in the form $y(x) = \phi_0(x) + \mu_1\phi_1(x) + \mu_2\phi_2(x)$ or (2.27). Depending on the boundary conditions, we proceed

as follows :

Boundary condition of the first kind :

We have the boundary conditions as $y(a) = \gamma_1$ and $y(b) = \gamma_2$.

Example 3.4. Solve the boundary value problem

$$u'' = \frac{3}{2}u^2$$
$$u(0) = 4, u(1) = 1$$

with $h = 1/3$. Use a second order finite difference method for its solution.

With $h = 1/3$, we have four nodal points $x_j = jh, j = 0, 1, 2, 3$, that is , $0, 1/3, 2/3, 1$. The values at the end points x_0 and x_3 are given by the boundary conditions, that is, we are given that $u_0 = 4, u_3 = 1$. The second order finite difference method gives the following system of equations

$$u_{j-1} - 2u_j + u_{j+1} = (u_j^2)/6, j = 1, 2.$$

For $j = 1 : u_0 - 2u_1 + u_2 = (u_1^2)/6$.

For $j = 2 : u_1 - 2u_2 + u_3 = (u_2^2)/6$.

Using the boundary conditions $u_0 = 4, u_3 = 1$ we get the equations

$$F_1(u_1, u_2) = u_1^2 + 12u_1 - 6u_2 - 24 = 0.$$

$$F_2(u_1, u_2) = u_2^2 - 6u_1 + 12u_2 - 6 = 0.$$

This system of equations can be solved by any iterative method. We use the Newton - Raphson method to solve this system. We have

$$\frac{\partial F_1}{\partial u_1} = 2u_1 + 12, \frac{\partial F_1}{\partial u_2} = -6,$$

$$\frac{\partial F_2}{\partial u_1} = -6, \frac{\partial F_2}{\partial u_2} = 2u_2 + 12.$$

$$J(u^{(s)})\Delta u^{(s)} = \begin{bmatrix} \frac{\partial F_1}{\partial u_1} & \frac{\partial F_1}{\partial u_2} \\ \frac{\partial F_2}{\partial u_1} & \frac{\partial F_2}{\partial u_2} \end{bmatrix} \begin{bmatrix} \Delta u_1^{(s)} \\ \Delta u_2^{(s)} \end{bmatrix}$$

Therefore, we have

$$\begin{bmatrix} 2u_1^{(s)} + 12 & -6 \\ -6 & 2u_2^{(s)} + 12 \end{bmatrix} \begin{bmatrix} \Delta u_1^{(s)} \\ \Delta u_2^{(s)} \end{bmatrix} = - \begin{bmatrix} (u_1^{(s)})^2 + 12u_1^{(s)} - 6u_2^{(s)} - 24 \\ (u_2^{(s)})^2 - 6u_1^{(s)} + 12u_2^{(s)} - 6 \end{bmatrix}$$

Since J is a 2×2 matrix, we may invert it and solve the system. We obtain

$$\begin{bmatrix} \Delta u_1^{(s)} \\ \Delta u_2^{(s)} \end{bmatrix} = -\frac{1}{D} \begin{bmatrix} 2u_2^{(s)} + 12 & 6 \\ 6 & 2u_1^{(s)} + 12 \end{bmatrix} \begin{bmatrix} (u_1^{(s)})^2 + 12u_1^{(s)} - 6u_2^{(s)} - 24 \\ (u_2^{(s)})^2 - 6u_1^{(s)} + 12u_2^{(s)} - 6 \end{bmatrix}$$

The next iterate is obtained from $\begin{bmatrix} u_1^{(s+1)} \\ u_2^{(s+1)} \end{bmatrix}$ where $D = (2u_1^{(s)} + 12)(2u_2^{(s)} + 12) -$

36.

The next iterate is obtained by

$$\begin{bmatrix} u_1^{(s+1)} \\ u_2^{(s+1)} \end{bmatrix} = \begin{bmatrix} u_1^{(s)} \\ u_2^{(s)} \end{bmatrix} + \begin{bmatrix} \Delta u_1^{(s)} \\ \Delta u_2^{(s)} \end{bmatrix}, s = 0, 1, 2, \dots$$

Taking $u_1^{(0)} = 2$ and $u_2^{(0)} = 1.5$, we obtain the following results.

$$u_1^{(1)} = 2.3014706, u_2^{(1)} = 1.4705882,$$

$$u_1^{(2)} = 2.2950429, u_2^{(2)} = 1.4679491,$$

$$u_1^{(3)} = 2.3014706, u_2^{(3)} = 1.4679474.$$

Example 3.5. Obtain the numerical solution of the nonlinear boundary value problem

$$u'' = \frac{1}{2}(1 + x + u)^3.$$

$$u'(0) - u(0) = -1/2, u'(1) + u(1) = 1.$$

with $h = 1/2$. Use a second order finite difference method.

The nodal points are $x_0 = 0, x_1 = 1/2, x_2 = 1$. We have $a_0 = -1, a_1 = -1, \gamma_1 = -1/2, b_0 = b_1 = \gamma_2 = 1$.

The system of nonlinear equations $-u_N + (1 + \alpha)u_{N+1} + \frac{h^2}{6}(f_N + 2f_{N+1}) = \frac{h\gamma_2}{b_1}$ becomes

$$(1 + h)u_0 - u_1 + \frac{h^2}{2}[\frac{1}{3}(1 + x_0 + u_0)^3 + \frac{1}{6}(1 + x_1 + u_1)^3] - \frac{h}{2} = 0.$$

$$-u_0 + 2u_1 - u_2 + \frac{h^2}{2}(1 + x_1 + u_1)^3.$$

$$-u_1 + (1 + h)u_2 + \frac{h^2}{2}[\frac{1}{6}(1 + x_1 + u_1)^3 + \frac{1}{3}(1 + x_2 + u_2)^3] - h = 0.$$

The Newton - Raphson method gives the following linear equations

$$\begin{bmatrix} \frac{3}{2} + \frac{1}{8}(1 + u_0^{(s)})^2 & -1 + \frac{1}{16}(\frac{3}{2} + u_1^{(s)})^2 & 0 \\ -1 & 2 + \frac{3}{8}(\frac{3}{2} + u_1^{(s)})^2 & -1 \\ 0 & -1 + \frac{1}{16}(\frac{3}{2} + u_1^{(s)})^2 & \frac{3}{2} + \frac{1}{8}(2 + u_2^{(s)})^2 \end{bmatrix} \begin{bmatrix} \Delta u_0^{(s)} \\ \Delta u_1^{(s)} \\ \Delta u_2^{(s)} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{3}{2}u_0^{(s)} - u_1^{(s)} + \frac{1}{8}[\frac{1}{3}(1 + u_0^{(s)})^3 + \frac{1}{6}(\frac{3}{2} + u_1^{(s)})^2] - \frac{1}{4} \\ -u_0^{(s)} + 2u_1^{(s)} - u_2^{(s)} + \frac{1}{8}(\frac{3}{2} + u_1^{(s)})^3 \\ -u_1^{(s)} + \frac{3}{2}u_2^{(s)} + \frac{1}{8}[\frac{1}{6}(\frac{3}{2} + u_1^{(s)})^3 + \frac{1}{3}(2 + u_2^{(s)})^3] - \frac{1}{2} \end{bmatrix}$$

and $u_0^{(s+1)} = u_0^{(s)} + \Delta u_0^{(s)}, u_1^{(s+1)} = u_1^{(s)} + \Delta u_1^{(s)},$

$$u_2^{(s+1)} = u_2^{(s)} + \Delta u_2^{(s)}.$$

Using $u_0^{(0)} = 0.001, u_1^{(0)} = -0.1, u_2^{(0)} = -0.0228,$ we get after three iterations

$$u_0^{(3)} = -0.0023, u_1^{(3)} = -0.1622, u_2^{(3)} = -0.0228.$$

The analytical solution of the boundary problem is

$$u(x) = \frac{2}{2-x} - x - 1.$$

$$u(0) = 0, u(1/2) = -0.1667, u(1) = 0.$$

Example 3.6. Solve the boundary value problem

$$u'' = u' + 1$$

$$u(0) = 1, u(1) = 2(e - 1),$$

Using a (i) second order method, (ii) fourth order method, with $h = 1/3.$ Compare with the exact solution $u(x) = 2e^x - x - 1.$

For $h = 1/3,$ we have four nodal points, $x_0 = 0, x_1 = 1/3, x_2 = 2/3, x_3 = 1.$

The values of u at x_0 and x_3 are given from the boundary conditions. We have $u_0 = 1, u_3 = 2(e - 1),$

The second order method (7.108) gives the difference equation

$$u_{j-1} - 2u_j + u_{j+1} = h^2(u'_j + 1) = h^2 \left[\frac{u_{j+1} - u_{j-1}}{2h} + 1 \right]$$

or $(1 + \frac{h}{2}) u_{j-1} - 2u_j + (1 - \frac{h}{2}) u_{j+1} = h^2, j = 1, 2.$

For $h = 1/3$ and $j = 1, 2$ we get the system of equations

$$(7/6)u_0 - 2u_1 + (5/6)u_2 = (1/9)$$

$$(7/6)u_1 - 2u_2 + (5/6)u_3 = (1/9)$$

Using the boundary conditions $u_0 = 1, u_3 = 2(e - 1)$ and simplifying, we obtain

$$-36u_1 + 15u_2 = -19$$

$$21u - 36u_2 = 32 - 30e = -49.548455.$$

Solving these equations, we get $u_1 = 1.454869, u_2 = 2.225019$.

(ii) We discretize the differential equation by the fourth order method (7.110).

We obtain

$$\begin{aligned} u_{j-1} - 2u_j + u_{j+1} &= \frac{h^2}{12} [\bar{f}_{j+1} + 10\hat{f}_j + 1\bar{f}_{j-1}] \\ u_{j-1} &= \frac{h^2}{12} [(\bar{u}'_{j+1} + 1) + 10(\hat{u}'_j + 1) + (\bar{u}_{j-1} + 1)], j = 1, 2. \end{aligned}$$

Using the approximations (7.111), we get

$$\begin{aligned} u_{j-1} - 2u_j + u_{j+1} &= \frac{h^2}{12} [(\bar{u}'_{j+1} + \bar{u}'_{j-1} + 12 + 10\{\bar{u}'_j - \frac{h}{20}(\bar{f}_{j+1} - \bar{f}_{j-1})\}] \\ &= \frac{h^2}{12} [12 + \bar{u}'_{j+1} + 10\bar{u}'_j + \bar{u}'_{j-1} - \frac{h}{2}\{(\bar{u}'_{j+1} + 1) - (\bar{u}_{j-1} + 1)\}] \\ &= \frac{h^2}{12} [12 + (1 - \frac{h}{2})\bar{u}'_{j+1} + 10\{\bar{u}'_j + (1 + \frac{h}{2})(\bar{u}'_{j-1})\}] \\ &= \frac{h^2}{12} [12 + \frac{1}{2h}(1 - \frac{h}{2})(3u_{j+1} - 4u_j + u_{j-1}) + \frac{10}{2h}(u_{j+1} - u_{j-1}) + \\ &\quad \frac{1}{2h}(1 + \frac{h}{2})(-u_{j+1} + 4u_j - 3u_{j-1})] \end{aligned}$$

Simplifying, we obtain the difference equation

$$\left(1 + \frac{h}{2} + \frac{h^2}{12}\right) u_{j-1} - \left(2 + \frac{h^2}{6}\right) u_j + \left(1 - \frac{h}{2} + \frac{h^2}{12}\right) u_{j+1} = h^2.$$

For $h = 1/3$ and $j = 1, 2$, we get the system of equations

$$\begin{aligned} \frac{127}{108}u_0 - \frac{109}{54}u_1 + \frac{91}{108}u_2 &= \frac{1}{9} \\ \frac{127}{108}u_1 - \frac{109}{54}u_2 + \frac{91}{108}u_3 &= \frac{1}{9} \end{aligned}$$

Using the boundary conditions, we obtain

$$91u_2 - 218u_1 = -115$$

$$218u_2 + 127u_1 = 12 - 91(2e - 2) = -300.727293.$$

Solving these equations, we get $u_1 = 1.457897, u_2 = 2.228808$.

Exact solution is $u(1/3) = 1.457892, u(2/3) = 2.228801$.

Example 3.7. Solve the boundary value problem

$$u'' = u + x$$

$$u(0) = 0, \quad u(1) = 0$$

with $h = 1/4$. Use the following methods

(i) the second order method, (ii) the Numerov method.

We divide the interval $[0, 1]$ into four subintervals. The nodal points are $x_j = jh$, $j = 0, 1, 2, 3, 4$ and $h = 1/4$.

(i) The second order method gives the following system of equations

$$\frac{u_{j-1} - 2u_j + u_{j+1}}{h^2} = u_j + x_j, \quad j = 1, 2, 3.$$

Multiplying by $-h^2$ we obtain

$$-u_{j-1} + 2u_j - u_{j+1} = -h^2(u_j + x_j), \quad j = 1, 2, 3.$$

For $h = 1/4$, we get

$$-16u_{j-1} + 33u_j - 16u_{j+1} = -x_j.$$

We have

$$\text{for } j = 1: \quad -16u_0 + 33u_1 - 16u_2 = -0.25.$$

$$\text{for } j = 2: \quad -16u_1 + 33u_2 - 16u_3 = -0.50.$$

$$\text{for } j = 3: \quad -16u_2 + 33u_3 - 16u_4 = -0.75.$$

Using the boundary conditions $u_0 = u_4 = 0$, we get the system of equations

$$\begin{bmatrix} 33 & -16 & 0 \\ -16 & 33 & -16 \\ 0 & -16 & 33 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = - \begin{bmatrix} 0.25 \\ 0.50 \\ 0.75 \end{bmatrix}$$

which gives

$$u_1 = -0.034885, \quad u_2 = -0.056326, \quad u_3 = -0.050037.$$

(ii) The Numerov method, with $q(x) = 1$, $r(x) = -x$, $h = 1/4$ gives the

following system of equations

$$u_{j-1} - 2u_j + u_{j+1} = \frac{1}{192} [(u_{j-1} + x_{j-1}) + 10(u_j + x_j) + (u_{j+1} + x_{j+1})], \quad j = 1, 2, 3.$$

or $191u_{j-1} - 394u_j + 191u_{j+1} = x_{j-1} + 10x_j + x_{j+1}.$

We have

$$\text{for } j = 1 : 191u_0 - 394u_1 + 191u_2 = 3.$$

$$\text{for } j = 2 : 191u_1 - 394u_2 + 191u_3 = 6.$$

$$\text{for } j = 3 : 191u_2 - 394u_3 + 191u_4 = 9.$$

Using the boundary conditions $u_0 = u_4 = 0$, we get the system of equations

$$\begin{aligned} -394u_1 + 191u_2 &= 3 \\ 191u_1 - 394u_2 + 191u_3 &= 6 \\ -191u_2 - 394u_3 &= 9 \end{aligned}$$

which gives

$$u(0.25) \approx u_1 = -0.0350481, \quad u(0.50) \approx u_2 = -0.0565914, \quad u(0.75) \approx u_3 = -0.0502765.$$

The exact solution is $u(x) = [(\sinh x / \sinh 1) - x]$ and the exact values are

$$u(0.25) = -0.0350476, \quad u(0.50) = -0.0565906, \quad u_{0.75} = -0.0502758.$$

The errors at the nodal points are

Second order method:

$$|\epsilon(0.25)| = 0.000163, \quad |\epsilon(0.50)| = 0.00265, \quad |\epsilon(1.0)| = 0.000239.$$

Numerov method

$$|\epsilon(0.25)| = 0.0000005, \quad |\epsilon(0.50)| = 0.0000008, \quad |\epsilon(1.0)| = 0.0000007.$$

Example 3.8. Solve the boundary value problem

$$u'' = xu$$

$$u(0) + u'(0) = 1, \quad u(1) = 1$$

with $h = 1/3$. Use the second order method

$$u_{j-1} - 2u_j + u_{j+1} = h^2 f_j.$$

With $h = 1/3$, we have four nodal points $x_j = jh$, $j = 0, 1, 2, 3$ that is

0, 1/3, 2/3, 1. The second order method gives the following system of equations

$$u_{j-1} - 2u_j + u_{j+1} = \frac{1}{9}x_j u_j, \quad j = 0, 1, 2, 3.$$

We have

$$\text{for } j = 0 : u_{-1} - 2u_0 + u_1 = 0.$$

$$\text{for } j = 1 : u_0 - 2u_1 + u_2 = \left(\frac{1}{27}\right)u_1.$$

$$\text{for } j = 2 : u_1 - 2u_2 + u_3 = \left(\frac{2}{27}\right)u_2.$$

Since the method is of second order, we may replace $u'(0)$ in the boundary condition by the approximation

$$u'(0) = (u_1 - u_{-1})/(2h)$$

which is also of second order. Thus, the boundary conditions become

$$u_0 + (3/2)(u_1 - u_{-1}) = 1 \quad \text{and} \quad u_3 = 1.$$

Eliminating u_{-1} , we get the equations

$$-2u_0 + 3u_1 = 1$$

$$u_0 - (55/27)u_1 + u_2 = 0$$

$$u_1 - (56/27)u_2 = -1.$$

Solving the system of equations we get

$$u(0) \approx u_0 = -0.9879518, \quad u(1/3) \approx u_1 = -0.3253012,$$

$$u(2/3) \approx u_2 = -0.3253012$$

Example 3.9. Solve the boundary value problem

$$u'' + u = x, \quad 0 < x < 1$$

$$u(0) = 4, \quad u(1) = 1$$

using the Ritz finite element method with linear piecewise polynomials for two and three elements of equal lengths. Compare with the exact solution $u(x) = x + 4[\sin(1-x)/\sin 1]$.

(i) **Two elements of equal length.** From equation $\frac{1}{12}$

$$\begin{bmatrix} -22 & 25 & 0 \\ 25 & -22 & 25 \\ 0 & 25 & -22 \end{bmatrix}$$

$$\begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix} - \frac{1}{24} \begin{bmatrix} 1 \\ 2+4 \\ 5 \end{bmatrix} = 0, \text{ we have}$$

$$\begin{bmatrix} -22 & 25 & 0 \\ 25 & -44 & 25 \\ 0 & 25 & -22 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 3 \\ 5/2 \end{bmatrix}$$

Using the procedure described in the text

$$\begin{bmatrix}
 1 & \bullet & \bullet & \bullet & \dots & \bullet \\
 \bullet & \times & \times & & \dots & \\
 \bullet & \times & \times & \times & \dots & \\
 \vdots & & & \times & \times & \bullet \\
 \bullet & \bullet & \bullet & \bullet & \bullet & 1
 \end{bmatrix}
 \begin{bmatrix}
 u_0 \\
 u_1 \\
 u_2 \\
 \vdots \\
 u_N \\
 u_{N+1}
 \end{bmatrix}
 =$$

$$\begin{bmatrix}
 \gamma_1 \\
 \times \\
 \times \\
 \vdots \\
 \times \\
 \gamma_2
 \end{bmatrix}
 - \gamma_1
 \begin{bmatrix}
 \bullet \\
 \times \\
 \times \\
 \vdots \\
 \times \\
 \times
 \end{bmatrix}
 - \gamma_2
 \begin{bmatrix}
 \times \\
 \times \\
 \times \\
 \vdots \\
 \times \\
 \bullet
 \end{bmatrix}
 , \text{ we can incorporate the boundary conditions}$$

$u_0 = 4, u_2 = 1$ and obtain the system

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -44 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 1 \end{bmatrix} - 4 \begin{bmatrix} 0 \\ 25 \\ 0 \end{bmatrix} - 1 \begin{bmatrix} 0 \\ 25 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ -122 \\ 1 \end{bmatrix} \quad \text{The solution}$$

is $u_1 = \frac{122}{44} = 2.77273$.

The exact solution is $u(0.5) = 2.77273$.

(ii) **Three elements of equal length.** From equation $\frac{1}{18}$

$$\begin{bmatrix} -52 & 55 & 0 & 0 \\ 55 & -104 & 55 & 0 \\ 0 & 55 & -104 & 55 \\ 0 & 0 & 55 & -52 \end{bmatrix}$$

$$\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} - \frac{1}{54} \begin{bmatrix} 1 \\ 6 \\ 12 \\ 8 \end{bmatrix} = 0, \text{ we get}$$

$$\begin{bmatrix} -52 & 55 & 0 & 0 \\ 55 & -104 & 55 & 0 \\ 0 & 55 & -104 & 55 \\ 0 & 0 & 55 & -52 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 2 \\ 4 \\ 8/3 \end{bmatrix}$$

Incorporating the boundary conditions $u_0 = 4, u_2 = 1$, we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -104 & 55 & 0 \\ 0 & 55 & -104 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 4 \\ 1 \end{bmatrix} - 4 \begin{bmatrix} 0 \\ 55 \\ 0 \\ 0 \end{bmatrix} - 1 \begin{bmatrix} 0 \\ 0 \\ 55 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ -218 \\ -51 \\ 1 \end{bmatrix}$$

Solving the resulting system of equations

$$\begin{bmatrix} -104 & 55 \\ 55 & -104 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} -218 \\ -51 \end{bmatrix}$$

We obtain $u_1 = 3.270055, u_2 = 2.2219741$.

The exact solution is $u(1/3) = 3.272804, u(2/3) = 2.222013$.