C Language

A Concise Book

Nilesh Ghule
Technical Head, SunBeam Infotech

Sunbeam Series

Scanned by CamScanner

CONTENTS

Scanned by CamScanner

Scanned by CamScanner

Scanned by CamScanner

Scanned by CamScanner

Scanned by CamScanner

Chapter 1 : Introduction

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972 by Dennis Ritchie. C was invented during development of UNIX operating system. UNIX was developed on PDP–7 machine using assembly and PL/1 programming language by Ken Thompson, Dennis Ritchie and team. During porting of this operating system on PDP–11 machine, they tried to use B language, which was written by Ken Thompson. Since B language was having some limitations, the new language was put forward by Dennis Ritchie called C language. After development of C many UNIX kernels were written using C language.

Development of C

The first version of C language was created by Dennis Ritchie called as K & R (Kernighan and Ritchie) version. It is also known as traditional C. This language is explained in The *C Programming Language* book by Kernighan and Ritchie in 1978. The language became popular soon. In fact it became most widely used programming language. However, the language was not standardized. The different compilers for C language added many extra features in the language.

To have uniform implementation of C language, a standardization committee was appointed by the ANSI i.e. American National Standards Institute. The standardized version is called as *ANSI C programming language.* This was first universal standard of C language done in year 1989. This language is also referred as C89. The compiler created as per this standard can be called as *ANSI C Compilers*. This book is mainly focused on this version of C language.

This standard was revised in 1994 with minor changes. The new standard version of the language is called as *C89 with Amendment 1* or *C95*. This standard was once again revised in 1999. This was major revision in the standard; However, the old syntax was not modified. This standard added many new features and libraries, which includes complex arithmetic, variable length array, etc. The new standard version of the language is called as *C99*.

C is general purpose programming language; but widely used for system level programming. Today embedded systems, is one of the prime domain of C. The C language has been customized with additional keywords for certain family of micro–controllers. One of the well known customization is C51 used for 8051 family of the microcontrollers.

Classification of Computer Languages

Till today many programming languages are developed. Each language is useful for some domain. However, these languages can be broadly classified as follows.

Low Level languages or machine oriented languages

These languages had been designed to give better machine efficiency i.e. closer access to hardware and programs written using these languages require relatively small amount of memory. However, these languages are difficult to learn and also they depend on machine architecture. All machine languages and assembly languages are low level languages. Note that machine language code use numbers to represent instructions and storage locations, so it can be directly fed to processors or micro controllers. However, assembly language code can be converted into the machine language by using a special program called as an *assembler*.

High level languages or problem oriented languages

These languages have been designed to give better programming efficiency i.e. easier program

Scanned by CamScanner

development. These languages are easier to learn and also they are independent of machine architecture. Basic, Visual Basic, FORTRAN are few examples of high level language. High level language source codes must be converted to machine level code in order to execute them. This job is done with the help of "compiler" or in some cases by "interpreter".

Assembler is system software that translates assembly language code into the machine level code. For every instruction of the assembly language corresponding instruction of the machine language is used.

Compiler is a program that translates high level language code (source program) into the machine level code (object code). For every instruction of the high level language code is translated into one or more instructions of the machine level code. Compilers also detect and indicate the syntax errors. If program has any error, object code will not be created by the Compiler. Also note that compiler cannot detect logical errors.

Large programs are often written into multiple modules, so that they are easy to develop among multiple programmers and are also more readable. Also often common functionalities can be bunched together to create the library. In fact, C compiler software (and also Compiler software for other programming languages comes with a standard library of functions. Linker is a program that is used to combine all such modules and libraries together to build final executable. Input for linker is *object code* while its output is *executable code.* (The detailed information of an Compiler and linker is given in chapter 5.)

Interpreter is a program that translates high level language code into the machine level code. It translates one Instruction of high level language code into the one or more instructions of the machine language, which are then immediately executed. Then Interpreter continues with next instruction of the high level language code and execute it in same way. Note that interpreter neither store object code in any file, nor it shows any kind of error before execution of the program. Errors are shown during execution of the program wherever necessary. Also translation is done every time when program is executed.

C language was designed to give better machine efficiency along with its simplicity. So it can be thought of as middle level language. It gives better programming efficiency than low level languages and also better machine efficiency than high level languages. Most importantly C source code is highly portable across different machines. In fact, this was the key behind writing UNIX kernel in C, so that it can be easily ported across any machine. Also C is a free form programming language, which means that C source code do not have any compulsion on the way program is indented.

Even though indenting program is not required for its functionality, However, It is strongly recommended. Indentation of the program is very much required for its readability. There are number of good ways of indentation. One of the standard ways of indentation is followed throughout this book.

This chapter discussed the brief history of C language, different versions of C and finally classification of languages along with introduction to compiler and linker.

Objective Questions :

Q.1. C is developed in the year _____

1) 1969
2) 1989
3) 1972
4) 1999

Q.2. C is developed by _____

1) Ken Thompson

Scanned by CamScanner

2) Brian Kernighan
3) Dennis Ritchie
4) Bjarne Stroustrup

Q.3. C is standardized in the year _____ and this standard is revised in the year_____.

1) 1972, 1989, 1994
2) 1989, 1994, 2000
3) 1989, 1994, 1999
4) 1972, 1994, 1999

Q.4. C is a _____ Language.

1) High Level
2) Low Level
3) Middle Level
4) Interpreted

Q.5. C language is designed while porting _____ operating system on _____.

1) windows, 8086
2) UNIX, PDP11
3) UNIX, 4040
4) LINUX, PDP11

Scanned by CamScanner

Chapter 2 : Getting Started

In previous chapter the history of C language was discussed. Now it is time to start actual C programming. This chapter gathers the basics required for writing programs. The fundamental concepts required to begin C programming are discussed in this chapter. It also discusses writing simple C programs.

Identifiers

Any program of C contains some identifiers. These identifiers are simply the names that are used to identify the variables, functions, macros, etc. Certain rules must be followed to create such identifiers. First character of identifier must be alphabet or underscore and no commas, blanks or special symbols (except underscore) are allowed in it. It is a good practice to give meaningful names as identifiers e.g. basicsal, m_hra, po_1235, create_node, etc.

Keywords

Keywords are the words treated specially by the compiler with their standard meaning. The keywords cannot be used as variable names. There are total 32 keywords in standard C language. There were only 27 keywords available in K & R version of C language. Another 5 keywords were added by ANSI during standardization. These 5 keywords are enum, void, const, volatile and signed. Few examples of keywords are : int, float, char, do, while, for, if, else, break, switch, case, default, etc. Note that few compilers also have some extra words reserved with certain meanings. For example turbo C compiler have reserved words like near, far, huge.

Data types

C program involves some computations. This needs data of different types to be stored in memory. Few languages do not support data types; they are referred as type less languages, e.g. B, BASIC, scripting languages. However, C language is typed language. Few built–in data types are given in the language. Also language allows defining new types if required. This chapter mainly focuses on primary types. Derived types and user defined types are discussed in later chapters. Figure 2.1 represents different data types.

Primary built in types

C language contains some keywords that work as modifiers to basic primary types given above. This

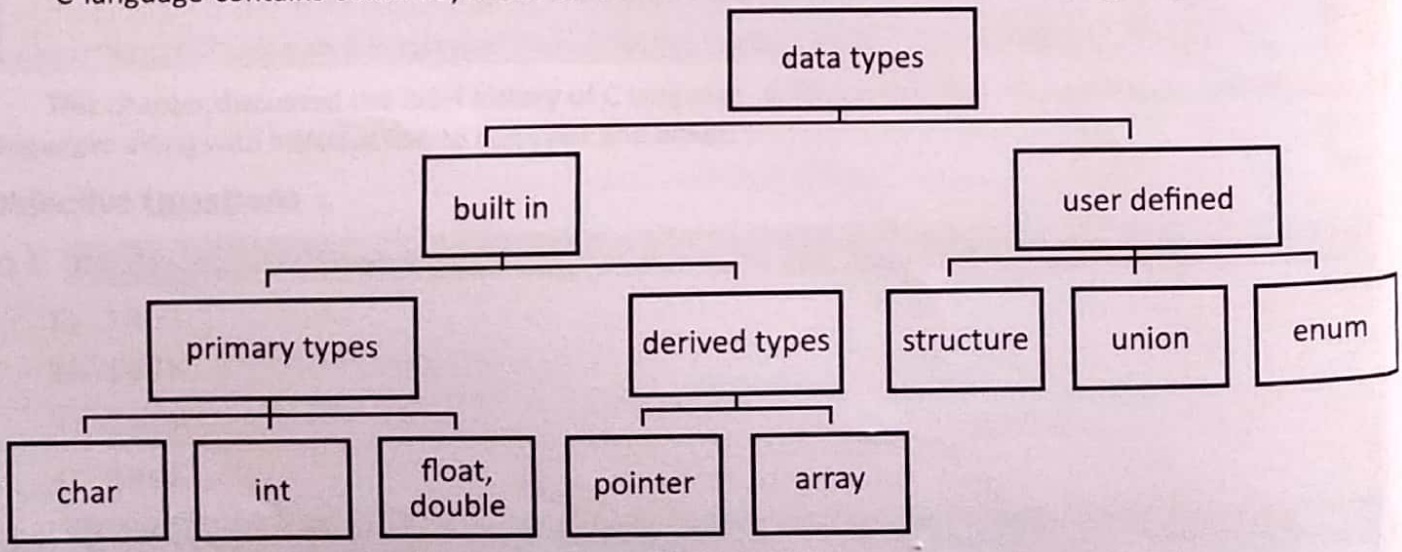


Figure 2.1 Data Types

Scanned by CamScanner

creates variety of data types in the language. The full chart of primary data types is given in Table 2.1

Basic Type	Modified Type	Size	Range
int	int	2 or 4	–32768 to +32767 or –2147483648 to +2147483647
	short int signed short int	2	–32768 to +32767
	unsigned short int	2	0 to 65535
	long int signed long int	4	–2147483648 to +2147483647
	unsigned long int	4	0 to 4294967295
char	signed char	1	–128 to +127
	unsigned char	1	0 to 255
float	float	4	3.4e–38 to 3.4e+38
double	double	8	1.7E–308 to 1.7E+308
	long double	10	3.4E–4932 to 1.1E+4932

Size of simple int data type varies from compiler to compiler. On 16-bit compiler its size is 2 bytes and it behaves like short e.g. Turbo C. On 32-bit Compiler its size is 4 bytes and it behaves like long e.g. GCC.
Also size of long double is 10 bytes according to ANSI C. But on some Compilers like Visual C++ Compiler, it can be found as 8 bytes.

Note that char range is given in integers. This indicates that characters are internally stored as integers. In fact every character is stored as some number in memory called as ASCII value of that character. For example character "A' has ASCII value 65. ASCII values for some characters can be found in table 2.2. The ASCII values for all the charterers can be found in ASCII chart given in Appendix A.

Table 2.2 : ASCII codes

Character	ASCII code
A – Z	65 – 90
a – z	97 – 122
0 – 9	48 – 57

Variables and Constants :

Using these data types we can declare the variables. Variable represents some memory location for a specific data type. Variables stores data of its own type i.e. integer variable can store only integer data. Typical syntax for declaring variable is as follows :

data_type variable_name; OR *data_type variable_name = value;*

For example :

```
int a;
double b = 2.3;
char c = 'A' ;
```

In the second example, *double* is data type, *b* is an identifier representing the variable and *2.3* is value stored in this variable. Value in this variable can be modified by doing some operations (like assigning a new value i.e. b=4.5 ;) on the variable, hence it is called as *variable*. In this declaration, during execution of the

Scanned by CamScanner

program, 8 bytes will be reserved in memory to store the data. Doing any operation on variable *b* will actually affect the contents of this location.

Note that we can assign some constant value to the variable or value within another variable. Constants are of different types like variables.

Integer constants :

Integer constants can be written in decimal, hexadecimal or octal number systems. Example : 23, –45, 0, +65, 0101, 0x41, etc. Using some suffix letter at the end of the integer constant can give some special meaning to it. Example : 12L represent long integer constant, 360U, 987u represents unsigned integer constants.

Floating point constants :

Floating point constants represent fractional numbers. They can be written in floating point format (e.g. 12.4, +0.0389, –2.76) or exponential form (e.g. 1.24e+1, +3.89e–2, –2.76e+0). By default floating point constant is considered of *double* type. Suffix *f* for the floating point constant will represent *float* constant (e.g. 12.4f).

Character constants :

Character constant represent a single character. They are always enclosed in single quotes. C compiler treats character constant as an integer of its ASCII value. For example character constant a is considered as 65. There are some special characters used in the program starting with \. They are called as escape sequences. List of these escape sequences is given in Table 2.3

Table 2.3 : Escape Sequence

Character	Name	Meaning or Use
\n	New line	Takes cursor on new line
\r	Carriage return	Takes cursor at the start of current line
\t	Tab	Takes cursor 8 characters ahead
\f	Form feed	Used in context of printer
\b	Backspace	Moves cursor one character back of current position
\a	Alarm	Gives a small beep
\'	Single quote	Prints single quote
\"	Double quote	Prints double quote
\\	Backslash	Prints backslash
\ooo	Octal ASCII	Prints character with octal ASCII value '\101' prints 'A' (octal 101 = decimal 65)
\xhh	Hex ASCII	Prints character with hex ASCII value '\x41' prints 'A' (hex 41 = decimal 65)

String constants :

One or more characters enclosed in double quotes forms a string constant. String constants are very often needed in program to print some messages. String constants may contain any characters or escape sequences. Example : "sunbeam infotech", "sunbeam\ninfotech", etc.

Scanned by CamScanner

Introduction to other data types

- Pointer is a derived data type that is used to keep the address of some memory location. This can be address of another variable.
- Array is a derived data type that is used to represent collection of the variables of the same data type. It is mainly used to represent a set of values of same type e.g. marks of different subjects, names of students, etc. Example : int a []={ 11,22,33,44} ;
- String is a derived type that is used to represent set of characters i.e. array of characters. For example `char c[] = "sunbeam" ;`
- Structure is a user defined type that is used to combine logically related data of different data types. For example : student information contains his roll number (int), name (string), average (double), etc.
- To store all this information together we may define a new data type struct student, whose variable can store information of one student.

```
struct student
{
    int roll;
    char name [20];
    double avg;
};
```

C Instructions

Any program is a set of instructions or statements. In this book, *statement* and *instruction* words are used alternatively. C program can have different types of instructions as follows :

Type declaration instructions :

Purpose of these instructions is to declare type of variables. The declaration statement must be at the start of the block before any executable statements. These statements give information to the compiler about the variables used. For example :

```
int basic_sal;
char name, code;
```

Input / Output instructions :

Purpose of these instructions is to take input from the user and display result on the console. The I/O instructions are given with the help of standard libraries. Consider the standard I/O statements or library functions.

printf() : standard output function used to print some set of characters or value of variables on console.

printf("<format string>", [list of variables]);

<format string> may be set of characters and/or format specifiers (place holders). *[list of variables]* is list of variables whose value is to be printed. For example :

```
printf("sum  : %d, avg  : %f", s, 23.4f);
```

scanf() : standard input function used to input values from user and assign to corresponding variables.

scanf("<format string>", [list of variables]);

<format string> is set of format specifiers only. *[list of variables]* is list of variables whose value is to be scanned. Each variable in this list must be preceded by &.

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

More details about using printf(), scanf() and their format specifiers are discussed in Appendix B.

Scanned by CamScanner

Arithmetic instructions :

Purpose of these instructions is to perform arithmetic operations between variables and constants. Arithmetic operations are performed using arithmetic and assignment operators. Generally, arithmetic instructions consists of variable name at the left of = operator and variables/constants on right hand side connected by operators like +, –, *, /, %. Variables and constants are called as operands and are operated by arithmetic operators. During execution expression on right hand side is evaluated and result is assigned to the variable at left hand side. For example

```
x = x+1;
si = prin * noy * roi / 100.0;
```

Arithmetic operators can be used with integers, real, characters. While doing operations on different data types, type of one of the variable is promoted to match with the other variable of higher type. The rules for this promotion can be given as follows :

- All short and character operands are converted to int.
- If one of the operand is long double, other is converted to long double and result will be long double.
- Else, if one of the operand is double, other is converted to double and result will be double.
- Else, if one of the operand is float, other is converted to float and result will be float.
- Else, if one of the operand is unsigned long int, other is converted to unsigned long int and result will be unsigned long int.
- Else, if one of the operand is long int, other is converted to long int and result will be long int.
- Else, if one of the operand is unsigned int, other is converted to unsigned int and result will be unsigned int.

Also remember that during assigning float to int, double to float or long to int may cause truncation of the data.

Type conversion can be done using explicit type casting. The syntax for explicit type conversion is :

(type–name) expression

Here, *expression* result will be converted into the type represented by *type–name*. The rules will be well understood with a very simple example are given in Table 2.4.

Table 2.4 : Type conversion examples

Operation	Remark
5 / 2 = 2	No conversion
5.0 / 2 = 2.5	2 converted to 2.0 (double)
5 / 2.0f = 2.5f	5 converted to 5.0f (float)
5.0f / 2.0 = 2.5	5.0f converted to 5.0 (double)
(double)5 / 2 = 2.5	5 is explicitly converted to double 2 is converted 2.0 (double)

Control instructions :

Purpose of these instructions is to control the sequence of execution of C statements. Control instructions are given using keywords like if, else, for, do, while, switch. These instructions will be explained in the next chapter.

Scanned by CamScanner

First C Program

After understanding fundamentals required for c program, now it's time write a couple of programs.

Program 2.1 : First C program

```
/* Simple C program */
#include <stdio.h>                                    /*Header file (s)  */
int main()              /*main function : statements are enclosed here*/
{
    printf("Sunbeam Infotech");                       /*Print message*/
    return 0;
}/*end of program*/
```

Program 2.1 starts with comment. Comments are enclosed within /* */. There can be any number of comments or there may not be any comment in the program. Comments are used to give extra information of the source code. Writing comments in the program is always a good programming practice. Note that comments can be multi line but cannot be nested. Most importantly text written in comments is never compiled; it is only for the programmer's reference.

Set of instructions in any C program must be enclosed within some function block. All C programs must have main function block. Program execution starts with the first instruction in the main function block. Last line returning zero indicate successful execution of the program. Any statement written after this will not be executed. More details about functions will be discussed in *Functions* chapter.

Finally note that every complete C statement must be ended with semicolon (;).

Second C program

After basic idea about the C program, now it is time for little advanced program, that takes input from the user and do some calculation and display the result (interest calculations).

Program 2.2 : Program to calculate Simple Interest

```
#include <stdio.h>
int main()
{
    float si, prin, roi, noy;                         /*type declaration*/
    printf("Enter Principle, Rate of Interest & Number of years..");
    scanf("%f%f%f", &prin, &roi, &noy);               /*input the values*/
    si = prin * roi * noy / 100;          /*formula for simple interest*/
    printf("Principle = %f\nSimple Interest = %f", prin, si); /*dis-
play results*/
    return 0;
}
```

Detailed steps for compiling and executing typical C program with different IDEs are given in appendix C and command line compilation is discussed in appendix D.

Operators

The arithmetic operators are already discussed in the previous topic. However, C has a variety of operators. Operators in C can be classified depending on number of operands or it can also be classified

Scanned by CamScanner

depending on its type/category.

Classification of operator based on number of operands

Unary : These operators need single operand. e.g. ++, --, unary +, unary –, etc.

Binary : These operators need two operands, e.g. +, –, *, /, &&, ||, <, >, etc.

Ternary : This operator (? :) needs three operands.

Classification of operator based on its category :

Arithmetic operators :

The arithmetic operators are +, –, *, % Along with these operators, there is one more arithmetic operator called as modulus % operator. This can be applied for integral data types and returns remainder. The sign of result is same as sign of numerator. For example :

```
23 % 5 = 3
-23 % 5 = -3.
```

Assignment and Shorthand operators :

As discussed earlier, assignment operator is used to assign value of an expression to some variable. Left hand side of assignment operator must be some variable, while right hand side can be any expression formed with variables and/or constants.

Shorthand operator is short representation of some operations. For example :

```
x += 3;
```

is same as

```
x = x + 3;
```

There are many shorthand operators like +=, –=, *=, /=, %=, &=, |=, etc.

Relational operators :

Relational operators are mainly used to compare two operands. There are 6 relational operators. They are <, >, <=, >=, ==, !=. The result of these operations can be 0 (if false) or 1 (if true).

Program 2.3: Relational Operators

```
#include <stdio.h>
int main()
{
    int a = 3, b = 6;
    int c = a < b;                  /*c will be 1; as a is less than b*/
    int d = a >= b;          /*d will be 0; as a is not greater than b*/
    printf("c=%d  d=%d", c, d);                    /*output: c=1  d=0*/
    return 0;
}
```

In program 2.3, value of c and d are 0 and 1 respectively; because relational operator can evaluate to 0 or 1. Also note that value of c is 1 as a < b condition is true and value of d is 0 as a >= b condition is false.

Logical operators :

Logical operators are mainly used to combine two conditions or to negate a condition. There are three

Scanned by CamScanner

logical operators. They are && (logical and), || (logical or), ! (logical not). The result of these operations can be 0 (if false) or 1 (if true).

Logical AND operation results false (zero) if any one condition is false (zero); It results true (one) if both conditions are true (non–zero).

Logical OR operation results true (one) if any one condition is true (non–zero); It results false (zero) if both conditions are false (zero).

Logical NOT operation results false (zero) if given condition is true; it results true (one) if given condition is false. It is unary operator.

Program 2.4 : Logical Operators

```
#include <stdio.h>
int main()
{
  int a = 3, b = 6, c = 9;
  int d = a < b && b < c;              /*d will be 1; as both conditions true*/
  int e = a < b || b > c;               /*d will be 1; as first condition true*/
  int f = ! (a > b);                     /*f will be 1; as condition is false*/
  printf("d=%d  e=%d  f=%d", d, e, f);                /*output: d=1  e=1  f=1*/
  return 0;
}
```

Bitwise operators :

Bitwise operators can be applied on integral types. They work on the binary bit pattern corresponding to that number. But obviously to understand working of bitwise operators, knowledge of number systems (i.e. decimal, binary, octal, hexadecimal) is must. Detailed discussion on number systems is out of scope for this book; However, some discussion on this topic can be found in Appendix E. Further discussion in this topic assumes that reader is aware of number systems.

Bitwise operators are heavily required for accessing system level features, where certain bits are used to represent certain functions. There are 6 bitwise operators. They are & (bitwise AND), | (bitwise OR), ~ (bitwise NOT), ^ (bitwise XOR), << (left shift), >> (right shift).

Program 2.5 : Bitwise Operators

```
#include <stdio.h>
int main()
{
  short a = 12, b = 5;
  short c = a & b;                      /*c will be 4*/
  short d = a | b;                     /*d will be 13*/
  short e = a ^ b;                      /*e will be 9*/
  short f = ~a;              /*f will be FFF3 (hex)*/
  short g = b << 2;                    /*g will be 20*/
  short h = b >> 1;                     /*h will be 2*/
  printf("c=%d  d=%d  e=%d  f=%x  g=%d  h=%d", c, d, e, f, g, h);
  return 0; /*Explanation for the output will be clear from the table 2.5*/
}
```


Scanned by CamScanner

Table 2.5 : Binary Operators example

Numbers	Binary Pattern	Explanation
12	0000 0000 0000 1100	In AND operation resultant bit is 1 if and only if both bits are 1; if any of the bit is 0, resultant bit will be 0.
& 5	0000 0000 0000 0101	
4	0000 0000 0000 0100	
12	0000 0000 0000 1100	In OR operation resultant bit is 0 if and if both bits are 0; if any of the bit is 1, resultant bit will be 1.
|5	0000 0000 0000 0101	
13	0000 0000 0000 1101	
12	0000 0000 0000 1100	In XOR operation resultant bit is 0 if and if both bits are 0; if any of the bit is 1, resultant bit will be 1.
^ 5	0000 0000 0000 0101	
9	0000 0000 0000 1101	
~ 12	0000 0000 0000 1100	In NOT operation, each bit is inverted i.e. 0 to 1 and 1 to 0.
FFF3	1111 1111 1111 0011	
5	0000 0000 0000 0101	In Left Shift operator each bit is shifted to left, and zero is padded to right side. Here shifting is done two times (<< 2).
<< 2	<< 2	
20	0000 0000 0001 0100	
5	0000 0000 0000 0101	In Right Shift operator each bit is shifted to right, and zero is padded to left side for unsigned numbers, while copy of right most bit (MSB) is padded to left side for signed numbers.
>> 1	>>1	
2	0000 0000 0000 0010	

Special operators

Other than these there are many operators in C language. They are used for different purposes. All these and above operators are listed in precedence and associativity Table 2.6. Details for most of these operators will be covered in respective chapters. The few operators are discussed here.

sizeof operator

sizeof (data-type)
sizeof (expression)

This is a unary operator. The operand of sizeof can be data–type or some variable or constant or expression. The first syntax gives the sizeof variable of the given *data type*, while second syntax gives the sizeof result of *expression*. It is evaluated during compilation only and returns size in bytes as unsigned int.

Program 2.6 : sizeof operator

```
#include <stdio.h>
int main()
{
    float a=2.3f;
    printf("%u %u %u", sizeof(char), sizeof(1.2), sizeof(a)); /*output: 1 8 4*/
    return 0;
}
```

Increment operator :

++ is the increment operator that increments the value of the variable by one. It is unary operator and

Scanned by CamScanner

its operand must be a variable (any constant or expression is not allowed).

```
int a = 4, b = 8;
++a;                                    /*pre increment form*/
b++;                                    /*post increment form*/
printf("a=%d b=%d", a, b);              /*Output : a=5 b=9 */
```

This operator has two flavors as follows :

pre–increment operator

Here value of the variable is first incremented and then used in expression (modified value will be used). Syntactically the operator will precede the operand.

post–increment operator

Here value of the variable is first used in expression (old value will be used) and then it is modified. Syntactically the operator will suffix the operand.

Program 2.7 : Increment operator

```
#include <stdio.h>
int main()
{
    int a = 4, b = 8, c, d;
    c = ++a;                                /*pre increment form*/
    d = b++;                                /*post increment form*/
    printf("a=%d  b=%d  c=%d, d=%d",
                a, b, c, d); /* Output: a=5  b=9  c=5  d=8*/
    /*note: c will get updated value of a, while d get old value of b*/
    return 0;
}
```

Decrement operator

-- is the decrement operator that decrements the value of the variable by one. Other properties of decrement operator are similar to increment operator.

Program 2.8 : decrement operator

```
#include <stdio.h>
int main()
{
   int a = 4, b = 8, c, d;
   c = --a;                                 /* pre decrement form */
   d = b--;                                 /* post decrement form */
   printf("a=%d  b=%d  c=%d", a, b, c, d);
                                            /* Output: a=3  b=7  c=3  d=8*/
   /*Note: c will get updated value of a, while d get old value of b */
   return 0;
}
```

Comma operator

Comma operator allows combining multiple expressions. All the expressions in the statement are evaluated from left to right order. The following example explains concept and syntax of comma operator.

Scanned by CamScanner

Program 2.9 : comma operator

```
#include <stdio.h>
int main()
{
      int a, b;
      a = (2,3);                              /*3 is assigned to a*/
      b = (5, ++a);                    /*++a i.e. 4 is assigned to b*/
      printf("a = %d b = %d", a, b);        /* output: a=4  b=4 */
      return 0;
}
```

Note that even though the each expression in the statement is evaluated, the right most value is assigned to the variable at left hand side of assignment operator.

Operator precedence and associativity

The following program explains how the expression will be evaluated if there is more than one operator are present :

Program 2.10 : Operator Precedence and associativity

```
#include <stdio.h>
int main()
{
    int a, b;
    a = 2 + 3 * 4; /*exprl: First multiplication and Then addition*/
    b = 5 - 6 + 7;    /*expr2: First subtraction and Then addition*/
    printf("a=%d  b=%d", a, b);                  /*output: a=14  b=6*/
    return 0;
}
```

C compiler evaluates these expressions by using a concept called precedence and associativity. In first expression, operators used are + and *. From the chart 2.6 it can be seen that precedence of * is higher than +. So * operator will bound by both of its operands i.e. 3 and 4. While + operator will bound by two operands i.e. 2 and result of multiplication. In second expression, operators used are + and –. From the chart it can be seen that precedence of + and – is same. In such case associativity is considered. Associativity of these operators is from left to right indicating that operator at left hand side will get preference. Thus – will be bound by both of its operands i.e. 5 and 6, while + will be bound by two operands i.e. result of subtraction and 7.

Table 2.6 : Operator Description precedence and associativity

opeator	description	precedence	associativity
()	function call	1	left–to–right
[]	array subscript	1	left–to–right
.	struct member access via variable name	1	left–to–right
→	struct member access via pointer name	1	left–to–right
++ —	increment/decrement	2	left–to–right
+ –	unary plus/minus	2	left–to–right
!	logical negation	2	left–to–right

Scanned by CamScanner

~	bitwise complement	2	right–to– left
(type)	cast (change *type*)	2	right–to– left
*	indirection	2	right–to– left
&	address of	2	right–to– left
sizeof	determine size in bytes	2	right–to– left
* / %	multiplication/division/modulus	3	left–to–right
+ –	addition/subtraction	4	left–to–right
<< >>	bitwise shift left, bitwise shift right	5	left–to–right
< <=	relational less than/less than or equal to	6	left–to–right
> >=	relational greater than/greater than or equal to	6	left–to–right
== !=	relational is equal to/is not equal to	7	left–to–right
&	bitwise and	8	left–to–right
^	bitwise exclusive or	9	left–to–right
|	bitwise inclusive or	10	left–to–right
&&	logical and	11	left–to–right
||	logical or	12	left–to–right
? :	ternary conditional	13	right–to–left
=	assignment	14	right–to–left
+= –=	addition/subtraction assignment	14	right–to–left
*= /=	multiplication/division assignment	14	right–to–left
%=	modulus	14	right–to–left
&= ^= |=	bitwise and/exclusive or/inclusive or assignment	14	right–to–left
<<= >>=	bitwise shift left/right assignment	14	right–to–left
,	comma (separate expressions)	15	left–to–right

Lab Exercise

Q. 1. Write a program to print your name, address, date of birth and qualification.

Q. 2. Write a program to accept two numbers and find its sum, difference, product and division.

Q. 3. Print the ASCII value of user entered character in decimal, hex, octal and character format.

Q. 4. Write a program to accept a 4 digit number and Display sum of all digits.

Q. 5. Write a program to convert temperature in Celsius to Fahrenheit and vice versa.

Formula for conversion is $°C = \frac{5}{9} \times (°F - 32)$

Scanned by CamScanner

Objective Questions

Q.1.

```
#include <stdio.h>
int main()
{
    char ch = 65;
    int x = 9;
    printf("%d, %d,", sizeof(ch), sizeof('A'));
    printf("%d, %d, ", sizeof(65), sizeof(++x));
    printf("%d", x);
    return 0;
}
```

1) 1, 1, 2, 2, 9
2) Compiler error
3) 1, 4, 4, 4, 9
4) 1, 1, 2, 2, 10

Q.2.

```
#include <stdio.h>
int main()
{
    printf("%d",  '\n' - '\r');
    return 0;
}
```

1) nr
2) -3
3) n-r
4) 3

Q.3.

```
#include <stdio.h>
int main()
{
    int x = 300;
    printf("%u", x * x / x);
    return 0;
}
```

1) 300
2) 81
3) 65235
4) +300

Q.4.

```
#include <stdio.h>
int main()
{
    int a = 3, b = 6;
    printf("%d, ", a = b);
    printf("%d, ", a == b);
    printf("%d, ", a != b);
    printf("%d, ", a=!b);
    return 0;
}
```

1) 6, 1, 0, 0
2) 0, 0, 1, 0
3) 0, 0, 1, 1
4) 3, 0, 1, 0

Q.5.

```
#include <stdio.h>
int main()
{
    int a = -10, b = 3, c = 0, d;
    d = a++ || ++b && c++;
    printf("%d, %d, %d, %d, ", a, b, c, d);
    a = -10, b = 3, c = 0;
    d = c++ && ++b || a++;
    printf("%d, %d, %d, %d", a, b, c, d);
    return 0;
}
```

1) -9, 3, 0, 1, -9, 3, 1, 1
2) -10, 4, 0, 1, 0, 4, 0, 1
3) -9, 4, 0, 1, 0, 4, 0, 1
4) -9, 4, 1, 2, -8, 3, 5, 1

Scanned by CamScanner

Q.6.
```
#include <stdio.h>
int main()
{
    int x = -1;
    printf("%u, %x, %d", x >> 1, x << 4,
    (unsigned)x >> 1);
    return 0;
}
```

1) 0fffc, -4
2) 4294967295, fffffff0, 2147483647
3) -1, ffff, -1
4) Compiler error

Q.7.
```
#include <stdio.h>
int main()
{
    char a = 255;
    char a = 127;
    b = ~b;
    a = a ^ b;
    printf("\n%d, %d" , a, b);
    return 0;
}
```

1) 127, 255
2) 255, 127
3) 127, -128
4) 127, 128

Q.8.
```
#include <stdio.h>
int main()
{
    printf("\n%d", 1 | 3 % 2);
    return 0;
}
```

1) 1
2) 3
3) 2
4) 6

Q.9
```
#include <stdio.h>
int main()
{
    int a = (1, 2, 3);
    int b = (++a, ++a, ++a);
    int c = (b++, b++, b++);
    printf("\n%d, %d, %d", a, b, c);
    return 0;
}
```

1) 3, 3, 3
2) 3, 6, 6
3) 6, 9, 8
4) 6, 6, 6

Q.10.
```
#include <stdio.h>
int main()
{
    printf("%d", ++4);
    return 0;
}
```

1) Compiler error
2) 4
3) 5
4) none of the above

Scanned by CamScanner

Chapter 3 : Control Structures in C

As discussed earlier, control instructions are used to control the flow of execution of the program. By default, program execution start with first statement in block of main() function and consecutive statements are executed till the end of the block. But many times there are situations, where the requirement is to execute certain set of instructions if some condition is satisfied, or to repeat some set of instructions again and again. This can be done by control instructions in C language. These instructions are basis of C programming, so one must be comfortable with understanding and using these instructions at appropriate place. There are five types of control instructions in C as follows :

Sequence control instructions :

Sequence control instructions ensure that the instructions are executed in the same order, in which they are written. This is default nature of program execution. All programs done till now, use sequential control instructions.

Decision control instructions :

Decision control instructions are used to take decision, which instruction is to be executed next based on some condition (whether it is true or false). They are implemented in C language using if, else keywords and conditional operator.

Case control instructions

Case control Instructions are used to select a particular option from a set of alternatives. It allows executing a certain set of statements out of many depending on some values.

Loop control instructions

Loop control instructions are used to execute a group of instructions number of times. The set of instructions will be repeated till the same condition is true.

Jump instructions

Jump instructions are used to jump from the current statement to some other statement in the block or function.

Decision control instructions

Decision control statements can be written using if, else keyword or conditional operators.

if statement

If the given *condition* is true, block of statements is executed, otherwise the statements within the block are skipped and the execution arrives from next statement after the block.

```
if (condition)
{
A block of statements.
}
```

Generally, the condition is expressed using relational operators and combined using logical operators as discussed in chapter 2.

To understand appropriate use of the if statement, consider one problem statement.

Scanned by CamScanner

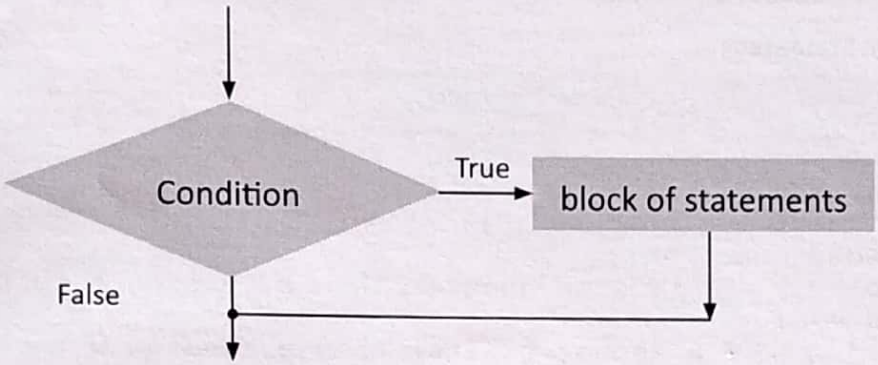


Figure 3.1 : if statement

Problem : Enter total sales made by salesman. If it is greater than Rs.5000, then give him commission of Rs.300. Assuming his basic salary of Rs.2500 and calculate the gross salary of salesman.

Program 3.1 : If statement

```
#include <stdio.h>
int main()
{
    float ts, bs=2500, com=0, gs;
    printf("Enter total sales..");
    scanf("%f", &ts);
    if(ts > 5000)
    {
        /*this block will execute if and only if ts is greater than
5000*/
    com = 300;
    }
    gs = bs + com;
    printf("commission = %f\n gross salary = %f", com, gs);
    return 0;
}
```

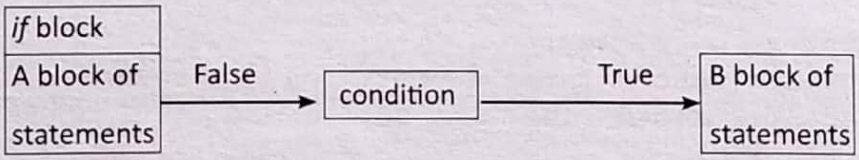
Note that, if there is a single statement in *if* block (or any other control structure block), writing curly braces is optional. However, it is good practice to write these curly braces along with proper indentation as shown, will improve the readability of the code.

if – else statement

```
if (condition)
{
   A block of statements.
}
else
{
   B Block of statements,
}
```

Figure 3.2 : If else Statement

If the given *condition* is true, block of Statements followed by *if* (A block) is executed, otherwise (if the given condition is not *true*) the block followed by *else* (B block) is executed. Note that *else* block should be immediate after



To understand appropriate use of the "if—else" statement, consider another problem statement.

Scanned by CamScanner

Problem : Print maximum of two numbers.

Program 3.2 : If else Statement

```
#include <stdio.h>
int main()
{
     int a, b, max;
     printf("Enter two numbers..");
     scanf("%d%d", &a, &b);
     if(a > b)       /*if a is greater than b, assign a to the max*/
          max = a;
     else       /* if a is not greater than b, assign b to the max*/
          max = b;
     printf("Max number: %d", max);
     return 0;
}
```

conditional operator

Syntax of conditional operator is : *(condition)? expression1 : expression2 ;*

If the given *condition* is true, *expression1* is evaluated, otherwise *expression2* is evaluated. Note tha writing statements instead of expressions that do not evaluate to any result will cause syntax error. Generall conditional operator appears to right side of assignment operator. So result of expression evaluated, is assigne to variable at left side of '='. Using this operator maximum of two numbers can be printed in program 3.3.

Program 3.3 : ternary operator

```
#include <stdio.h>
int main()
{
     int a, b, max;
     printf("Enter two numbers..");
     scanf("%d%d", &a, &b);
     max = (a>b  ? a  : b);
     printf("maximum =  %d", max);
     return 0;
}
```

It must be noted that, it is possible to have nested if statements, nested if–else statements and neste conditional operators. To illustrate this consider the following programs, which finds maximum of thre numbers :

Program 3.4 : nested if statements

```
#include <stdio.h>
int main()
{
     int a, b, c, max;
     printf("Enter three numbers..");
     scanf("%d%d%d", &a, &b, &c);
     if(a > b)
     {
          if(a>c)
```


Scanned by CamScanner

```
            max = a;
        else
            max = c;
    }
    else
    {
        if(b>c)
            max = b;
        else
            max = c;
    }
    printf("maximum = %d", max);
    return 0;
}
```

Program 3.5 : nested ternary operators

```
#include <stdio.h>
int main()
{
    int a, b, c, max;
    printf("Enter three numbers..");
    scanf("%d%d%d", &a, &b, &c);
    max = a>b ? ( a>c ? a : c ) : ( b>c ? b : c );
    printf("maximum = %d", max);
    return 0;
}
```

Logical Operators

Basic working of logical operators is already discussed in previous chapter. Working of logical NOT is extremely simple, as it simply inverts the condition. The program 3.6 shows an example of logical NOT.

Program 3.6 : Logical not operator

```
#include <stdio.h>
int main()
{
    int a=4, b=9;
    if(a<b)                    /*condition is true, so print "one"*/
        printf("one ");
    if(!(a<b))                    /*inverse of conditoin is false*/
        printf("two ");             /*so this line is not printed*/
    else
        printf("three");                /*but this line is printed*/
    printf("b=%d  !b=%d  !!b=%d", b, !b, !!b);
                    /*final output one three  b=9  !b=0  !!b=1*/
    return 0;
}
```


Scanned by CamScanner

The final output of the program is *one three b=9 !b=0 !!b=1*. Half of the output is explained in program itself with the help of comments. The last line is somewhat interesting. Remember that, logical operators always results in 0 or 1. On last line value of b will be printed as 9. !b will print 0, as it will invert non–zero (9) value to zero value. At the same time !!b will invert the output of !b. Thus !!b will result into 1.

To understand functionality of logical OR and logical AND operators, consider the following example :

Problem : input the year from the user and check whether it is leap year or not. Note that leap year is that year which is divisible by 4, but it should not be divisible by 100. In case if it is divisible by 100, it must be divisible by 400. e.g. 1996 is Leap year (divisible by 4), but 1997 is not (not divisible by 4). Also 2000 is leap year (divisible by 400), but 1900 is not Leap year (Not divisible by 400).

Program 3.7: logical operators

```
#include <stdio.h>
int main()
{
	int y;
	printf("enter year:");
	scanf("%d", &y);
	if( (y%4==0 && y%100!=0) || y%400==0)
		printf("leap year");
	else
		printf("not leap year");
	return 0;
}
```

Note that, in the program 3.7, three conditions are combined together using logical AND, logical OR operators. The first condition (y%4==0) checks for divisibility by 4. And second condition (y%100!=0) checks for non divisibility by 100. As per problem statement year is leap year if it is divisible by 4 and not by 100. When both the conditions are true, year will be the leap year (e.g. 1996). If any of the condition is false (e.g. 1997 is not divisible by four and 1900 is divisible by 100), the combination is considered false. Thus first operand for logical OR is false, so it will evaluate its second operand condition (y%400==0). If this condition is true, year is leap year (e.g. 2000). At first look, this might be confusing, but revising it one more time should make it clear.

Case control instructions

```
switch (integer expression)
{
	case constant1:
		1st set of instructions;
		break ;
	case constant2 :
		2nd set of instructions;
		break ;
	/*can be any number of cases*/
		default :
	nth set of instructions;
}
```

Many times there is need to execute some set of instructions depending on the option selected from the set of available choices. For this C have few keywords : switch, case, break and default.

Note that constant1, constant2, etc are called as case constants. They must be of integral type; floating point constant not allowed. As discussed earlier, *Integer expression* in syntax is any simple or complex arithmetic expression that evaluates to an integer value. The result of expression is compared with case constants. The set of instructions followed by matching case is executed. These instructions continue to execute till *break* is reached or end of switch block is reached. So writing break keyword after last case is not required. If the result of integer expression do not match with any of the constant, then the

Scanned by CamScanner

statements correspond by to *default* case are executed. However, the *default* case is optional. When the set of instructions within the matching case are executed, control comes out of switch and continues to execute the next statement after switch block. To understand application of switch statement, consider the following example.

Problem : Input the employee id and department number from the user and display the detailed information message. (For example : Employee with employee id 101 works in Marketing department.). Use the following department numbers : 10 (Marketing), 20 (Sales), 30 (Production), 40 (Accounts), Other (Unknown).

Program 3.8 : switch statement

```
#include <stdio.h>
int main()
{
     int empid, deptno;
     printf("enter employee id and dept number:");
     scanf("%d%d", &empid, &deptno);
     printf("Employee with employee id %d works in ", empid);
     switch(deptno)
     {
          case 10:
               printf("Marketing");
               break;
          case 20:
               printf("Sales");
               break;
          case 30:
               printf("Production");
               break;
          default;
               printf ("Unknown");
               break;
     }
     printf(" department.");
     return 0;
}
```

The switch statement is best suited to implement menu driven program. In a menu driven program, number of choices are offered to user, out of which any one is selected at a time. The code corresponding to each menu item is written under particular case. Using a loop the menu is repeated and particular task is done according to user's selection. Hence the program is called as *Menu driven Program.*

The switch statement works well for integral constants. This includes integer and character constants, as characters are internally stored as integers. The following program use character constants :

Program 3.9 : Switch statement

```
#include <stdio.h>
int main()
{
     char ch;
     printf("Enter a character:");
     ch = getchar();
     switch(ch)
     {
          case 'A':
```


Scanned by CamScanner

```
                printf("Apple");
                break;
            case 'B':
                printf("Baby");
                break;
            case 'C':
                printf("Cat");
                break;
        }
        return 0;
}
```

Jump instructions

C also has some instructions that can be used to take the control directly to some other statement For this four keywords are used. Those are goto, break, continue and return. The use of break with switc is already discussed. The break and continue is used context of loops. So it can be found in next topic. Th return keyword is used in context of functions, so it is discussed in chapter 4.

The *goto* keyword is used to take control to some label statement in that function. Labels can be defined an where inside the function block. The following example shows use of *goto* keyword in a menu driven program.

Program 3.10 : goto statement

```
#include <stdio.h>
int main()
{
    float num1, num2, result;
    int choice;
    START:    /*This is Label "START" */
    printf("Enter two numbers : ");
    scanf("%f%f", &num1, &num2);
    printf("\n1.Add\n2.Subtract\n3.Multiply\n Enter choice: ");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
            result = num1 + num2;
            printf(" Add : %f\n", result);
            break;
        case 2:
            result = num1 - num2;
            printf(" Subtract : %f\n", result);
            break;
        case 3:
            result = num1 * num2;
            printf(" Multiply : %f\n", result);
            break;
        default:
            printf("Invalid choice\n");
    }
    printf("\n0. Exit\n1. Repeat\n Enter choice:");
    scanf("%d", &choice);
    if(choice!=0)
        goto START; /*now control will jump to label and*/
            /*continue execution after that*/
    return 0;
}
```


Scanned by CamScanner

When *goto* is executed, control is transferred to label START and statements after that line are executed sequentially. Note that *goto* statement has some serious drawbacks. Using number of *goto* statements will make the program unreadable and difficult to debug. It is always preferred to use loops instead of *goto* keyword. The corresponding example can be found in the do–while loop topic. Still in certain situations, like deep nesting of loops, *goto* can be helpful.

Loop Control Instructions

Many times some action needs to be repeated again and again. For doing this mechanism topic introduced in C is called as *loop*. A certain set of instructions can be repeated for the specified number of times or until a particular condition is satisfied.

In C, three types of loops are supported :

- **while** loop
- **for** loop
- **do–while** loop

Before learning syntactical details of loops, first understand the basic concept of loop. Every loop has four important parts :

Body : Every loop has a body that contains the set of instructions to be repeated number of times.

Initialization : some variable like counter is initialized in initialization statement.

Test condition : Till this condition is true, loop will be repeated. If condition is false, execution of loop will be stopped and execution continues with next statement after the loop.

Modification : During multiple execution of loop, some of the variables like counter must be modified.

while loop

The situation where loop is to be repeated until any condition is false, while loop is most preferred. The working and syntax of while loop is given in Figure 3.3

```
Initialization,
While (condition)
{
body;
modification
}
```

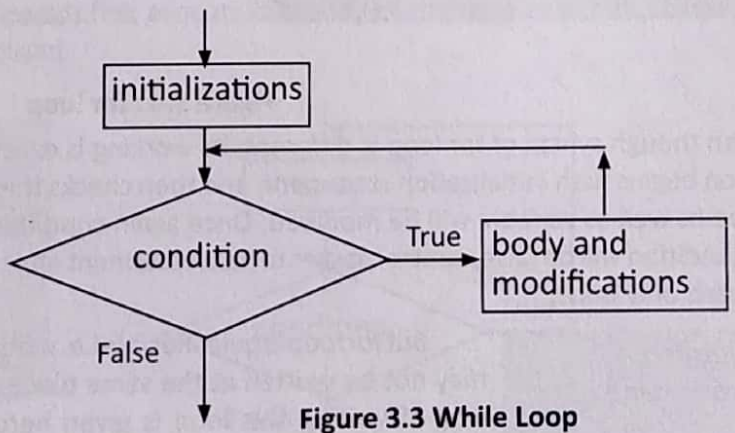


Figure 3.3 While Loop

Typically, execution begins with initialization statement, and then checks the condition. If condition is true, body will be executed as well as variable will be modified. Once again condition will be checked and process is repeated. When condition will be false, control passes to the next statement after the loop. Note that initialization statement is executed only once.

The following program will illustrate use and syntax of while loop :

Problem : Input a number from the user and find sum of all its digits.

Scanned by CamScanner

Program 3.11 : while loop

```
#include <stdio.h>
int main()
{
	int num, sum, digit;
	printf("Enter a number: ");
	scanf("%d", &num);                          /*initialization*/
	sum=0;
	while(num!= 0)                                /*condition*/
	{
		digit = num % 10;                   /*separate last digit*/
		sum = sum + digit;                        /*add the digit*/
		num = num / 10;                            /*modification*/
	}
	printf("sum : %d", sum); /*next statement after loop*/
	return 0;
}
```

for loop

The situation where loop is to be repeated fixed number of times, *for* loop is most preferred. The working and syntax of for loop is given in figure 3.4

```
for (initialization ; condition ; modification)
{
body;
}
```

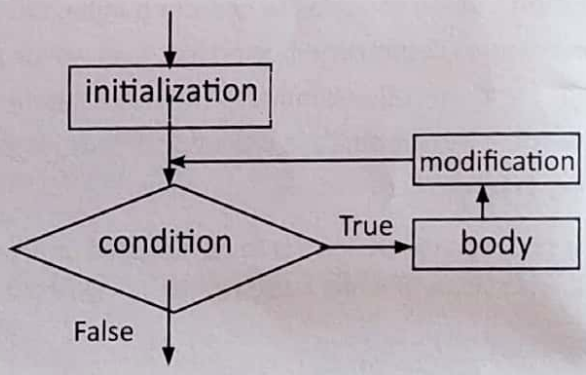


Figure 3.4 : for loop

Even though syntax of for loop is different, its working is exactly similar to the while loop. Just to revise, execution begins with initialization statement, and then checks the condition. If condition is true, body will be executed as well as variable will be modified. Once again condition will be checked and process is repeated. When condition will be false, control passes to next statement after the loop. Note that initialization statement is executed only once.

```
Initialization;
for (;condition;)
{
body;
modification;
}
```

But for loop is quiet flexible i.e. writing initialization, condition and modification may not be written at the same place as defined in the syntax. Another possible way of writing this loop is given here. Note that semicolons before and after condition is compulsory. This looks much similar to while loop.

Also initialization statement may have initialization of multiple variables separated by comma. And even modification statement can have modification of multiple variables separated by comma.

The example program 3.12 explains the concept and syntax of for loop. Also note that, even though syntax of for loop and while loop is different, their working is similar. So one can use these loops interchangeably i.e. for loop can be used instead of while loop and vice versa.

Problem : Print the table of the given number.

Scanned by CamScanner

Program 3.12 : for loop

```
#include <stdio.h>
int main()
{
	int a, num;
	printf("Enter a number: ");
	scanf("%d", &num);
	for (a=1; a<=10; a++)
	{
		printf("%d\n", a*num);
	}
	return 0;
}
```

Most importantly each loop must have condition and modification statement in such a way, so that condition will be false at some interation. otherwise, loop will continue to execute and referred as "infinite loop." Note that, no statement after this will be executed. Examples :

```
While (1) {
	/*body*/
}
```

```
a = 1;
While(a < 10) {
	/*body*/
}
```

```
for (; ;) {
	/*body*/
}
```

do–while loop

In for and while loops, we have seen that if condition is true, body is executed and continues to execute till condition is true. In case of do–while loop it first execute the body of the loop and then checks for the condition. If condition is true, body is repeated.

```
do
{
body of loop;
modification;
} while (test condition);
```

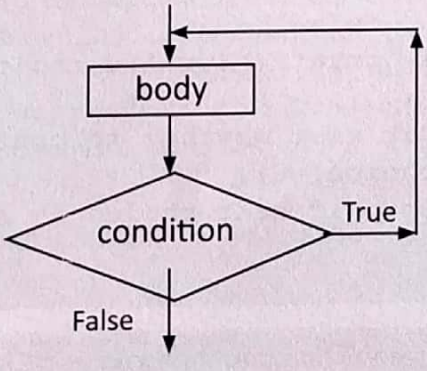


Figure 3.5 : do while loop

In do–while loop, initialization statement is optional, because value of counter can be modified at modification statement. The do–while loop is best suited for the programs, where body of loop should be executed at least once. Hence do–while loop is mostly used with menu driven programs instead of *goto* keyword shown in the program 3.10.

See the following program that shows implementing menu driven program using do–while loop.

Scanned by CamScanner

Program 3.13 : do–while loop

```
#include <stdio.h>
int main()
{
	float num1, num2, result;
	int choice;
	do
	{
		printf("Enter two numbers : ");
		scanf("%f%f",&num1, &num2);
		/*display menu and get choice from user*/
		pointf("\no.Exit");
		printf("\n1.Add\n2.Subtract\n3.Multiply\n Enter choice: ");
		scanf("%d", &choice);
		/*select the case depending on user choice*/
		switch(choice)
		{
			case 1: /*Add*/
				result = num1 + num2;
				printf(" Add : %f\n", result);
				break;
			case 2: /*Subtract*/
				result = num1 - num2;
				printf(" Subtract : %f\n", result);
				break;
			case 3: /*Multiply*/
				result = num1 * num2;
				printf(" Multiply : %f\n", result);
				break;
			default:
			printf("Invalid choice\n");
		}
		/*Ask user whether to continue for other operation*/
	}while(choice!=0);
	/*exit loop if user choice is zero*/
	return 0;
}
```

Till this time seen all the control structures in C language have been discussed. They are if, if-else, while, for, do-while, switch-case. In C, any control structure can be nested within any other control structure. C is also called "Structured programming language."

break statement

Many times there are situations where control needs to jump directly out of the loop, without waiting to get back to test the condition. The keyword *break* is used for such purpose. When keyword *break* is encountered inside any loop, control automatically passes to the first statement that follows body of loop.

Scanned by CamScanner

break is usually associated with an *if* as shown in the following example :

Problem : Write a program to determine whether a given number is prime or not.

Program 3.14 : break statement

```
#include <stdio.h>
int main()
{
  int num, n;
  printf("Enter number: ");
  scanf("%d", &num);
  for(n=2; n<num; n++) /* check divisibility for all numbers, from 2 to num-*/
  {
        if(num%n==0)     /*if number is divisible, it is not prime*/
        {
            printf("Not a prime");
            break;       /*if not prime, do not repeat the loop*/
        }
    }
    if(n==num) /*if not divisible by any number, num is prime*/
        printf("Prime");
    return 0;
  }
```

Note that break can be used only inside the loop or switch statement, it cannot be written in standalone if or else block.

continue statement

In some situations, there is need of immediately continuing next iteration of the loop bypassing the remaining statements in the body of the loop. The keyword, *continue* can be used for this purpose. When the keyword, *continue* is encountered inside any loop, control continues with next iteration of the loop. In case of while and do–while loop, continue statement takes the control to the test condition. But in case of for loop, continue keyword takes control to the modification statement and after executing modification statement, condition will be checked. If condition is true, body will execute again and if condition is false next statements after loop will be executed. A continue statement is usually associated with an *if* as shown in the following example :

Problem : All combinations of numbers 1, 2 and 3 so that in a combination any two numbers are not same.

Program 3.15 : continue statement

```
#include <stdio.h>
int main()
{
    int a, b, c;
    for(a=1; a<=3; a++)
    {
        for(b=1; b<=3; b++)
```


Scanned by CamScanner

```
        {
            for(c=1; c<=3; c++)
            {
                if(a==b || b==c || c==a)
                    continue;
                printf("%d %d %d\n", a, b, c);
            }
        }
    }
    return 0;
}
```

The output of the program will be :

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Please note that, when break and continue is used in nested loops, they affect the loop in which they are placed. It means that, break takes control out of the loop in which it is placed (i.e. it do not take control out of all loops) and continue repeats the iteration of the loop in which it is placed.

Lab Exercise

Q. 1. Write a program to determine whether given number is odd or even using if else statement as well as ternary operator.

Q. 2. Write a program to display number of days in the given month and year using if, else if statement and || operator and && operator

Q. 3. Write a program to accept a character c and display category of the input character.

Q. 4. DIGIT : c is a digit (0 to 9)

Q. 5. LOWERCASE : c is a lowercase letter (a to z)

Q. 6. UPPERCASE : c is an uppercase letter (A to Z)

Q. 7. Write a program to accept Employee Id, Department No, Designation from user and display output with reference to following tables

Dept No.	Dept. Name	Dsgn Code	Designation
10	Marketing	'M'	Manager
20	Management	'S'	Supervisor
30	Sales	's'	Security Officer
40	Designing	'C'	Clerk

Example : If input given is

Employee Id : 101

Dept No : 30

Designation Code : M

Then output should be :

Employee with employee id 101 is working in "Sales" department as "Manager" .

Scanned by CamScanner

Q. 5. Write a program to display n terms of Fibonacci series .

Q. 6. Write a program to accept a number and Calculate sum of digits.

Q. 7. Write a program to print prime numbers between 1 and 100.

Q. 8 . Write a program to print following pattern.

```
      1 1
    2 1 1 2
  3 2 1 1 2 3
4 3 2 1 1 2 3 4
```

Objective Questions

Q.1.

```
#include <stdio.h>
int main()
{
    int a =3, b = 6;
    if (a = b)
        printf("%d, %d", a, b);
    else
        printf("%d != %d", a, b);
    return 0;
}
```

1) 6, 6
2) 3, 6
3) a ! = b
4) 3 !, = 6

Q.2.

```
#include <stdio.h>
int main()
{
    int a =3, b = 6;
    printf("%d", a!=b?0?b:a:b);
    return 0;
}
```

1) 6
2) Compiler error
3) 3
4) 0

Q.3.

```
#include <stdio.h>
int main()
{
    int a =3;
    while(a--){
        printf("%d, ", a);
    }
    return 0;
}
```

1) Compiler error
2) 3, 2, 1,
3) 2, 1,
4) 2, 1, 0,

Q.4.

```
#include <stdio.h>
int main()
{
    int a = 1000000L;
    for(a=1; a<10; ++a);
    {
        printf("%d, ", a);
    }
    return 0;
}
```

1) 1, 2, 3....., 0
2) 10,
3) 11,
4) Compiler error

Scanned by CamScanner

Q. 5.

```
#include <stdio.h>
int main()
{
	char ch;
	while(ch <256)
	{
		printf("%d, ", ch);
	}
	return 0;
}
```

1) 1, 2, 3..., 255,
2) infinite loop
3) Compiler error
4) Garbage Value

Q. 6.

```
#include <stdio.h>
int main()
{
	char i = 0;
	do{
		printf("%d, ", i);
	}while(--i);
	return 0;
}
```

1) 0, -1..., -128,127...1,
2) infinite loop
3) none
4) 0,

Q. 7.

```
#include <stdio.h>
int main()
{
	char i = 0;
	do
	{
		printf("%d, ", i);
	}while(i--);
	return 0;
}
```

1) 0,-1...-128, 127...1,
2) infinite loop
3) none
4) 0,

Q. 8.

```
#include <stdio.h>
int main(){
	int i, j;
	for(i = 1; i <= 3; i+=2)
		for(j = 1; j <= 3; j+=2)
			printf("%d, ", i+j);
	return 0;
}
```

1) 5, 1, 5, 3,
2) 2, 4, 6, 8,
3) 2, 4, 4, 6,
4) 2, 3, 4, 5,

Scanned by CamScanner

Chapter 4 : Functions

Function is a block of statements designed for some specific task. Function may take some input (called as arguments), process it and may give some output (called as return value). In this sense, function is much similar to the program. In fact, C program is a collection of functions. Hence function is also called as subprogram. Function is also referred as routine or method in different contexts.

C program must have minimum one function, that is main(). Program execution begins from main(), so main is also called as entry point function. Till this point all programs in this book have only main() function. However, splitting the code into multiple functions allows using modular approach while programming. This approach is really useful for larger programs, where multiple programmers work on the same project. The program source code must be divided into logically isolated tasks and each task should be accomplished in a separate function. This makes the program readable, easy to fix the bugs and maintenance. Also reusing the functions by creating their library will reduce development time significantly. Using user defined function library or built in C function library is a very common programming practice.

> Keep the habit of writing code into functions right from learning days. It will improve the readability of the program and will clear the concept of reusability. The continual use of functions will help to improve thinking process in an appropriate direction, which is necessary for any C programmer.

Function Basics

For understanding functions three fundamental terminologies must be understood. These three terminologies are :

- Function declaration (also called as function prototype or signature)
- Function definition (also called as implementation or function body)
- Function call

Program 4.1 Function Declaration, definition, call

```
#include <stdio.h>
int factorial(int   n); /*function declaration*/
int main()
{
    int   num, result;
    printf("enter a number: ");
    scanf("%d",  &num);
    result = factorial(num);          /*function call*/
    printf("%d\n", result);
    return 0;
}
int factorial (int n)             /*function definition */
{
    int i, res = 1;
    for (i=1; i<=n; i++)
        res = res * i;
    return res;
}
```


Scanned by CamScanner

The program 4.1 provides an introduction to these terminologies. The important terminologies from this program are explained below.

Function declaration

Function declaration is written at the top of source file or in the declaration section of other function where it will be called. To avoid declaration in every function; it is much better to declare it at the top. The function declaration like a variable declaration, informs the compiler about the name, arguments and return type of the function. The syntax of function prototype is given as :

return–type function–name(argument–data–type arg1–name, argument–data–type arg2–name, ...);

return–type of the function represents the data type of the value to be returned from the function. In this example, result of factorial is returned which is taken as integer. According to ANSI standard, if there is no need of returning any value from the function, its return type must be given as void. The void return type indicates that no value will be returned from the function. If return type is not specified, by default it is considered as int. However, it is best programming practice to specify the necessary return type, to improve readability and avoid unexpected errors.

function–name is any valid C identifier, but it is expected to use the name, that will clear the purpose of the function. In above example, function is used to calculate the factorial of a given number, hence the name is given as *factorial.*

Argument is the input values given to the function for the processing. Function may have multiple arguments or no arguments. The declaration must mention data types for each argument. These data types of arguments must be separated by commas. As a good practice, declaration may also have some argument name, to make the declaration more understandable. In above example, there is a single argument *n* of type *int*. According to ANSI standard, if there is no argument for the function, it should be written as void. Table 4.1 gives some examples of function declarations.

Table : 4.1 Function declarations

Declaration	Description
int fact(int n);	accept an int argument and return an int value.
void print(char ch);	accept a char value and do not return any value.
float sum(float a, float b);	accept two float arguments and return a float value.
void task(void);	do not accept any argument and do not return any value.
int fun();	returns an int value, and may take any arguments (specified in definition).

Most importantly, function declaration must be ended with as semicolon.

Function definition

As discussed earlier, function is a block of statements. These statements are written inside function definition. Syntax for writing function is given as :

return–type function–name(data–type arg1, data–type arg2, ...)

{

*/*set of statements*/*

Scanned by CamScanner

```
return some–value;      /*if return–type is not void*/
}
```

Function definition starts with writing function declaration; However, writing argument names are compulsory and there is no semicolon after declaration. Actual set of instructions are written within curly braces after declaration syntax as shown above. These instructions can be control instructions, arithmetic instructions, type declaration instructions or jump instructions as required in the source code of that function. Note that, function definition cannot be enclosed in function definition of another function.

Many times, function returns some value to the calling function using return keyword followed by the value to be returned. The return statement should be the last statement in function body. In any function, there can be more than one return statements, but please note that when any of the return statement is executed control returns back to the calling function and hence any statement after that return statement will not be executed. Any function cannot return more than one values. If there is no need to return any value from the function, return statement without any value can be written (i.e. return;) or return statement can be omitted from the function. Also in this case return type must be void, as mentioned earlier.

One more thing that should be understood while programming functions is that, any variable or argument of one function cannot be accessed directly into other function. The value of the variable can be passed to function while calling function (discussed in next topic), but cannot be directly modified into the function. Thus variables *num* and *result* is accessible only in main(), while variables *i, res* and argument *n* is accessible only in factorial(). Trying to access them outside their scopes will cause a compile time error. More discussion on life and scope of variables is done into next chapter.

Function Call

Function definition contains actual code for the function, while function declaration gives information of the function to the compiler before it is defined. Every C program execution begins from main() and it is ended when all statements in main() are executed. Thus, any other function will not execute, unless it is called from main() or from any other function that is called from main(). Syntax for calling function is given as :

variable–name =function–name(arg1, arg2, ...);

Function is called within any other function, by its name and passing the necessary values followed by semicolon. The values passed to the function call within parenthesis are called as parameters or actual arguments, as opposed to that arguments declared in function definition are called as arguments or formal arguments. Note that values within actual arguments are copied into formal arguments. In the factorial & example, actual argument is num and it is copied to formal argument n.

As a rule, one must remember that function always returns back to the place from where it is called. Thus when all statements in the body of the function are finished or return statement is executed, the next instruction after that function call will be executed. If function is returning some value, it can be assigned to some variable as shown in the syntax. In above example, return value of factorial function is collected in result variable. However, it is not mandatory to collect return value of the function. In some typical cases, the return value may be ignored. e.g. printf() function.

Scanned by CamScanner

Flow of execution

To illustrate some of the concepts related to functions and to understand the flow of execution, see the program 4.2. The example program 4.2 is created by doing simple modifications in the previous example 4.1. While discussing function declaration, it was mentioned that declaration can be done within the function (local declaration) in which it has to be called or can be done outside the function on the top of source code file (global declaration). The program 4.2 shows examples for both kind of declaration.

Program 4.2 Flow of execution

```
01   #include <stdio.h>
02   void print(int a);          /*function declaration (global)*/
03   int main()
04   {
05   int num, result;
06   int factorial(int n);       /*function declaration (local) */
07   printf("enter a number: ");
08   scanf("%d", &num);
09   result = factorial(num);                  /*function call*/
10   print(result);
11   return 0;
12   }
13   /*function to calculate factorial*/
14   int factorial(int n)                    /*function definition*/
15   {
16   int i, res = 1;
17   for(i=1; i<=n; i++)
18        res = res * i;
19   print(res);
20   return res;
21   }
22   /*function to print a value*/
23   void print(int a)                       /*function definition*/
24   {
25   printf("refult : %d\n", a);
26   }
```

This example is introducing the extra function and it is called from main() as well as factorial() function just for making the program bit complicated. Function print() is used to display some integer value, so there is no need to return any value from the function. Hence return type of the function can be seen as void. Now the program execution will go as follows :

- Program execution begins from main() function [Line 03].
- The declaration statements for variables and factorial() function is used for informing compiler about them [Line 05 and 06].
- Then printf() function is called and display the message on screen [Line 07].
- The scanf() function waits for user input and when number is entered, that value is stored in num variable [Line 08].
- Now factorial() function is called, with parameter num [Line 09]. The control will be shifted to factorial() function [Line 14] and the value of actual argument num, will be copied to formal

Scanned by CamScanner

argument n.

- The body of factorial() function will be executed to calculate factorial of n [Line 16, 17 and 18].
- Then print() function is called, with parameter res [Line 19]. The control will be shifted to print() [Line 23] and value of actual argument res will be copied to formal argument a.
- The body of print() will be executed [Line 25].
- The print() function is finished [Line 26], so control will be shifted back to the calling function i.e. factorial() [Line 19] and code after the call statement will be executed.
- The result of factorial is returned back to the calling function [Line 20]. Control will be shifted back to the calling function i.e. main() [Line 09] and return value of the function is assigned to the variable result and code after this will be continued.
- The print() function is called, with parameter result [Line 10]. The control will be shifted to print() [Line 23] and value of actual argument result will be copied to formal argument a.
- The body of print() will be executed [Line 25].
- The print() function is finished [Line 26], so control will be shifted back to the calling function i.e. main() [Line 10] and code after the call statement will be executed.
- Now main() function returns 0 [Line 11] and then program will be terminated.

Learning the entire flow will help to understand functions very well. In fact one must understand every line of code written into the program in order to make any kind of modification or bug fixing into that code.

Types of functions

The functions are of two types i.e. User defined functions and Library functions. User defined functions are defined according to program logic, while library functions are pre–defined and their library is also standardized as part of C compiler software.

User defined functions

User defined functions are the functions defined by the programmer. In above example, factorial() is a user defined function. These functions are designed by the programmer as required for certain program. In certain cases, programmer may design a set of commonly used functions and reuse them for different applications.

The design of user defined functions may change from programmer to programmer. However, some guidelines should be followed for writing better programs :

- The source code must be divided into logically isolated tasks. The function must be designed keeping some goal in mind. The goal can be as simple as printing some data, calculating certain formulae or doing some algorithm.
- The arguments and return values must be decided appropriately, so that function can handle certain variations in the process and also its return value can be properly used in the program.
- While writing functions, they should be made reusable. Once function is written it can be called from main() or any other function as per the requirement.
- To assure reusability of the functions, it is better practice to avoid console input output into the function. All the necessary input can be passed as arguments (instead of scanning it from user in the function) and its result can be returned from the function (instead of printing on the console in the function). This really helps while porting the program on the different platforms, as input output facilities can be changed on different technologies.

Scanned by CamScanner

Library functions

The library of pre–defined C functions is available as part of any C compiler. All these functions are declared in header files (.h) like stdio.h, string.h, etc. The definitions of these functions are given in form of library files (.lib or .so). How these function definitions are attached to the source code and how it is executed as a single unit is discussed into the next chapter. Discussing all functions is standard C library is really difficult. However, the most commonly used function declarations are given in Appendix G.

main() function

main() is just like any other user–defined function, However, program execution begins from main(). When program is executed, operating system gives call to main() function of the program. Since function is called by operating system, it must have some specific prototype. The standard declaration of main is given as :

```
int main(int argc, char *argv[]);
```

This declaration of main() is mainly used to take some information, when program run from command prompt or shell. The more details about these arguments are discussed in Chapter 7. However, some other prototypes of main are also widely used. They are given as :

```
int main(void);
int main();
```

According to ANSI standard return type of main() must be int. Since this if called by the operating system, the return value from the function is given to the operating system. The different return value has a different meaning for the operating system. There are three values can be returned from the main(). The return value 0 is most common, which indicates that the program has been executed successfully. The value 1 returned by the program, when program has to be terminated due to lack of some resource like some file or memory. In chapter 10, the main() returns 1 indicating the failure while opening file. The value –1 is returned by the main() to indicate abnormal program termination. This is not generally written with the return statement; in fact the program returns –1 internally in case of abnormal termination. Finally few non–standard implimentations allow return type as void instead of int. This declaration looks as :

```
void main ();          OR          void main (int argc, char *argv[]);
```

In these cases, main() do not return any value, so by default return value to operating system is considered as 0.

Nested function Calls

Even though we cannot write definition of one function into another function, but one function call can be nested into another function call. The typical example is :

```
printf("%d", printf("sunbeam"));
```

This will print sunbeam7. Note that, inner printf() call will be executed first and print *sunbeam*. printf() always returns number of characters printed on the screen. This value is passed to outer printf() call and hence it will print 7.

Pointers

Every memory location is identified by a unique positive integer called memory address. A pointer is a variable that stores the address of some memory location. Pointer is also called as *address variable*.

Scanned by CamScanner

The following diagram helps to explain the concept of pointer :

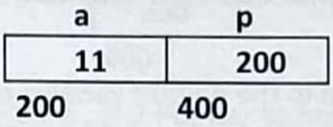


Figure : 4.1 Pointers

As shown in Figure 4.1, *a* is an int variable at the address 200 and store value11. Assuming size of *int* is 4 bytes, location 200 to 203 will be occupied by variable *a*. Also, *p* is a variable at the address 400 and store the address 200. So it can be said that, *p* is a pointer to *a*.

The concept of pointer can be best understood by visualizing it. It is always best practice to draw the diagrams, to understand how variables are stored in memory and how pointers keep their addresses. However, addresses of the variables may change with different compilers. Still to understand the concept, some addresses are taken arbitrarily in every diagram and program. Running the programs given in this book may not print the same addresses. Always remember that, the address will be given to variables and pointers by the compiler and operating system and one cannot modify them within the program.

The program 4.3 will illustrate the concept of pointers :

Program 4.3 Introduction to Pointers

```
#include <stdio.h>
int main()
{
  int a = 11;                    /*declaring and initializing variable*/
  int *p;                                    /*declaring a pointer*/
  printf("a=%d  &a=%u\n", a, &a);           /*outputs  a=11  &a=200*/
  p = &a;                                    /*initializing pointer*/
  printf("p=%u  &p=%u  *p=%d\n", p, &p, *p); /*Outputs p=200  &p=400  *p=11*/
  return 0;
}
```

Pointer declaration

Pointer can keep the address of some variable, However, to keep the address of different type of variable pointer should be declared of different type. The typical syntax for declaring pointer is :

*data–type *variable–name;*

The above example declares p as an integer pointer. The other type of pointers can be declared similarly.

```
char *p1, *p2;
```

This declares two pointers *p1* and *p2* of char type. Note that pointer stores the address of variable, which is an unsigned integer. So the size of pointer variable is equal to size of unsigned int. Also it is same for all pointers irrespective of its data type; because even though size required for value is changed, size of address remains same. The size of pointer is compiler dependent like size of int.

Pointer initialization

Declaration of any pointer is just like declaration of a variable. The pointer variable must be initialized with some valid address. It should be assigned to the address of the variable before it is used further in

Scanned by CamScanner

the program. The address of any operator can be obtained by using address of operator (&). It is also referred as direction operator or reference operator. Thus address of variable a can be accessed as &a.

```
p = &a;
```

This statement assigns address of *a* to the pointer variable *p*. Note that, address of integer variable must be stored in an integer pointer. Trying to store address of one type of variable in other type of pointer can cause warning or error message by the compiler. However, this message can be eliminated by using casting operator. But this feature should not be used without any special reason.

Dereferencing pointer

Once address of variable is assigned to the pointer, value of that variable can be accessed using that pointer. This can be done using value at operator (*). It is also referred as indirection operator or dereference operator.

```
printf("p=%u &p=%u *p=%d\n", p, &p, *p) ;
```

In this statement, *&p* represents the address of *p* variable (which is 400), *p* represents address stored in the *p* (i.e. address of a which is 200) and **p* represents the value at the address stored in *p* (i.e. value of *a* which is 11).

It is always observed, that new learners have some kind of fear with the word pointer. Actually pointers are really simple like anything else in C. However, few good habits into learning days can help to conquer pointers too easily.

The very first thing is that, one should be able to see the pointers. In other words, pointers must be visualized by drawing appropriate diagrams. The required number of example of diagrams are available in this book. So try to learn drawing the diagrams for the example programs and objective questions in this book on your own.

The second thing is that, one should be able to read the pointers. In other words, pointer syntax must be read properly to understand the meaning of it. Most of the times it is seen that, & and * are pronounced as ampersand and star; instead it must be strictly pronounced as "address of" and "value at"with the understanding of its meaning as discussed earlier.

If this practice is followed from the early days of learning it will not be so difficult. In fact, pointers will be found as a lovable topic in C.

Program 4.4 takes knowledge of pointer to one step ahead. It is just extension of the previous program, where address of pointer p is kept into another pointer variable pp.

Program 4.4 Pointer to Pointers

```
#include <stdio.h>
int main()
{
  int a = 11;                          /*declaring and initializing variable*/
  int *p;                                          /*declaring a pointer*/
  int **pp;                                /*declaring pointer to pointer*/
  printf("%d, %u\n", a, &a);                                  /*11, 200*/
  p = &a;                                        /*initializing pointer*/
  printf("%u, %u, %u\n", p, &p, *p);                        /*200, 400, 11*/
  pp = &p;                                 /*initializing pointer to pointer*/
  printf("%u, %u, %u, %d\n", pp, &pp, *pp,  **pp);      /*400, 600, 200, 11*/
  return 0;
}
```


Scanned by CamScanner

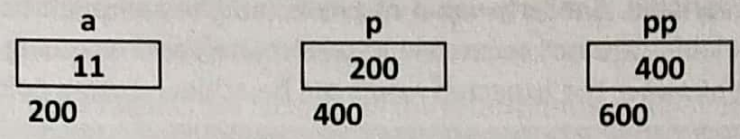


Figure : 4.2 Pointer to pointer

In this example, pp is pointer to pointer. Its declaration and initialization is much similar to the simple pointer. However, note that &pp represents address of pp variable (which is 600), pp represents address stored in pp (i.e. address of p which is 400), *pp represents the value at the address stored in pp (i.e. address stored in p which is 200) and **pp represents the value at the address represented by *pp (i.e. value of a which is 11). The level of indirection for p pointer is 1, while for pp it is 2.

This knowledge of pointers should build the basic foundation for the pointers. The more knowledge of the pointer, specially using the pointers will be there in subsequent chapters.

Call by value and Call by address

This concept can be well understood by an example. Also the following program will explain one of the uses of pointers.

Problem : Write a function calculate() to calculate sum and product of two integers passed as parameters. The result must be printed in main ().

Program 4.5 Call by value vs. Call by address

```
#include <stdio.h>
void calculate(int a, in b, int* psum, int* ppro); /*function declaration*/
int main()
{
     int num1, num2, sum, pro;
     printf("enter two numbers: ");
     scanf("%d%d", &num1, &num2);
     calculate(num1, num2, &sum, &pro);              /*function call*/
     printf("\n Sum = %d\n Product = %d", sum, pro);
     return 0;
}
void calculate(int a, int b, int *psum, int *ppro)/*function definition*/
{
     *psum = a + b;          /*value at the address in psum = a + b;*/
     *ppro = a * b;          /*value at the address in ppro = a * b;*/
}
```

As discussed before, more than one value cannot be returned to the calling function. But somehow values of sum and pro variables in main() should be modified to store the result of calculations. While calling the function calculate() values of num1, num2 and addresses of sum and pro are passed. The values of num1 and num2 are copied into formal arguments a and b respectively; also addresses of sum and pro are collected into formal arguments psum and ppro. Obviously to hold these addresses arguments must be pointers. Passing the value of the variable, is referred as *Call by value*; while passing the address of the variable, is referred as *Call by address* (or *call by reference*). Also note that, value of a + b is assigned to value at the address stored in psum (i.e. address of sum variable), which indirectly

Scanned by CamScanner

modifies value of sum variable. Similarly value of pro is modified through pointer ppro. Thus even though sum and pro variables are not accessible into calculate() and returning more than one values from the function is not allowed, the expected result can be achieved using pointers.

Call by value	Call by address
The variable i.e. its value is passed as the argument to the function.	The address of the variable is passed as argument to the function.
The actual argument is copied into formal argument.	The address of the actual argument is copied into formal argument.
The actual argument cannot be modified into the called function.	The actual argument can be modified into the called function.

After learning the difference between, one can understand the reason behind using address of operator with every argument of scanf() function. In fact, scanf() takes the address of the variable and then modify it into the function by taking its value from the user.

> In many discussions, passing the address of the variables is called as Call by reference. Purposefully here the term Call by address is used to avoid the confusion with concept of references in C++ .
>
> Going a step ahead, if above example is studied properly, it can noticed that value of num1 is copied into a, while address of sum is copied into psum. In that sense, both are simply call by value. In first case value of num1 i.e. an int value is copied and in second case address of sum i.e. address value (unsigned int) is copied. The concept call by address is included in the book only because it is commonly visited topic in many texts.

Recursion

As discussed earlier, any function can call itself. Further function can call itself if required. Such a function is called as recursive function and this feature is called as recursion.

The most important thing one should understand is when to use recursion. Using the recursion anywhere will cause the program complex as well as inefficient. Recursion is best suited for the cases, where some process can be explained in terms of itself. The following table tries to explain few examples for recursion :

Table : 4.2 Recursion formulae

Concept	Formula	Condition
Factorial	$n! = n * (n-1)!$	n>=1 [0! = 1]
Powers	$a^b = a*a^{(b-1)}$	b> = 1 [a^0 = 1]
Fibonacci	$t_n = t_{n-1} + t_{n-2}$	n>=3 [t_1 = 1, t_2=1]

Consider the first example, where factorial of any number is expressed in terms of factorial of other number. This is perfect situation for the factorial.

Recursive function has call to itself. So when such function is called, the control will keep rotating into multiple calls of the same function. While writing such function care should be taken so that at certain time recursive execution of the function is stopped and control returns back to the calling function. Thus appropriate end condition must be written. Many times this end condition is defined by the process itself possibly as part of mathematical formula. For recursive factorial, one must remember

Scanned by CamScanner

that 0! = 1. If end condition is missing or condition is given wrong, then program would terminate abnormally. The more discussion on the termination is done next chapter.

The following program calculates the factorial of the given number. The first example In this chapter also calculates the factorial. But program 4.6 uses recursion instead of loop.

Program 4.6 Recursion

```
#include <stdio.h>
int fact(int n);                          /*function declaration*/
int main()
{
     int num, result;
     printf("enter a number: ");
     scanf("%d", &num);
     result = fact(num);                          /*function call*/
     printf("%d\n", result);
     return 0;
}
/*function to calculate factorial*/
int fact(int n)                           /*function definition*/
{
     int r;
     if(n<=o)                                     /*end condition*/
          return 1;                                      /*0! = 1*/
     r = n * fact(n-1);          /*recursive call as per formula*/
     return r;
}
```

Writing a recursive program is always easy if the recursive formula and the end condition is well defined. The above implementation of fact() function illustrate this fact. Even though writing the function is easier, understanding the flow of execution is bit tedious due to recursive calls. Assuming that number scanned is 3, the calculation of factorial of 3, will be done as shown in the diagram,

Recursion works much similar to substitution method of mathematics. The mathematics says :

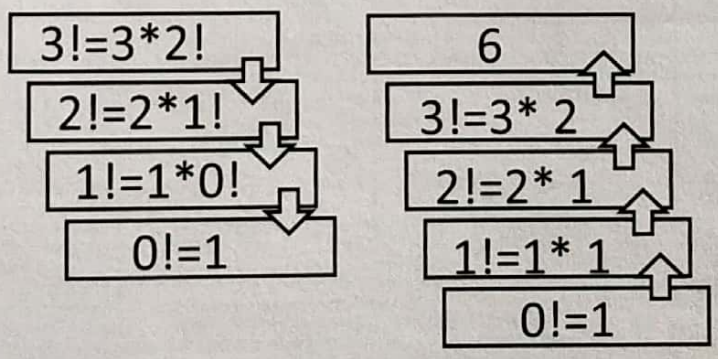


Figure : 4.3 Back substitution

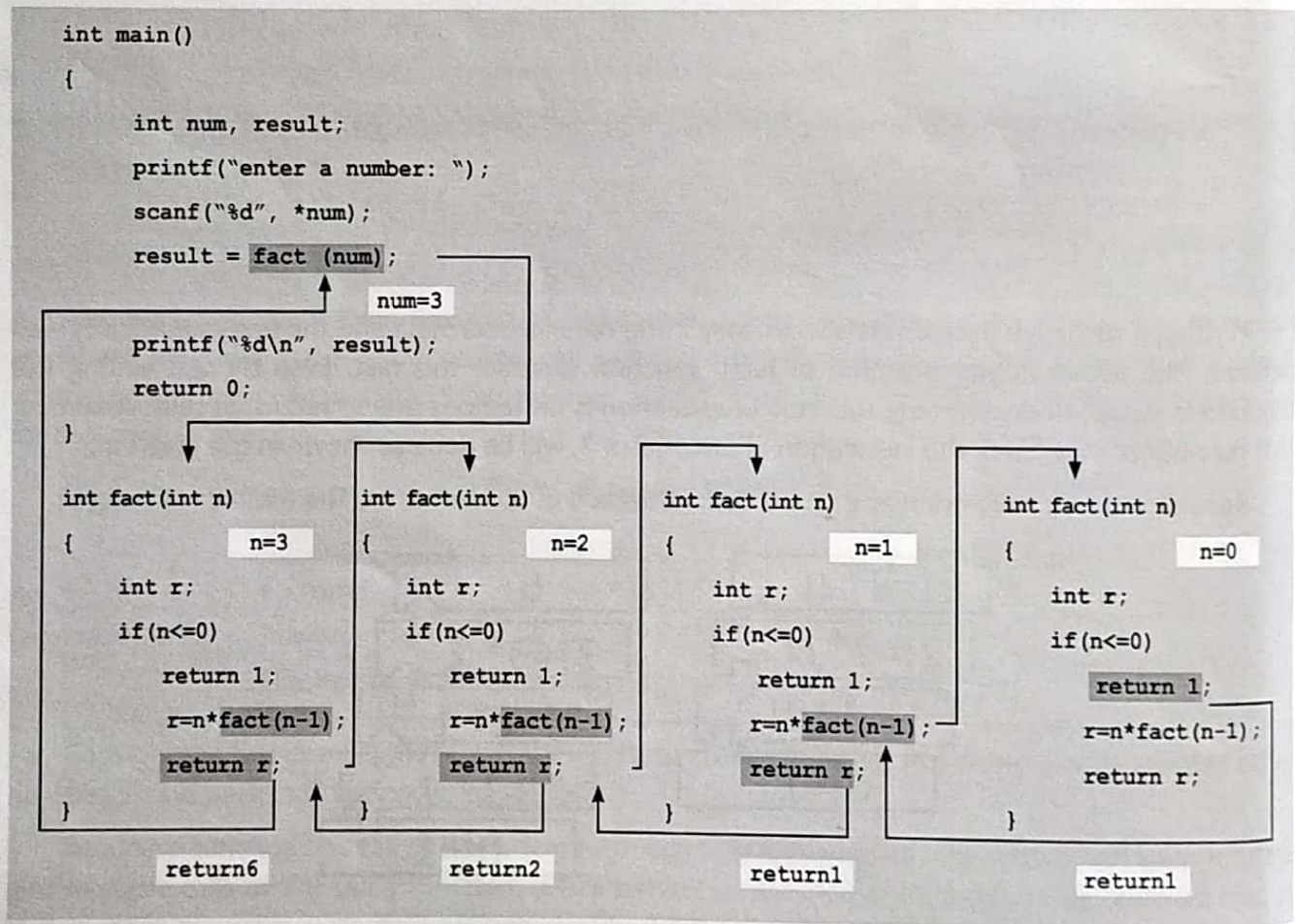
And the substituting the result of each into the previous step starting from the last step will give the result. Recursion works in exactly same manner.

Scanned by CamScanner

The Figure 4.4 should give enough idea about how the recursive function is executed. However, looking above program may give Impression that recursion should be used instead of loops. This misconception can be dangerous. Recursion must be used only for the processes that can be explained in terms of that process.

The recursion is mainly used to simplify the program. If first job of finding recursive process and its end condition is done, writing recursive program can be too easy. However, recursion also has some serious drawbacks. Having multiple calls to the function makes recursion much slower as compared to a loop. Also with every function call some data has to be stored on the stack (more details are discussed in next chapter) and so recursive calls needs much more memory compared to a loop. In that sense, above example of factorial is not that efficient as compared to loop version of the same program. It is always advised to avoid recursion if possible.

Till this time a lot of topics related to functions are covered. But still there is much more remaining. Anyhow, function is most fundamental concept in C language. For learning some more advanced details, few concepts related to operating systems must be known. This includes knowledge of process, data section, code section, stack section, compiler and linker etc. Even this knowledge is also required for another important topic in C language called as *storage classes*. All these details along with storage classes are discussed in next chapter.



Program 4.4 : Depth of Recursion

Scanned by CamScanner

Lab Exercise

Q. 1. Write a function drawline() for drawing a given character given number of times. Use this function to draw following output. In the chart given below, character * is printed 35 times using drawline().

```
**********************************
Characters          ASCII
**********************************
A–Z                 65–90
a–z                 97–122
0–9                 48–57
**********************************
```

Q .2. Write a function to calculate combination nCr. The formula for nCr is given as : nCr = n! / r! * (n–r)!

Q .3. Call this function from main() and print the result in main().

Q. 4. Write a function to calculate the power (a, b). Call this function from main() and print the result in main(). Use recursion.

Q. 5. Write a function to calculate Fibonacci series of given number of terms. Use recursion. fibonacci series is : 1 1 2 3 5 8 13....

Q. 6. Write a single function to check if a given number is positive or negative as well as even / odd. Call this function from main() and print the result in main().

Q. 7. Write a function to swap two given numbers. Call this function from main() and print the result in main().

Objective Questions

Q. 1. During function call _____ arguments are copied into _____ arguments.

1) primary, secondary
2) secondary, primary
3) formal, actual
4) actual, formal

Q. 2. _____ is entry point function for a c program

1) main ()
2) winMain ()
3) E32Main ()
4) Main ()

Q. 3. _____ of library function is kept in header files and their _____ are kept in library files.

1) definition, declaration
2) declaration, definition
3) call, prototype
4) prototype, call

Scanned by CamScanner

Q. 4. Which of the statements are true? (multiple choice)

1) Function cannot return multiple values.
2) Function cannot take multiple arguments.
3) Function can take multiple arguments.
4) Function can take multiple values.

Q. 5.

```
#include <stdio.h>
int main()
{
	int main = 49;
	printf("%.01f", sqrt(main));
	return 0;
}Bate moruderissum ublicae cutus pulut
```

1) 7
2) 7.0
3) garbage
4) compiler error

Q. 6.

```
#include <stdio.h>
int main()
{
	int a = 1;
	printf("%d", ++a);
	main();
	return 0;
}
```

1) 11111 ...
2) 22222 ...
3) compiler error
4) runtime error

Q. 7.

```
#include <stdio.h>
int main()
{
	int a = 1;
	int *p = &a;
	int *q = p;
	*p = *p + *q;
	printf("%d%d%d",*p,a,*q);
	return 0;
}
```

1) 111
2) 222
3) 211
4) 221

Q. 8.

```
#include <stdio.h>
int sunbeam()
{
	return a*a;
}
int main()
{
	int a = 3;
	printf("%d", sunbeam());
	return 0;
}
```

1) 9
2) garbage
3) compiler error
4) runtime error

Scanned by CamScanner

Q. 9.

```
void sunbeam(int *p )
{

     ++*p;
     *p++;
}
int main()
{
     int a=4;
     sunbeam(&a);
     printf("value =%d",a);
}
```

1) 1
2) 5
3) 43
4) 34
5) 6

Q. 10.

```
#include <stdio.h>
int sunbeam(int **q)
{
     int a = /*$$$*/;
     return a;
}
int main()
{
     int a = 3;
     int *p = &a;
     a=sunbeam(&p);
     printf("%d", a);
     return 0;
}
/*What code should be at $$$ to print 9?*/
```

1) a * a
2) *p * *p
3) *q * *q
4) **q * **q

Scanned by CamScanner

Chapter 5 : Storage Classes

In C language, variable declaration can be accompanied with the storage class for that variable. As name suggest, storage class gives information about where the memory for that variable will be allocated. But obviously to learn about the storage classes, one must understand the memory organization for the program during its execution.

All chapters till now discuss about writing source code of the program. Now it is time to see how program is executed. Actual journey from source code to its execution includes lot of stages. Giving full details of each stage will be unaffordable in this book; still sufficient information on each stage is collected here. The entire process can be broadly split into two parts. First part explains how source code is converted to executable code (build process) and the other part explains how executable code is executed on the operating system.

Steps for compilation

Since computer can understand only machine code, C source code or any other source cannot be directly executed. So any source code must be converted to the machine code for its execution. This task is done with the help of a special program called as compiler. This information is already given in the chapter 1. However, compilation is not the only step, rather there is series of operations done on the source code. The following diagram shows all the operations done on the source code till it is converted to executable code. The discussion done here is specific to C program and with little or no variations may be applied to source code of any other language.

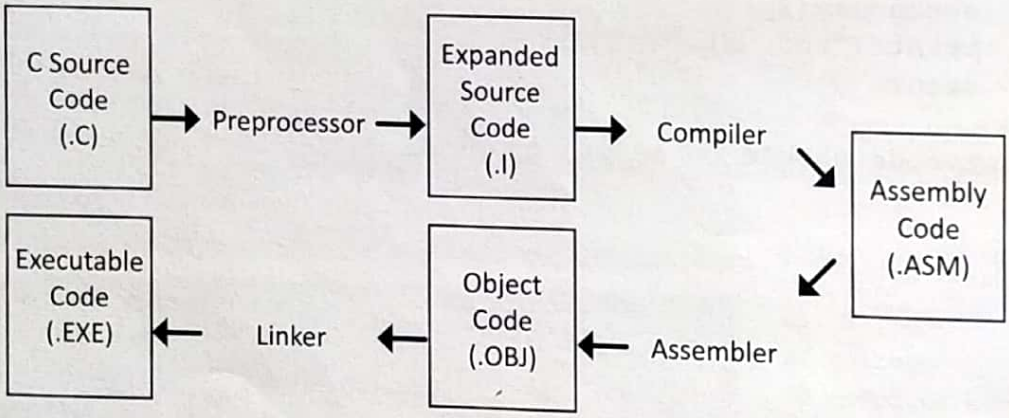


Figure 5.1 : Steps of Compilation

C Source code

C Source code is written into a text file with the extension. C (e.g. SUNBEAM.C). Typical C source code starts from inclusion of header files, some global declarations, implementation of functions including main() and optionally some informative comments.

Preprocessor

Preprocessor is a program that processes the source code before execution. The statements in the program starting with # are called as preprocessor directives and these statements are mainly executed by the preprocessor. Preprocessor mainly expands the source code according to the

Scanned by CamScanner

statements #define, #include, etc. So the output of the preprocessor is referred as expanded source code. Preprocessor may remove the comments from the program and may add the information of line number and filename.

Expanded Source Code

The source code is expanded by the preprocessor according to the given instructions. After expansion this expanded source code may be saved in a temporary text file. On the compiler like Turbo C, extension of this file is .I (e.g. SUNBEAM.I). This extension may change from compiler to compiler or in compilers like GCC (LINUX C compiler), may create this file in memory and pass to the next stage. This file is deleted automatically during compilation. However, this file can be viewed by using options in command line compilation (Appendix B).

Note that expanded source code is a simple text file contains expanded code along with information on line number and file name. Since the comments are removed by the preprocessor, those are absent in this file. In short, this file contains all that code and information which will be compiled by the compiler.

Compiler

Compiler is a program that translates the expanded source code to the assembly language code. Compiler is a huge topic and compiler design is an altogether a new subject. Still the main functions of the compiler can be listed as :

Compiler checks for syntax errors in the source code. If there is any error, compiler reports it and further processing is not done.

- Compiler can also find some possible mistakes in the code and warn about them e.g. uninitialized local variables, type mismatch while assignment, etc.
- Compiler converts the source code into assembly language code and saves it into some temporary text file with extension .ASM (e.g. SUNBEAM.ASM). This extension may change compiler to compiler.
- Compiler links the function calls to the function definitions, provided functions are defined in the same source file. The similar action is taken for the global variables in that file.
- Compiler assigns some relative addresses to the functions and global variables. These addresses can be considered as offsets within output file.
- Compiler creates the symbol table that includes information of all global variables and functions in that file.

> Compiler shows errors as well as warnings. Errors mainly occur due to syntactical mistakes and hence such program is not allowed to be executed. But warnings occur due to possible logical mistakes and such program is allowed to be executed even though it may give unexpected results. However, it is best practice to execute error free and warning free code. The warnings in the program can be disastrous specially while working at embedded systems or some system level programming. So never ignore the warnings read them carefully; understand them and remove them by doing appropriate modifications in the source code.

Assembly Code

Assembly code is generated by the compiler after compiling the error free source code. Assembly code is generally saved in the temporary text file and will be deleted in consecutive stages. It is the well known fact that, assembly language is depends on the processor architecture. So while writing

Scanned by CamScanner

compilers it must be targeted for specific processor architecture. For example, Turbo C compiler is targeted for 8086 compiler and using some options on command line or IDE it can be used for 80186 and 80286. Another example, Visual C compiler of Microsoft is targeted for 80386.

The compiler can generate assembly code for the processor for which it is designed. In most of the cases, compilers are designed for some minimum machine configuration. Now assembly code generated for this processor can work for all higher but compatible processors. For example, assembly code of 8086 processor can be executed on 80186, 80286 and other compatible processors. One should note that, executing 8086 assembly on the higher version processors cannot use the efficient processing supported by these processors, but can be executed to yield results.

Assembler

Assembler is a program that translates assembly code into machine level object code. Assembler converts op codes in assembly language to the corresponding numbers in the machine language.

In some cases, assembler can be found as a built in part of compiler program. Such compilers do the job of compilation as well as job of assembler. Assemblers generate relocatable object code.

Object Code

The output of assembler is an object code. Assembly code is saved into the text file, but the object code is saved into the binary file. The extension of object code file is .OBJ (e.g. SUNBEAM.OBJ). However, on GCC compiler it will be .o (e.g. SUNBEAM.o). This file contains the code that can be directly executed by the processor. Since this file is directly derived from assembly code, it contains all those things which are present in the assembly code but in binary (i.e. machine understandable) format.

The object code file has some specific format. This format is well known as Common Object File Format. The file is logically divided into multiple sections. The short description for these sections is given below :

Table 5.1 : Sections in Object File

Section	Description
Header	This section mainly contains the information about all the sections in this file. This information includes name of the section, its starting offset and its size.
Text Section	This section contains machine level code corresponding to the source code in .C file. This section is also referred as Code Section. Thus one can conclude that all functions in source file are stored in this section.
Data Section	This section contains all global and static variables and there initial values in the source file. However, uninitialized global variables are put into a separate section called as BSS Section (Block Started by Symbol).
Symbol Table	This section contains the information about the global, static variables and functions in this file. This information includes size, data type / return type and location of the symbol (variable or function). This information is mainly used while debugging the program. It can be kept minimum to disable debugging feature.

.OBJ file has relocatable object code. The relocatable means, the program is ready except the addresses for the functions and variables are not finalized. These addresses can be considered as simple offsets in this file, while the actual addresses can be given at runtime as per memory availability.

Scanned by CamScanner

Even though object file has machine level code, it cannot be directly executed because of few reasons :

- The definition of each function called in a file may not be available in the same file. Typically definitions of library function are kept in separated library files (.LIB on Windows and .so on Linux). This may include functions in other source files too.
- The machine level code can be understood by the processor, but this format may not be recognized by the operating system for the execution. The operating systems expected, certain format for the executable files as discussed later in this chapter.

Linker

Linker is a program that combines object code of source code files and object code of library files and creates final executable file. Till now all the programs have written in a single file. When program become complex and multiple people work on the same project, it is mandatory to split the entire code into multiple files. Obviously function written in one file can be accessed into other file. The following example shows a very simple program that is split into three files.

Program 5.1 : Multi File Program

```
/*main.c*/
#include <stdio.h>
#include "fact.h"
int main()
{
  int num, result;
  printf("enter num:");
  scanf("%d", &num);
  result = fact(num);
  printf("%d\n", result);
  return 0;
}
```

```
/*fact.c*/
int fact(int n)
{
  int res=1, i;
  for(i=1; i<=n; i++)
    res = res * a;
  return res;
}
```

```
/*fact.h*/
int fact(int n);
```

This is the same program that was discussed in previous chapter. It can be executed using any of the C compilers on command line or IDE. Every IDE for C programming has support for creating the project that can contains more than one source file. The information for creating multi file project in Visual C++ 6.0 IDE can be found in Appendix A.

The main functions of linker can be listed as :

- Linker creates final executable file in the appropriate format for targeted operating system.
- Linker links the definition of all external functions to their calls. The external function can be function defined in other source file of the same project or some library function. This thing is also applicable for global variables.
- Linker combines the data section of all object code files and library files into a single data section of executable file.
- Linker combines the text section of all object code files and library files into a single text section of executable file.
- Linker adds executable header to executable file that contain the information about the executable.

Any kind of spelling mistake can prevent the linker from finding that function or global variable

Scanned by CamScanner

into other object code file or library files. In this case, linker reports that error and do not create the executable file. Such error is referred as linker error.

The linkage specification tells how the symbol i.e. variable or function used in some statement is linked to its definition. Every function and even variables can have three kinds of linkage specification.

Linkage	Description
No Linkage	Typically local variables do not have any linkage specification.
Internal Linkage	The global variables and functions can be given as internal linkage. It indicates that this variable or function cannot be accessed outside that source file. Even linking of these variables and functions are done at the compile time as the call and definition present in the same file. This linkage can be specified using static keyword before the declaration of function or global variable.
External Linkage	The global variables and functions have by default external linkage. It indicates that this variable or function can be accessed outside that source file. Even linking of these variables and functions are done by the linker, if the call and definition present in different file. This linkage can be specified using extern keyword before the declaration of function or global variable.

Executable Code

This is final output of the complete build process discussed till now. Executable code is generated by the linker as discussed earlier and then saved in an executable file with extension .EXE (e.g. SUNBEAM.EXE). On Linux platform the extension can be .out or even there may not be any extension. The executable code must be in the format that can be understood by the operating system.

On Windows operating system executable file format is known as *Portable Executable* (PE) and on Linux operating system it is known as *Executable and Linking Format* (ELF). The file is logically divided into multiple sections. The short description for these sections is given below :

Table 5.2 : Sections in Executable File

Section	Description
Exe Header	This section mainly contains the information about all the sections in this file. This information includes name of the section, its starting offset and its size. This section contains the type of executable file (Console application or Graphical User Interface application or Library). This section also contains *magic number* that is a specific number representing the operating system on which the program can be executed. Magic number will be different for Windows and Linux. Most importantly this section includes address of entry point function i.e. main().
Text Section	This section contains machine level code. This section is also referred as *Code Section.* It is created by combining text section of all object code files and library files.
Data Section	This section contains all global and static variables and there initial values. It is created by combining data section of all object code files and library files. Same thing is true for BSS section.
Symbol Table	This section contains the information about the global, static variables and functions in this file. It is created from combining symbol tables of all object files in that project.

Scanned by CamScanner

The complete build process is discussed till now. Now the executable file is created from the source code. This file can be executed on the processor by the operating system.

The many topics in this chapter use the terminologies from the operating systems. This is unavoidable; because C language is designed from the operating system and its programs are again targeted to the operating systems. The detailed information of operating system concepts is out of scope for this book; However, minimum required information about the term is given when that term is introduced.

Steps for Execution

C source code is converted to executable code by the C compiler software using its elements like preprocessor, compiler and linker and this executable image is stored on secondary storage device, typically hard disk. This executable can be run from command prompt (Windows) or shell (Linux). Also in the Windows or Linux explorer one can double click the exe icon to run it. The steps for execution are shown in diagram 5.1.

Loader

When program is tried to be executed, the shell of the operating system initiates the process. Simplest definition of the process is said as program in execution. Operating system allocates the memory for the process into the RAM. The program cannot run on the hard disk, it should be loaded into the primary memory i.e. RAM so that it can executed with proper speed. This job is done by one of the module of an operating system called as loader. Loader loads the executable image into the memory allocated for the process into the RAM. During this loader first checks the file format of the executable. If file is not in the format that is required for the operating system, program is not loaded. This is simple reason why windows executable cannot be executed on Linux and vice versa.

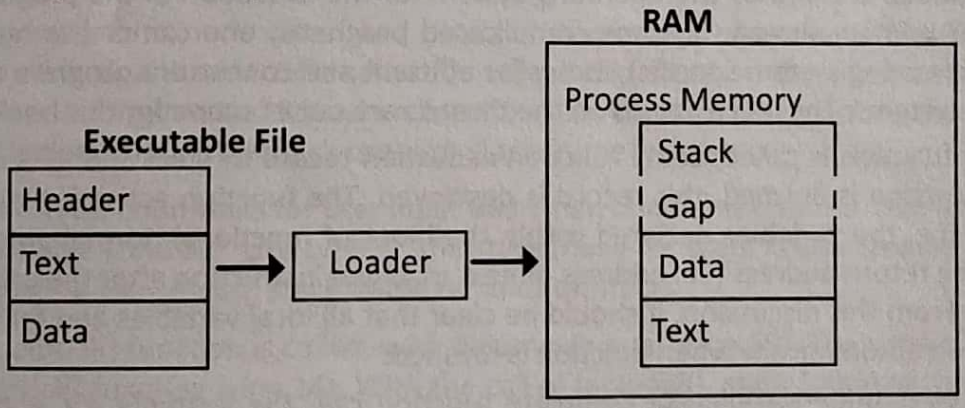


Figure 5.2 : Execution of C Program

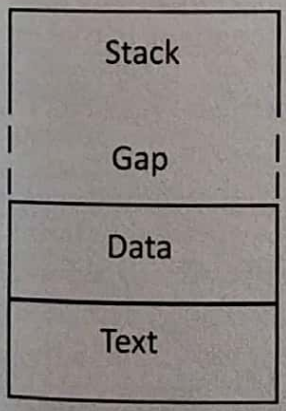


Figure 5.3 : Process Memorry

As discussed earlier, the executable image has some sections. Even the memory allocated for the process can be logically divided into multiple sections. The following diagram can explain the structure of process memory in the RAM.

The memory for the process is divided into four main sections :

Text Section

This section contains the binary machine level code loaded from the text section of executable image. All user–defined and library function definitions are kept here for execution. The

Scanned by CamScanner

machine level instructions to be executed by the processor are passed from this section.

Data Section

This section contains the all global and static variables along with their initial values. It is simply copied from data section of executable image. Similarly BSS section is loaded into the memory. In the diagram BSS section is considered as part of data section and hence not displayed separately.

Stack Section

This section is most important from program execution point of view. The text and data section is simply copied from the executable image, but the stack section is created at run time as a part of process memory.

Stack is a utility data structure, means that it is mainly used to process the data rather than storing it. It has LIFO nature i.e. the lastly added element is removed first. The stack section in the process memory has the similar behavior. The stack initially empty with function calls it grows in size towards data section. The *gap* between stack section and data section (shown in the figure 5.3) is the space to grow for the stack. The more details about the stack section are discussed ahead.

Function Calls

To be precise, operating system do not only create a process and allocate the memory to it for keeping text, data and all resources required for execution, but it also creates a thread for the execution of the program. The thread is defined as lightweight process and is responsible for the execution of the program. In fact, the stack section belongs to the thread and gets space into process memory. The thread created by the operating system for the execution of the program is called as main thread or primary thread. In some complicated programs, one can create multiple threads using native (operating system specific) library for efficient and concurrent program execution on a uni–processor systems. The more details on the threads are out of scope for this book.

When any function is called, some function activation record or stack frame is created on the stack. When function is finished, this record is destroyed. The function activation record contains, local variables (i.e. the variables declared within the block of functions), formal arguments of that function and the return address (i.e. address of next machine instruction after the call instruction of that function). From this discussion, it should be clear that all local variables and formal arguments will be destroyed automatically when function is finished.

For the proper understanding of complete function call, the example 4.2 is repeated again including the some more details.

Program 5.2 : Function Calls

```
1    #include <stdio.h>
2    void print(int a);                  /*function declaration (global)*/
3    int main()
4    {
5         int num, result;
6         int factorial(int n);          /*function declaration (local)*/
7         printf("enter a number: ");
8         scanf("%d", &num);
9         result = factorial(num);                    /*function call*/
10        print(result);
```


Scanned by CamScanner

```
11          return 0;
12      }
13      /*function to calculate factorial*/
14      int factorial(int n)      /*function definition*/
15      {
16      int i, res = 1;
17      for(i=1; i<=n; i++)
18      res = res * i;
19      print(res);
20      return res;
21      }
22      /*function to print a value*/
23      void print(int a)    /*function definition*/
24      {
25      printf("result : %d\n", a);    /*Line 26*/
26      }
```

- Initially stack section is empty. The main thread starts executing main() function in the code section [Line 03]. The function activation record for main() is created on the stack. It includes the local variables num, result and also stores the return address where the function returns when it is finished.
- The declaration statements for variables and factorial() function is used for informing compiler about them [Line 05 and 06]. These statements are not executed at run time.
- Then printf() function is called and display the message on screen [Line 07]. Since printf is also a function, its function activation record is created on the stack containing its local variables and arguments. But when function returns back after printing the message, that record is destroyed. Now stack contains stack frame for main().
- The scanf() function waits for user input and when number is entered, that value is stored in num variable [Line 08]. Like printf(), the stack frame for scanf is also created and destroyed immediately. Only main() stack frame remains on stack.
- Now factorial() function is called, with parameter num [Line 09]. The control will be shifted to factorial() function [Line 14]. With the call of factorial(), stack frame of the factorial() will be created on the stack. It contains local variables i, res, argument n and return address of next instruction (Line 09 for source code). The value of actual argument num, will be copied to formal argument n. Now stack has stack frame for main() and stack frame for factorial().
- The body of factorial() function will be executed to calculate factorial of n [Line 16, 17 and 18].
- Then print() function is called, with parameter res [Line 19]. The control will be shifted to print() [Line 23]. With the call of print(), stack frame of the print() will be created on the stack. It contains argument a and return address of next instruction (Line 19 for source code). The value of actual argument res, will be copied to formal argument a. Now stack has stack frame for main(), stack frame for factorial() and stack frame for print(). See the figure 6.4 (a).
- The body of print() will be executed [Line 25].

Scanned by CamScanner

- The print() function is finished [Line 26], so control will be shifted back to the calling function i.e. factorial() by retrieving return address from the stack frame of print() [Line 19 of Source code]. Also stack frame of print() is destroyed. Now stack has stack frame for main() and stack frame for factorial().
- The result of factorial is returned back to the calling function [Line 20]. Control will be shifted back to the calling function i.e. main() by retrieving return address from the stack frame of factorial() [Line 09 of Source code]. The return value of the function is assigned to the variable result and code after this will be continued. Also stack frame of factorial() is destroyed. Now stack has stack frame for main().
- The print() function is called, with parameter result [Line 10]. The control will be shifted to print() [Line 23]. With the call of print(), stack frame of the print() will be created on the stack. It contains the argument a and return address of next instruction (Line 11 for source code). The value of actual argument result, will be copied to formal argument a. Now stack has stack frame for main() and stack frame for print(). See the figure 6.3 (b).
- The body of print() will be executed [Line 25].
- The print() function is finished [Line 26], so control will be shifted back to the calling function i.e. main() by retrieving return address from the stack frame of print() [Line 11 of Source code]. Also stack frame of print() is destroyed. Now stack has stack frame for main().
- Now main() function returns 0 [Line 11] to the operating system by retrieving return address from the stack frame of main() and stack frame of main() destroyed and stack becomes empty. Now all the memory allocated for the process is released and then program is terminated.

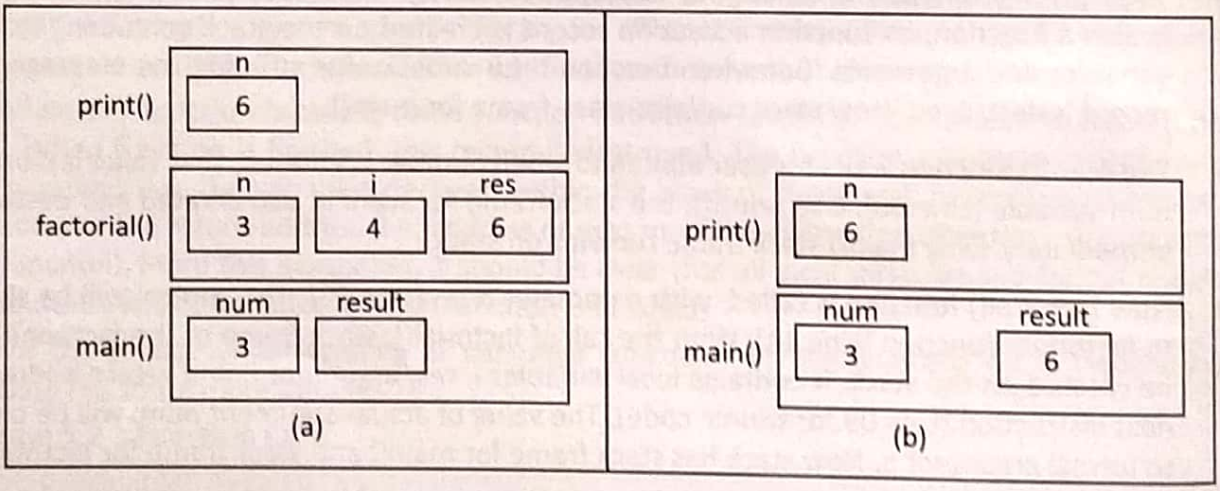


Figure 5.4 : Stack Frames

As discussed above and shown in figure 5.4, one can understand that stack keep growing with every call to the function and shrink with termination of that function. In case of recursion, multiple recursive calls are given to the function. This grows size of the stack by creating stack frame for every function call. The problem can occur if end condition for recursion is missing or given wrong. In this case, stack keeps growing, exhausting the gap available for the growth of the stack. Thus stack overflow occurs and program will be terminated abnormally. So while writing recursive functions, one must write the end condition properly.

Scanned by CamScanner

Calling Conventions :

The program 5.3 illustrates the idea of calling convention. Also note that function declaration is not required for this program, because the definition occur before its call.

Program 5.3 : Calling Conventions

```
#include <stdio.h>
void fun(int x, int y, int z)      /*function definition*/
{
    printf("%d%d%d\n", x, y, z);              /*6 6 6*/
}
int main()
{
    int a=3;
    fun(++a, ++a, ++a);              /*function call*/
    return 0;
}
```

In this program, one should notice that value of x is 6, y is 5 and z is 4. This makes it sure that arguments are evaluated from right to left.

Calling convention of the function mainly explains the order of pushing arguments on the stack (left to right or right to left) and stack cleanup responsibility (the arguments are popped by called function or calling function). There are few calling conventions listed into the table 5.3

The default calling convention for the C program is cdecl stands for c declarator. So arguments are pushed from right to left and when function is finished, arguments are removed from the stack by calling function. The size of machine level code for this calling convention is comparatively more due to repetition of machine level pop instructions after every call of the function.

Table 5.3 : Calling Conventions

	Calling Convention	Argument push Order	Stack Cleanup
1	cdecl	right to left	calling function
2	pascal	left to right	called function
3	stdcall	right to left	called function

Calling conventions should not be confused with *call by value or call by address.* Call by value or call by address simply explains whether copy of the parameter is created or its address is passed. Calling convention explains order of arguments and stack cleanup responsibility.

Storage classes

With the detailed knowledge of execution of the program learning storage classes will be fairly easy. It also helps to learn the behavior of the storage class and reason behind it.

There are two concepts widely used for storage classes i.e. Life and Scope of the variable. Life of variable is related to its existence in memory, while scope of variable is related to access or visibility of the variable at certain line of source code. As a common sense, if variable does not exist it can not be accessed; However, in certain cases the existing variable may not be accessible every where in the source code. Such attempt will report a compile time error.

There are total four storage classes in C. All the parameters related with these storage classes are mentioned in the table 5.4.

Scanned by CamScanner

Table 5.4 : Sections in Executable File

storage Class	keyword	memory	default value	Life	scope
local or automatic	auto	Stack	garbage	Block	block
register	register	cpu registers	garbage	Block	block
global or extern	extern	data section	zero	program	Program
static	static	data section	zero	program	limited

Automatic Storage class

All the variables declared within a function block (local variables) and formal arguments of that function have this storage class. As discussed earlier, these variables are automatically created as a part of stack frame they are destroyed, when function is finished along with the stack frame. The declaration of local variable can be optionally preceded by the keyword *auto*. All variables used in all the programs till now have auto storage class. However, a typical case for auto variables is shown in the program 5.4.

Program 5.4 : Auto Variables

```
#include <stdio.h>
int main()
{
  auto int a = 10;
  {
    auto int a = 20;
    {
      printf("%d", a);/*20*/    /* access variable declared in outer block*/
    }
  printf("%d", a);       /*20*//* access variable declared in its block*/
  }
  printf("%d\n", a);     /*10*//* access variable declared in its block*/
  return 0;
}
```

Note that in this example, a variable is declared three times within three different blocks. When variable is accessed in same block if access, the variable declared in that block. If the variable is not declared in that block, it can access the variable in the enclosing block and so on. However, the variable or argument of one function cannot be accessed into other function directly.

Local variables are stored on the stack and they have garbage value before any initialization. Garbage value is any unpredictable value. One should take care of initializing the variable before its use, otherwise unexpected results will appear.

Register storage class

Any local variable can be preceded with register keyword and such variable will be stored in CPU register if register is available. Note that making a variable register is just a request to the system and if the CPU register is not available the variable will be simply treated as a local variable. The default value for these variables is garbage and they are accessible in the block where they are declared just like local variables.

The main advantage of register variable is the faster access compared to the other variables. Register variables are used for the variables that are accessed frequently. The best example can be loop variables as shown in the program 5.5.

Scanned by CamScanner

Program 5.5 : Register Variables

```
#include <stdio.h>
int main()
{
    register int a;
    for(a=1; a<=100; a++)
        printf("%d ", a);
    return 0;
}
```

Extern storage class

Program 5.6 illustrates the use and syntax of global variables. The variables declared outside any function is called as global variables. Generally they are declared at the top of source code after file inclusion statements. These variables are located into data section of executable file and from there they are loaded into data section of the process in the RAM. Thus these variables are created as soon as program starts and are destroyed when entire process memory is released. In other words, life time of these variables is throughout the program.

These variables are accessible in any function throughout the program. The variables declared in one file of the project can be accessed in the other file using extern keyword. The default value of the global variables is zero.

Program 5.6 : Extern Storage Class

```
/*main.c*/

#include <stdio.h>
#include "sumpro.h"
/*defining global variables*/
int sum, pro;
int main()
{
    int num1=12, num2=4;
    sumpro(num1, num2);
    printf("sum : %d\n", sum);
    printf("pro : %d\n", pro);
    return 0;
}
```

```
/*sumpro.c*/

/*declarations of
global variables*/
extern int sum, pro;

void sumpro(int a, int b)
{
sum = a + b;
pro = a * b;
}
```

```
/*sumpro.h*/

/*function declaration*/
void sumpro(int , int);
```

The file main.c defines the global variables while *sumpro.c* file just declare them. The main difference between definition and declaration is that, definition causes to allocate the memory for that symbol while declaration simply provides information of that symbol before it is used. This rule applies to global variables which are stored in data section. Even the rule is applies to function which are stored in the text section.

It can be observed that new learners use global variables heavily because of their ease of use. However, there are some serious problems of global variables.

- Global variables can be accessed through out the program and hence it will be kept in main memory during the entire execution of the program even though it is not used all the time.
- Global variables can be modified from any function in the program. This makes it difficult to track its value in case of large programs where hundreds of functions are involved. This really troubles in bug fixing.
- Global variables are located in executable image, so using number of global variables will increase the size of executable image.

It is always advised to use global variables minimum possible. It should be used if and only if most of the functions access that variable.

Scanned by CamScanner

STATIC STORAGE CLASS

The static variables can be declared locally or globally preceded by the static keyword. In one line, static is same as global variable except that their scope is limited where they are declared. If static variable is declared within a function block, its scope is limited to that block and cannot be accessed outside that block; while if static variable is declared globally its scope is limited to that file and it cannot be accessed in other file of the same project.

Program 5.7 : Static Variables

```
#include <stdio.h>
void fun()
{
      /*declaring static variable*/
      static int a = 1;
      printf("%d ", a);
      a++;
}
int main()
{
      fun();
      fun();
      fun();
      return 0;
}
```

Like global variables life time of static variables is throughout the program and its default value is zero. The following example illustrates the concept and syntax of static variables.

The output of the program is

1 2 3

Initially when program is loaded into the memory all static variables are initialized with the given values like the global variables. So the statement

static int a = 1;

is not executed in every call of function. Hence calling the function consecutively print the value of a and increment it every time.

As a common mistake, static variable is considered as constant. It should be clear that the word static represents static scope not the static value. For constant or non–modifiable variables const keyword is there in C language.

Like global variables using number of static variables can increase the size of executable image. So static must be used carefully.

Lab Exercise

Q.1. Declare a global variable *error_value* in your program. Write a function to perform four function calculator. If the condition divide by zero set the value of global variable *error_value* to –1. Write a function to print error message.

Q.2. A linear congruential generator (LCG) represents one of the oldest and best–known pseudorandom number generator algorithms. The Algorithm is as follows

$$X_{n+1} = (a \times X_n) \bmod m$$

Where X_{n+1} is next random number. X_n is current random number. *a* is multiplier. M is modulus which is greatest random number plus one. Using static variable, write a function to generate random numbers.

Scanned by CamScanner

Objective Questions

What will the output of the following programs?

Q.1.
```
#include <stdio.h>
register int x;
int main()
{
    printf("%x", x);
    return 0;
}
```

1) 0
2) garbage
3) compiler error
4) 0x

Q.2.
```
#include <stdio.h>
extern int x;
int main()
{
    printf("%d", x);
    return 0;
}
int x;
```

1) 0
2) garbage
3) compiler error
4) linker error

Q.3.
```
#include <stdio.h>
int main()
{
    int x = 3;
    static int y = x;
    printf("%d, %d", x, y);
    return 0;
}
```

1) 3, 3
2) 3, 0
3) compiler error
4) linker error

Q.4.
```
#include <stdio.h>
int main()
{
    register int x = 3;
    printf("%p, %x", &x, x);
    return 0;
}
```

1) address of x, 3
2) garbage, 3
3) compiler error
4) address of x, 3h

Q.5.
```
#include <stdio.h>
int main()
{
    int main = 3;
    printf("%o", main);
    return 0;
}
```

1) 3
2) compiler error
3) 3h
4) 03

Scanned by CamScanner

Q.6.
```
#include <stdio.h>
int main = 3;
int main()
{
     printf("%o", main);
     return 0;
}
```

1) 3
2) compiler error
3) 3h
4) 03

Q.7.
```
#include <stdio.h>
int x = 3;
int main()
{
  int x = 4, i;
  for(i = 0; i < x; ++i)
  {
    int x = 0; printf("%d ,", x++);
  }
  return 0;
}
```

1) 0, 1, 2
2) compiler error
3) 0, 0, 0, 0,
4) 0, 1, 2, 3,

Scanned by CamScanner

Chapter 6 : Preprocessor Directives

The C preprocessor is a program that processes the source program before it is passed to the compiler. All statements in C source code starting with # are treated as commands to preprocessor and are called as *Preprocessor Directives*. These directives can be placed anywhere in the program but are most often placed at the beginning of a source code file. As a standard practice, these directives should be written at the start of the line. (Without any leading white space or character) These directives are processed by the compiler and hence are not available in compilation stage. The result of processing is placed in a temporary file. The file can be created in memory or hard drive depending on implementation of the compiler. However, using some options at command line compilation, one can get the contents of this file. For more details on command line compilation, see Appendix D. Preprocessor directives are classified as : –

- Macro expansion
- File inclusion
- Conditional compilation
- Miscellaneous

Macro expansion

Macros provide a mechanism for token replacement the preprocessor. Macros may have some arguments or may not have arguments.

Macros without arguments

#define macro_name macro_expansion

Macro is generally written at the top of the program as per the syntax given above. For example :

```
# define PI       3.142
# define FALSE    0
# define MSG      "welcome to SunBeam\n"
```

Preprocessor replace the macro name with macro expansion. The concept will be clear with the following example.

Program 6.1 Macros without Argument

Original source code

```
#define PI 3.142
int main()
{
    double radius, area;
    printf("Enter radius:");
    scanf(("%lf", &radius);
    area = PI * radius * radius;
    printf("area %.2lf\n", area);
    return 0;
}
```

After preprocessing

```
int main()
{
    double radius, area;
    printf("Enter redius:");
    scanf(("%lf", &radius);
    area = 3.142 * radius * radius;
    printf("area %.2lf\n", area);
    return 0;
}
```


Scanned by CamScanner

Macros are really helpful when some constants are to be used throughout the program. Change at a single place, will reflect the changes at all occurrences of that constant in the source code. Such constants are also referred as symbolic constants. Example : writing PI macro will be helpful, to easily change the value of PI from 3.142 to 3.1415, and improve the accuracy in the entire program.

Macros with arguments

#define macro_name(arguments) macro_expansion

Macros can even take arguments as per the given syntax. For example :

```
#define SQR(a) a*a
```

Program 6.2 Functions With Macros

Original source code	After preprocessing

`#define SQR(a) a*a`
`int main()`
`{`
`  int num, res;`
`  printf("Enter a number: ");`
`  scanf("%d", &num);`
`  res = SQR(num);`
`  printf("%d", res);`
`  return 0;`
`}`
	`int main()`
	`{`
	`  int num, res;`
	`  printf("Enter a number: ");`
	`  scanf("%d", &num);`
	`  res = num*num;`
	`  printf("%d", res);`
	`  return 0;`
	`}`

Here, *a* is an argument of the macro. If macro has multiple arguments, they should be separated by commas. Remember that macros follow *find and replace* principle. It should be written properly otherwise can give unexpected results e.g. res = SQR(2+7); get replaced as res = 2+7*2+7; And will give wrong result. So macro can be modified to give valid results as :

```
#define SQR(a) (a)*(a)
```

Macros may not be defined only at the top of the source file, they can be defined at the middle of the source code and will take effect thereafter into the source code.

Also macros can be given during compilation on command line or using some setting in IDE. e.g. in Visual C++ 6.0 in the menu Project –> Settings, On C/C++ tab option is provided for Preprocessor Directives. This way of giving macro declaration while compilation is mostly used as a standard practice, because it avoids modification is source code again and again.

There is no need of confusing between macros and functions. There are few differences between them as per Table no 6.1. For short formulae and code snippets macros are very efficient, while for multi–line code snippet functions will be a better choice.

Table 6.1 Difference between Function and Macro

Function	Macro
Functions are called at runtime.	Macros are replaced before compilation.
Due to runtime call, functions are slower.	Macros are much faster.
Function arguments have some data type.	Macro arguments do not have data type.
Functions can be recursive.	Macro cannot be recursive.
Calling same function multiple times, will not increase size of the file.	Calling multi–line macro multiple times, will replace it multiple times and will cause increase in size of file.

Scanned by CamScanner

File inclusion

Right from the first program, the first line of code we have written as #include <stdio.h>. Note that this statement start with "#" means that it is preprocessor directive. This is called as file inclusion directive. The syntax is as follows :

#include <filename> OR *#include "filename"*

This directive tells preprocessor to include the code from file namely "filename', into the source file. The first version use <>, while second version use"".

If file name is given in triangular brackets (<>), then file is searched into standard directory. The path of standard directory is given in compiler settings. If file name is given in double quotes (""), then file is searched into the current directory (directory of source file) and if it is not found then it is searched into standard directory.

Settings for standard directory may be different for every compiler.

- For Standard directory settings in Turbo C compiler IDE follow the following steps :
  - Go to menu Options – > Directories
  - In "Include directories" path of standard directories can be found.
- For standard directory settings in Visual C ++ 6.0 IDE follow the following steps.
  - Go to menu Tools – > Options and then directories tab.
  - In Show directories for combo box select Include file, then following list box shows standard include directories.

File inclusion directive can be used for any kind of file to be included. However, it is mainly used to include header files (.h files). Header file contains declaration of functions, macros, structures, etc. Standard header files are always available with the compiler software and also we may write our own header file if needed. Generally predefined header files are included using triangular brackets, as they are kept in standard directory and user defined header files included using double quotes, as they are kept in current project directory.

Conditional compilation

There can be the requirement to compile a set of code if certain condition is satisfied. Many times this can be required for writing some code for different kind of devices or to port some code on different compilers. At this time some piece of code will be applicable for one kind of device and other code must be compiled for other kind of devices. Also in some cases there is need to have different code for ASCII version and different code for UNICODE version.

ASCII is 8 bit code for characters, while UNICODE is 16 bit code for characters. Being 8 bit code ASCII cannot have support for alphabets in multiple languages. But in case of UNICODE, due to use of 16 bits the wide range of values typically 0 to 65535 is available. UNICODE supports all widely used languages, which makes applications to be used in different languages.

C language do not have built in support for UNICODE, However, one data type has been created by the name wchar_t (declared in wchar.h) which is internally typedef to unsigned short.

The conditional compilation directives are #if, #else, #elif, #ifdef, #ifndef, #endif, #undef. The following example program 6.3 gives idea of conditional compilation. This program allows writing a code that will be compiled on C compiler and even on C++ compiler. Another example of conditional compilation is given in program 6.7.

Scanned by CamScanner

Program 6.3 Conditional Compilation

```
#include <stdio.h>
#ifdef ___cplusplus
     /*C++ specific code*/
#else
     /*C code*/
#ifdef __STDC__
     /*standard C code*/
#else
     /*non-standard C code*/
#endif
#endif
```

Miscellaneous directives

Other than the directives discuss above few more preprocessor directives are available. These directives are discussed below.

Operator

This is a preprocessor operator, which converts a macro argument to string constant. The example 6.4 shows use of operator # :

Program 6.4 # Operator

Original Source Code	After Preprocessing

`#define PRINT(arg) printf(#arg)`
`int main()`
`{`
`  PRINT(Sunbeam);`
`  return 0;`
`}`
	`int main()`
	`{`
	`  printf("Sunbeam");`
	`  return 0;`
	`}`

Operator

This is a preprocessor operator, which is used to combine two tokens together.

Program 6.5 # # Operator

Original Source Code	After Preprocessing

`#define PRINT(a,b) printf("%d",a##b)`
`int main()`
`{`
`  int basicsalary=5000;`
`  PRINT(basic, salary);`
`  return 0;`
`}`
	`int main()`
	`{`
	`  int basicsalary=5000;`
	`  printf("%d", basicsalary);`
	`  return 0;`
	`}`

The use of # and ## operators is not very common in programming. But it can used while designing compilers and for certain features. The typical example for ## operator can be used to create UNICODE string.

Scanned by CamScanner

Program 6.6 Unicode Strings

```
#define TEXT(str) L##str
int main()
{
  int a = sizeof("Sunbeam");
  int b = sizeof(TEXT("Sunbeam")); /*int b = sizeof(L("Sunbeam"));*/
  printf("ASCII:%d UNICODE:%d", a, b);
  /*output as ASCII:8 UNICODE:32*/
  /*Since sizeof UNICODE is twice the sizeof ASCII character*/
  return 0;
}
```

#error directive

If this directive is processed by the preprocessor, preprocessing is stopped and the message is printed as an error.

Program 6.7 #error directive

```
#include <stdio.h>
#define VERSION 2
int main()
{
#ifndef VERSION
/*if above VERSION is removed, this error will be displayed*/
#error "Version is Not defined"
#endif

#if VERSION == 1
printf("version1 : specific code");
#elif VERSION == 2
    printf("Version2 specific code");
#else
    printf("Other Version code");
#endif
return 0;
}
```

#line directive

As per ANSI standard, there are few predefined preprocessor symbols available, which are listed in following table. By default,__LINE__symbol represents number 1 for first line and keep incrementing for each line. This information is used by the compiler during compilation to display errors.

Table 6.2 Predefined Preprocessor Constants

Preprocessor Symbol	Meaning
__LINE__	Integer that represent current line number
__FILE__	String that represent current file name
__DATE__	String that represent current date
__TIME__	String that represent current time

However, current line number or file name information can be modified using #line directive. Its syntax is given below, where writing file name is optional.

#line number "filename"

Scanned by CamScanner

Program 6.8 # Line directive

```
01   #include <stdio.h>
02   int main()
03   {
04   printf("%s : %d\n", __FILE__, __LINE__);      /*print filename and */
05                                                 /*current line number */
06        #line 100
07        printf("%d", __LINE__);                 /*print line number as 100*/
08        return 0;
09   }
```

#pragma directive

This is a special purpose directive that can be used to turn ON or OFF certain features of compiler and linker. Note that #pragma directives will vary from compiler to compiler. The information about the #pragma on the specific compiler can be found into the documentation or help file for that compiler. These directives can be used to achieve some optimization, suppress some warnings, etc. Since they are compiler dependent, to avoid the confusion with different compiler the example is avoided.

Lab Exercise

Write a macro to print the given integer variable value along with its name.

Q. 1. Write a macro to determine whether given character is lowercase or not.

Q. 2. Write a macro to change the case of given character to uppercase.

Q. 3. Write a macro to swap values of two values.

Q. 4. Write all above macros in file myheader.h. Include the same file in your program.

Objective Questions

What will the output of the following programs?

Q. 1.

```
#include <stdio.h>
#define area(x)  (x * x)
int main()
{
     printf("%d ",area(6 + 3));
     return 0;
}
```

1) 81
2) 27
3) 63
4) 21

Q. 2.

```
#include <stdio.h>
#define print_mul(x, y) printf(#x" * " #y "= %d", x * y)
int main()
{
     int a = 3;
     print_mul(a, 3);
     return 0;
}
```

1) # x * #y = 9
2) X * 3 = 9
3) a * 3 = 9
4) compiler error

Scanned by CamScanner

Q.3.
```
#include <stdio.h>
int main()
{
    printf("%d", __LINE__);
    return 0;
}
```

1) 3
2) __LINE__
3) 6
4) 4

Q.4.
```
#include <stdio.h>
#define max(a, b) (a > b)? a : b
int main()
{
    int x = 3, y = 4, z;
    z = max(++x, y++);
    printf("%d, %d, %d", x, y, z);
    return 0;
}
```

1) 4, 6, 5
2) 4, 5, 6
3) 3, 4, 3
4) 4, 5, 4

Q.5.
```
#include <stdio.h>
#define greater(a, b) (a > b)? a : b
int main()
{
    int x = 3, y = 4;
    if(greater(x, y))
        printf("sun");
    else
        printf("beam");
    return 0;
}
```

1) sun
2) beam
3) sunbeam
4) compiler error

Scanned by CamScanner

Chapter 7 : One Dimensional Arrays

Many times there is need of grouping elements of same data types. For example, marks of 6 subjects. Now marks of 6 subjects can be stored in 6 int variables like marks0, marks1, etc. However, accessing these 6 values as a single variable would be easier to use and will make the program more readable. In C language grouping is done using the language feature called as *arrays*.

Array is a finite collection of similar data elements in consecutive memory locations. All elements in the array share the same name i.e. name of array. However, each element in the array is uniquely identified by some index or subscript. The first element in the array is indexed as zero. So the last element in the array has index equal to number of elements minus one. In short, array with n elements have range of index from 0 to n–1. The individual member of the array can be accessed using subscript operator ([]). The *arr [i]* can be used to access i^{th} element of the array, where *i* is some valid index for that array.

1–D arrays

As defined earlier, array is finite collection of similar data elements in consecutive memory locations. The number of elements in the array must be specified while declaring the array along with the data type of the array. The array elements are always stored in consecutive memory locations. The following diagram shows array of 5 double elements. The starting address is taken arbitrarily as 400. It can be observed that address of two consecutive elements differ by 8, because size of double is 8 bytes.

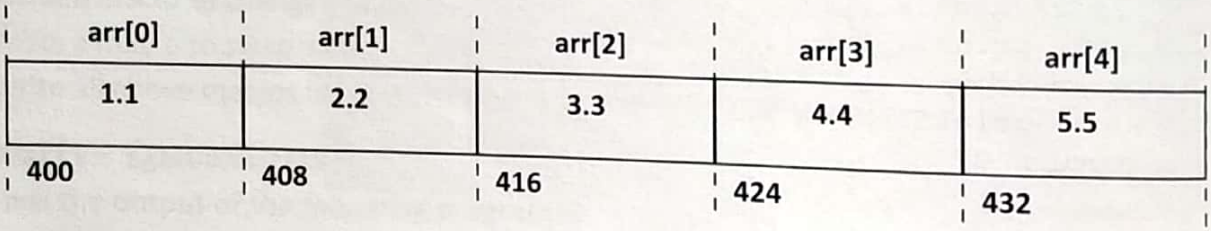


Figure 7.1 Representation of 1–D array in memory

This makes arrays accessible by using *pointer arithmetic*. The more details will be available in upcoming topics.

1–D Array Basics

The following example will help you to understand the syntax and concept of array.

Problem : Input the marks of 6 subjects for a student from the user. Calculate the average and display it along with the marks entered.

Program 7.1 : 1–D Array

```
01   #include <stdio.h>
02   int main()
03   {
04   int marks[6];  /*array declaration  */
05   int i, sum=0;
06   double avg;
07   printf("enter array elements:");
```


Scanned by CamScanner

```
08   for(i=0; i<6; i++)
09        scanf("%d", &marks[i]);  /*scan array elements  */
10   for (i=0; i<6; i++)
11        sum = sum + marks[i];    /*summing array elements  */
12   avg = (double)sum / 6;   /*calculate the average  */
13   for(i=0; i<6; i++)
14        printf("%d ", marks[i]); /*print array elements  */
15   printf("\navg: %.2lf", avg);  /*print the average  */
16   return 0;
17   }
```

Array declaration

Syntactically one dimensional array is declared as :

data–type variable–name [array–size];

The above declaration means, the array with name *variable–name* can store *array–size* number of elements of type *data–type*. In above example array *marks* can store 6 elements of type *int* [Line 04]. Since array is declared locally and it is not initialized, all elements will by default get garbage value.

Accessing array elements

Each element in the array can be accessed using subscript operator. Typically *arr[i]* represents i^{th} element of the array, provided *i* is some valid index for that array. Now *arr[i]* can be used for reading value or can be used for assigning some value to the i^{th} element of the array *arr*. For example :

```
/*array declaration*/
int a[4];
/*assigning 5 to a[2] i.e. third element of a*/
a[2] =5;
/*assigning third element to second element of a*/
a[1] = a[2];
/*sum of second and third element is assigned to first element of a*/
a[0] = a[1] + a[2];
```

To access all elements of the array, it is common practice to use a loop where loop counter *i* increments from *0* to *size–1*. In the example program 7.1, all array elements are read for summing them [Line 10 and 11].

Finally one must remember that, the valid index must be used for accessing elements, (i.e. 0 to n–1, if n is size of array). Accessing element at invalid index; will not cause any compile time error. This may access an unexpected error or even terminate the program abnormally. So remember that checking array bounds is responsibility of programmer, not a Compiler.

Printing array

Each element of the array can be simply passed to printf() for printing it. Obviously appropriate format specifier must be given in format string. For example, for array of int %d or for array of double %lf must be given. The same thing is done in above program [Line 14 and 15].

Scanned by CamScanner

Array initialization

The array can also be initialized at the point of declaration. For example :

```
int a[5] = {11, 22, 33, 44, 55};
int b[] = {11, 22, 33, 44};
int c[5] = {11, 22, 33};
```

In above declaration *a* is an array of 5 elements and all 5 elements are given in curly braces. This initializes each element in the array from left to right fashion.

For array *b* size is not declared, However, 4 elements are given in curly braces, so size of b is considered as 4. Thus if array is initialized at the point of declaration, giving size of array is optional; in this case it will be implicitly calculated by the compiler.

The size of array *c* is declared as 5, but only 3 elements are initialized. In this case last two element will be initialized to zero. Thus if array is initialized partially at its point of declaration, remaining elements are initialized to zero.

Also array can be initialized by initializing each member of the array. As discussed above, arr[i] will represent i^{th} element of the array. Such initialization example is given below :

```
int arr[3];                    /*declaring array*/
arr[0] = 10;          /*initializing first element*/
arr[1] = 20;         /*initializing second element*/
arr[2] = 30;          /*initializing third element*/
```

For a compact code, one can use loop to access all elements of the array. In above example, all elements are initialized by scanning values from the user using scanf() [Line 08 and 09]. Note that & *marks[i]* represent address of individual element in the array.

1–D Array and pointers

The array and pointers are closely tied with syntax as well as concepts. Arrays use *pointer arithmetic* heavily to access the array elements. The arrays cannot be learned properly without having good knowledge of pointers.

Pointer to array

The pointer to array means the pointer variable store the address of very first element (index 0) of the array. The pointer variable must be declared of the same type as that of first element of the array, which is same as type of array. For example, pointer to array of double must be of double type.

Program 7.2 : Pointer to 1–D array

```
#include <stdio.h>
int main()
{
  double arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};   /*array declaration*/
  double *ptr;                                  /*pointer declaration*/
  ptr = &arr[0];                               /*pointer initialization*/
  printf("ptr=%u\n", ptr);                /*address of 0th element (400)*/
  printf("*ptr=%.11f", *ptr);              /*value of 0th element (1.1)*/
  return 0;
}
```


Scanned by CamScanner

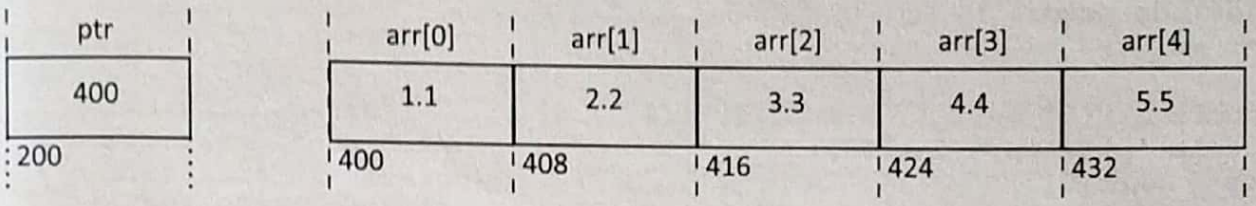


Figure 7.2 : Pointer to 1–D array

In program 7.2 double pointer *ptr* keeps the address of first element of the array, so it can be considered as pointer to array *arr*. The diagram 7.2 is drawn to make the visual understanding of the program. The address of array and pointer are chosen arbitrarily. The expected output is written into comments of the program.

Pointer arithmetic

In chapter 2, it is discussed that there are arithmetic operators like +, –,*, / and %. However, we used these operators for built–in primary types like int, float, etc. The *pointer arithmetic* discusses the arithmetic operations done on the pointer.

The most important terminology in the pointer arithmetic is the *scale factor* of the pointer. The size of data type of the pointer is known as *scale factor* of that pointer. For example, scale factor of double pointer is 8, scale factor of char pointer is 1, etc.

The first type of operation involves one operand as pointer of some type and another operator is some integer value. In other type, both the operands are pointers.

The sum of integer *n* and any pointer will result in the address, which will be *n * scale factor* bytes greater than the address in the pointer. For example, if double pointer *ptr* contains address 400, then *ptr + 1* will result in 408, *ptr + 2* will result in 416 and so on.

The subtraction of integer *n* from any pointer will result in the address, which will be *n * scale factor* bytes smaller than the address in the pointer. For example, if double pointer *ptr* contains address 432, then *ptr –1* will result in 424, *ptr – 2* will result in 416 and so on.

Note that multiplication, division and modulus of any integer with the pointer is not allowed. At the same time, addition, multiplication and division of two pointers is not allowed.

The subtraction of two pointers can be meaningful if both are storing the addresses of the elements of the same array. In this case, it gives the number of elements between the elements whose address is stored into the pointer. The simple calculation suggest that,

$$\text{result} = \frac{\text{(address in pointer 1 — address in pointer 2)}}{\text{scale factor of pointer 1}}$$

Note that, subtracting address of some element from the address of 0^{th} element of the array will give the index of that element. The program 7.3 will help to understand all possible arithmetic operations on the pointer. Also the figure 7.2 can be used to visualize the pointers in memory. Note that all results are explained in comments.

Scanned by CamScanner

Program 7.3 : Pointer Arithmetic

```
#include <stdio.h>
int main()
{
double arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};      /*array declaration*/
double *ptr, *p = &arr[3];                      /*pointer declaration*/
ptr = &arr[0];                                 */pointer initialization*/
printf("%u %u %u\n", ptr, ptr+1, ptr+2);        /*output: 400 408 416*/
ptr = ptr + 3;
printf("%u\n", ptr);                                        /*output: 424*/
ptr++;
printf("%u\n", ptr);                                        /*output: 432*/
printf("%u %u %u\n", ptr, ptr-1, ptr-2);        /*output: 432 424 416*/
ptr = ptr - 3;
printf("%u\n", ptr);                                        /*output: 408*/
--ptr;
printf("%u\n", ptr);                                        /*output: 400*/
printf("%d\n", p-ptr);                        /*output: 3 i.e. (424-400)/8*/
return 0;
}
```

Accessing array using pointer

As an example program 7.3, shows that each address in the array can be accessed using pointer arithmetic. The same concept can be enhanced to access array element at that address using *value at* operator(*). The next program 7.4, illustrate this concept.

Program 7.4 : Accessing 1–D Array via Pointer

```
01 #include <stdio.h>
02 int main()
03 {
04 double arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5);      /*array declaration*/
05 double *ptr;                                    /*pointer declaration*/
06 int i;                                                  /*loop counter*/
07 ptr = &arr[0];                                  /*pointer initialization*/
08 printf("%u %.1lf\n", ptr, *ptr);    /*address and value of 0th element*/
09 /*print all address and value of each element*/
10 for(i=0; i<5; i++)
11   printf("%u %.1lf\n", ptr+i, *(ptr+i));
12 printf("%u %.1lf\n", ptr, *ptr);    /*address and value of 0th element*/
13 /*print all address and value of each element*/
14 for(i=0; i<5; i++)
15 {
16   /*print address and value of current element*/
17   printf("%u %.1lf\n", ptr, *ptr);
18   /*address in pointer incremented to next element*/
19   ptr++;
20 }
21 return 0;
22 }
```

The above program is self explanatory if pointer arithmetic is understood well. Assuming the

Scanned by CamScanner

base address of array is 400 as shown in figure 7.2, the output of the above program will be :

```
400 1.1     [Line 08]
400 1.1     [Line 11]
408 2.2     [Line 11]
416 3.3     [Line 11]
424 4.4     [Line 11]
432 5.5     [Line 11]
400 1.1     [Line 12]
400 1.1     [Line 17]
408 2.2     [Line 17]
416 3.3     [Line 17]
424 4.4     [Line 17]
432 5.5     [Line 17]
```

Base Address

Array name itself represents the address of 0th element of the array, which is also referred as base address. It can be proved by single line of code :

```
printf("%u %u %.11f", &arr[0], arr, *arr);
```

Keeping figure 7.2 in mind, the above line prints 400 400 1.1.

```
ptr = &arr[0];
ptr = arr;
```

Also both of these lines mean same and can be used alternatively. Since array name is equivalent to the address of first element, the rules of pointer arithmetic can be applied to the array name. This can be explained by the next program :

Program 7.5 : Base address of 1–D array

```
01 #include <stdio.h>
02 int main()
03 {
04   double arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};   /*array declaration*/
05   int i;
06   printf("%u %.11f\n", arr, *arr);    /*address and value of 0th element*/
07   /*print all address and value of each element*/
08   for(i=0; i<5; i++)
09       printf("%u %.11f\n", arr+i, *(arr+i));
10   return 0;
11 }
```

Assuming the base address of array is 400 as shown in figure 7.2, the output of the above program will be :

```
400 1.1     [Line 06]
400 1.1     [Line 09]
408 2.2     [Line 09]
416 3.3     [Line 09]
424 4.4     [Line 09]
432 5.5     [Line 09]
```


Scanned by CamScanner

Base address and pointer

As discussed earlier in this chapter array element can be accessed using subscript operator on array name. But the above program use *value at* operator on array name to access the array. This is possible only because, array name represents address of the 0th element.

Studying a little depth will reveal that, subscript operator internally follow the pointer arithmetic only. In other words, if *arr* is considered as name of array then one can understand that :

```
arr[i] = *(arr + i)
```

And little common sense can explain that,

```
arr[i] = *(arr + i) = *(i + arr) = i[arr]
```

The next program 7.6 will prove the same thing. If it is well understood that subscript operator is internally replaced with pointer notation by the compiler, then one may conclude that :

```
*(ptr+i) = ptr[i]
```

And this really works well. Now, one should not surprise for :

```
ptr[i] = *(ptr + i) = *(i + ptr) = i[ptr]
```

The following program summarizes all the concepts discussed. This program trying to read the array using number of different notations :

Program 7.6 : Pointer Vs Base Address (a)

```
#include <stdio.h>
int main()
{
    double arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};      /*array declaration*/
    double *ptr;                                    /*pointer declaration*/
    int i;                                          /*loop counter*/
    ptr = arr;                                      /*pointer initialization*/
    printf("\narr[i] : ");
    for(i=0; i<5; i++)
       printf("%.1lf ", arr[i]);
    printf("\n*(arr+i) : ");
    for(i=0; i<5; i++)
       printf("%.1lf  ", *(arr+i));
    printf("\n*(i+arr) : ");
    for(i=0; i<5; i++)
       printf("%.1lf ", *(i+arr));
    printf("\ni[arr] : ");
    for(i=0; i<5; i++)
       printf("%.1lf ", i[arr]);
    printf("\nptr[i] : ");
    for(i=0; i<5; i++)
       printf("%.1lf ", ptr[i]);
    printf("\n*(ptr+i) : ");
    for(i=0; i<5; i++)
       printf("%.1lf ", *(ptr+i));
    printf("\n*(i+ptr) : ");
    for(i=0; i<5; i++)
      printf("%.1lf ", *(i+ptr));
    printf("\ni[ptr] : ");
    for(i=0; i<5; i++)
       printf("%.1lf ", i[ptr]);
    return 0;
}
```


Scanned by CamScanner

The output of the above program prints the same array again and again as :

```
arr[i]   : 1.1   2.2   3.3   4.4 5.5
*(arr+i)   : 1.1   2.2   3.3   4.4   5.5
*(i+arr)   : 1.1   2.2   3.3   4.4   5.5
i[arr]   : 1.1   2.2   3.3   4.4   5.5
ptr[i]   : 1.1   2.2   3.3   4.4   5.5
*(ptr+i)   : 1.1   2.2   3.3   4.4   5.5
*(i+ptr)   : 1.1   2.2   3.3   4.4   5.5
i[ptr]   : 1.1   2.2   3.3   4.4   5.5
```

The program 7.6 tries to use subscript operator as well as value at operator on array name, similarly it tries to use subscript operator as well as value at operator on pointer. This may leads to confusion that array name and pointer can be used alternatively. However, this is not true. The following program tries to explain the difference.

Program 7.7 : Pointer Vs Base Address (b)

```
#include <stdio.h>
int main()
{
    double arr[5] = {1.1, 2.2, 3.3, 4.4, 5.5};          /*array declaration*/
    double *ptr;                                        /*pointer declaration*/
    int i;                                                    /*loop counter*/
    ptr = arr;                                        /*pointer initialization*/
    printf("%d%d", sizeof(ptr), sizeof(arr));                 /*output: 4 40*/
      ptr++;                                                      /*no error*/
    /* arr++; */                                             /*compiler error*/
    return 0;
}
```

The major difference that can be observed from the above program is that, size of pointer will be 4 bytes and size of array will be 40 bytes (assuming 32–bit compiler). Since *ptr* is a variable that keeps the address, 4 bytes are sufficient and *arr* is an array of 5 double elements, its size is 40 bytes (size of one double is 8 bytes).

Also since *ptr* is a variable, address stored in that can be modified by using increment or decrement operator. However, *arr* do not store the address (it simply represents that address when used in some expression) and hence using increment or decrement operator on *arr* raise compile time error.

Passing array to the function

Array can be passed as an argument to function. To pass the array to the function array name (i.e. base address of the array) is passed as parameter. Since it represents address of the first element, it can be accepted into the pointer. However, the formal argument can be declared in two different ways. The typical declaration of function will be one of the follows :

```
void sort(double *a, int n);
void sort(double a[], int n);
```

Understand that both of the above are equivalent and can be used alternatively. In both the

Scanned by CamScanner

cases *a* is a pointer that keeps the address of first element of the array, only syntax is different. However, second syntax can be preferred to represent that array is passed as an argument. Using any of the declaration of the function will not make any difference in the implementation of the function.

The following program can be used to understand the syntax of passing array to the function and accessing it into body of the function.

Program 7.8 : Passing 1–D array to Function

```
#include <stdio.h>
void input(double a[], int n);
void display(double a[], int n);
void sort(double a[], int n);
int main()
{
	double arr[5];
	input(arr, 5);
	sort(arr, 5);
	display(arr, 5);
	return 0;
}
void input(double a[], int n)
{
	int i;
	printf("enter %d elements:", n);
	for(i=0; i<n; i++)
		scanf("%lf", &a[i]);
}
void display(double a[], int n)
{
	int i;
	for(i=0; i<n; i++)
		printf("%.2lf ", a[i]);
	printf("\n");
}
void sort(double a[], int n)
{
	int i, j;
	double t;
	for(i=0; i<n-1; i++)
	{
		for(j=i+1; j<n; j++)
		{
			if(a[i] > a[j])
			{
				t = a[i];
				a[i] = a[j];
				a[j] = t;
			}
		}
	}
}
```


Scanned by CamScanner

This example shows standard way of using array in the program. All user defined functions in this program (except main()), take array as first argument and its size as second argument. Due to this function becomes independent of the size of the array and no need to modify it even if array size is changed.

In any of these function e.g. print() the argument *a* to the function is actually the pointer to array. This can be verified by printing the size of *a* in the function; it comes out to be 4 on 32 bit compilers. However, in these functions array is accessed in the same manner as it was accessed in main() in program 7.1. This is possible due to the fact that, subscript notation can be used on pointer to the array. The same program can modified to use pointer notation, However, array notation is much readable as compared to pointer notation.

The sort() function arrange all the elements in ascending order. It implements a well known algorithm called as *selection sort*. The detailed explanation of selection sort can be found in chapter 12.

Strings

The examples of 1–D array shown above are mainly of *double* data type. But 1–D array can be of any type like int, long, char, etc. However, array of character have some special significance, because in the program many times there is a need of showing some text message to user or to take some information from the user in form of set of characters.

The array of character can be declared just like any other array. The following program 7.9 and diagram below that will make the things clear.

Program 7.9 : Array of Characters

```
#include <stdio.h>
int main()
{
  int i;
  char arr[] = {'W', 'i', 'M', 'C'}; /*declare and initialize array*/
  printf("size = %d\n", sizeof(arr));                    /*output: 4*/
  for(i=0; i<4; i++)
     putchar(arr[i]);                    /*print characters one by one*/
  return 0;
}
```

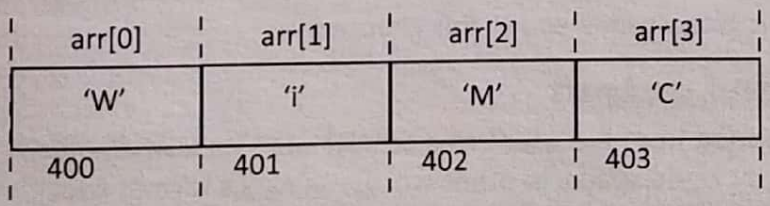


Figure 7. 3 : Array of Characters

The *arr* is declared as an array of characters and four characters have been initialized, so size of characters will be taken as 4 bytes. Then each element of the array is accessed using loop and the characters are printed on the screen. For printing a character printf() or putchar() macro can be used. Above program use putchar().

However, there is a special kind of array of characters. String is an array of characters terminated

Scanned by CamScanner

with null character. The null character is simply character with ASCII value 0 and it can be written as '\0'. Thus above program and figure can be modified as follows :

Program 7.10 : Strings

```
#include <stdio.h>
int main()
{
  int i;
  char arr[] = {'W', 'i', 'M', 'C', '\0'};/*declare and initialize array*/
  char str[] = "WiMC";
  printf("arr size = %d\n", sizeof(arr));                    /*output: 5*/
  printf("str size = %d\n", sizeof(str));                    /*output: 5*/
  for(i=0; arr[i]!='\0'; i++)
     putchar(arr[i]);                        /*print characters one by one*/
  for(i=0; str[i]!='\0'; i++)
     putchar(str[i]);                        /*print characters one by one*/
  return 0;
}
```

This program declares two character arrays *arr* and *str*. *arr* is initialized just like simple array, while *str* is initialized using string constant. The string constants (enclosed into double quotes) always have an invisible null character at its end, so size of str is 5 bytes. Also note that, both of these arrays are accessed in exactly same way using a loop. The end condition of the loop is based on null character instead of length of the array, because strings always terminate with null character. The physical layout of both the arrays in memory will be same and is shown in figure 7.4.

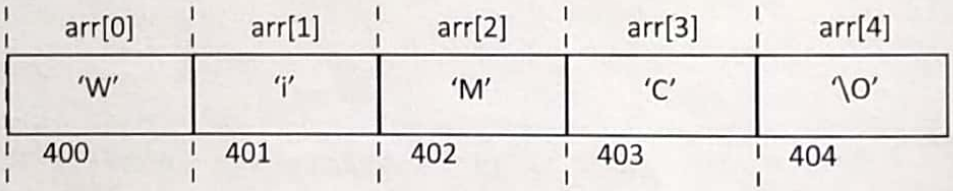


Figure 7.4 : String

Even though the concept of string is much similar to the 1–D array, the programming differs a lot due to wide range of string functions present into standard C library. There is a dedicated header file for string i.e. string.h. The prototypes of most widely used functions are given into appendix @ and use of few functions is discussed ahead in this chapter.

String input and output

String can be scanned from the user using scanf() with %s format specifier, However, by default scanf() is terminated by white space. In other words, using %s format specifier any multi word string cannot be scanned. Therefore the format specifier has been modified. There are two more format specifiers given. The first one defines a scan set and can be used to accept a given set of characters e.g. %[a–z] (scan only small case letters), %[a–z A–Z](scan all alphabets and space), %[12345] (scan any digit out of 1,2,3,4,5), etc. Thus scanf() will scan anything enclosed into %[...]. The second format specifier defines a delimiter and can be used to deny any character in the given set of characters, all characters before that will be scanned e.g. [^\n] (scan all characters till \n is entered), [^0–9] (scan all character till any digit is input), etc.

Scanned by CamScanner

A new function is introduced to scan full line of input i.e. gets(). It reads number of characters given by user till new line character. The typical examples of scanf() and gets() can be given as :

```
/*char str[64];*/
scanf("%s", str);              /*scans first word of given input*/
scanf("%[^\n]", str);              /*scans full line (till \n)*/
scanf("%[a-z A-Z]", str)       /*scans all alphabets and space*/
 gets(str);                        /*scans full line (till \n)*/
```

Note that in scanf() *str* is passed as argument not *&str*. This is simply because, scanf() needs address of the variable and *str* itself represents base address of the array.

String can be printed using printf() with format specifier or puts() function. Both of them print the string till '\0' character. The puts() prints a string followed by a new line character on the screen. The typical examples of printf() and puts() can be given as :

```
/*char str[64];*/
/*initialize string here*/
printf("%s", str);          /*prints string*/
puts(str);                  /*prints string followed by new line \n*/
```

Note that printf() and puts() take the base address of the string to be printed.

Pointer to String

Like pointer to 1–D array, pointer to string can be declared. Obviously the pointer declared will be char pointer and will keep the address of first character. Also array name will represent address of 0th element and can be used as pointer. The following example shows a simple example of pointer to string.

Program 7.11 : Pointer to String

```
#include<stdio.h>
int  main()
{
     char arr[5]="WiMC";                          /*string declaration*/
     char *ptr;                                  /*pointer declaration*/
     int i;
     ptr = arr;                                /*pointer initialization*/
     for(i=0; ptr[i]!='\0'; i++)
          putchar(ptr[i]);                        /*using array notation*/
     for(i=0; *(ptr+i)!='\0'; i++)
          putchar(*(ptr+i));                   /*using pointer notation*/
     while(*ptr!='\0')
     {
          putchar(*ptr);                       /*using pointer notation*/
          ptr++;                                    /*increment pointer*/
     }
     return 0;
}
```

The above example is self explanatory and based on pointer arithmetic. It prints WiMC three times on the screen using different notations each time. The diagrammatic view of the pointer to

Scanned by CamScanner

string can be seen in figure 7.5 given below.

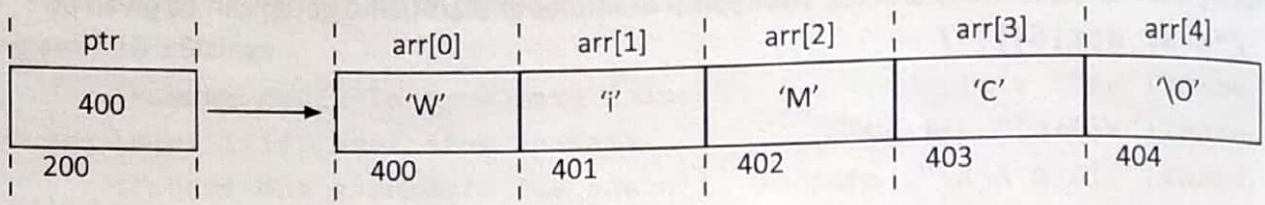


Figure 7.5 : Pointer to String

Passing string to function

Strings can be passed to the function much like 1–D array. The following program tries to pass string to function. Same function is written in two different ways. The first version implements functionality using array notation, while second implements the same functionality using pointer notation. Understand the following program.

Problem : Write a function to count the number of spaces in the string.

Program 7.12 : Passing String to Function

```
#include <stdio.h>
int count1(char a[]);
int count2(char *a);
int main()
{
	char str[64] = "sunbeam infotech, pune, karad";
	int cnt;
	cnt = count1(str);
	printf("count : %d\n", cnt);  /*output: 4*/
	cnt = count2(str);
	printf("count : %d\n", cnt);  /*output: 4*/
	return 0;
}
/*function using array notation*/
int count1(char a[])
{
	int i, cnt=0;
	for(i=0; a[i]!='\0'; i++)
	{
		if(a[i]==' ')
			cnt++;
	}
	return cnt;
}
/*function using pointer notation*/
int count2(char *a)
{
	int i, cnt=0;
	for(i=0; *a!='\0'; i++)
	{
```


Scanned by CamScanner

```
        if(*a== '  ')
            cnt++;
        a++;
    }
    return cnt;
}
```

String Library Functions

There are many pre defined functions available in C library for string manipulation. Most of them are mentioned in appendix @. The following program helps to understand syntax of these functions.

Program 7.13 : String Library Functions

```
#include <stdio.h>
#include <string.h> /*header file for string functions*/
int main()
{
   char a[64], b[64], *p, ch;
   int choice, temp;
   do
   {
      printf("\n\n1.string length\n2.string copy\n3.string concat\n");
      printf("4.string compare\n5.search character\n");
      printf("0.exit\nenter choice:");
      scanf("%d", &choice);
      if(choice==0)
        break;
   printf("enter a string :    ");
   fflush(stdin);
   scanf("%s", a);
   switch(choice)
   {
   case 1: /*string length*/
      temp = strlen(a);                    /*length = strlen(string);*/
      printf("length : %d\n", temp);
      break;
   case 2: /*string copy*/
      strcpy(b, a);                /*strcpy(dest_string, soruce_string);*/
      printf("copied string : %s\n", b);
      break;
   case 3: /*string concat*/
      printf("enter string to append : ");
      scanf("%s", b);
      fflush(stdin);
      strcat(a, b);                /*strcat(dest_string, soruce_string);*/
      printf("modified string : %s\n", a);
      break;
```


Scanned by CamScanner

```
    case 4 /*string compare*/
       printf("enter second string : ");
       fflush(stdin);
       scanf("%s", b);
       temp = strcmp(a, b);     /*difference=strcmp(string1, string2);*/
       printf("difference : %d\n", temp);
       break;
    case 5: /*search character*/
       printf("enter character to be searched :     ");
       fflush(stdin);
       ch = getchar();
       p = strchr(a, ch);                          /*ptr=strchr(string, ch);*/
       if(p==NULL)
       printf("character not found\n");
       else
       printf("character found at index %d\n", p-a);
       break;
    }
    }
  while(choice!=0);
  return 0;
}
```

There are so many library functions for string manipulation. There documentation can be found in the help documents associated with compilers. The program 7.13 gives demonstration of most widely used string library functions strlen(), strcpy(), strcat(), strcmp() and strchr(). A brief idea of these functions is given below.

- **strlen()**

```
length = strlen(string);
```

This function takes string as an argument and returns its length. The length of the string is number of characters in the string excluding null character.

- **strcpy()**

```
strcpy(dest_string, source_string);
```

This function takes two string arguments. It copies source string (2nd argument) to the destination string (1st argument). One must ensure that destination string has enough space to accommodate source string.

- **strcat()**

```
strcat(dest_string, source_string) ;
```

This function takes two string arguments. It copies source string (2nd argument) at the end of the destination string (1st argument). One must ensure that destination string has enough space to accommodate combination of source string and destination string.

- **strcmp()**

```
difference = strcmp(string1, string2);
```

This function takes two string arguments. It compares two strings and returns difference between them as integer. The comparison is done on the basis of ASCII value of characters of these strings. If both the strings are same, *difference* will be 0. If *string1* is greater than *string2*, *difference* will be

Scanned by CamScanner

greater than zero. If *string1* is smaller than *string2*, *difference* will be less than zero. This comparison is case sensitive as it is based on ASCII values. Therefore, if *string1* is "sunbeam" and *string2* is "SunBeam", they are considered to be different. However, if strings are compared using stricmp(), the case is not considered. The syntax of stricmp() is exactly same syntax.

- **strchr()**

```
ptr = strchr(string, ch);
```

This function takes first argument as string and second as character to be searched in that string. It compares the character with each character in the string and address of first matching character will be returned. This address can be subtracted from the base address of array to find the index of that character. If character is not found it returns NULL pointer.

Pointer containing zero address is called as NULL pointer. NULL is a symbolic constant declared in header file stdio and it is #defined to 0. In the source code, one can use 0 instead of NULL, but it is advised to use NULL macro while comparing or assigning pointers to improve the readability of the program.

The NULL pointer is used by many library functions to indicate the failure e.g. strchr(), strstr(), fopen(), etc. It is always good practice to initialize pointer with NULL address instead of keeping garbage address into it.

To be specific, NULL pointer does not keep the address of RAM location 0. Internally it is mapped to certain unused address. One should take care of not reading or writing any value at this location. Attempting such thing will cause runtime error.

Array of pointers

1–D array can be of pointer type. Once again there is not much change in the syntax. The program 7.14 shows the example of array of character pointers and it can be visualized as shown it the figure 7.6. Each pointer keeps the address of one string.

Program 7.14 : Array of Pointers

```
#include <stdio.h>
int main()
{
    char *arr[] = {"SunBeam", "DAC", "WiMC", "Pune", "Karad"};
    char **ptr;
    int i;
    printf("size : %d\n", sizeof(arr));     /*prints 20*/
    /*accessing array of pointers using array notation*/
    for(i=0; i<5; i++)
        puts(arr[i]);
    ptr = arr;      /*initializing pointer*/
    for(i=0; i<5; i++)
    {
        puts(*ptr);     /*printing a string*/
        ptr++;     /*increment pointer to next element*/
    }
    return 0;
}
```

As shown in the above program, *arr* is array of 5 char pointers, so its size is 20 bytes on 32–bit compiler. Each element *arr[i]* keeps address of string constant shown in the figure 7.6. The *ptr* is pointer to this array i.e. first element of the array. Since first element of the array is array of char pointer, *ptr* must be char pointer to pointer so that it can hold the address of the element. Now *ptr[i]* or **(ptr+i)* or **ptr* represents address of char, which is passed to puts().

Scanned by CamScanner

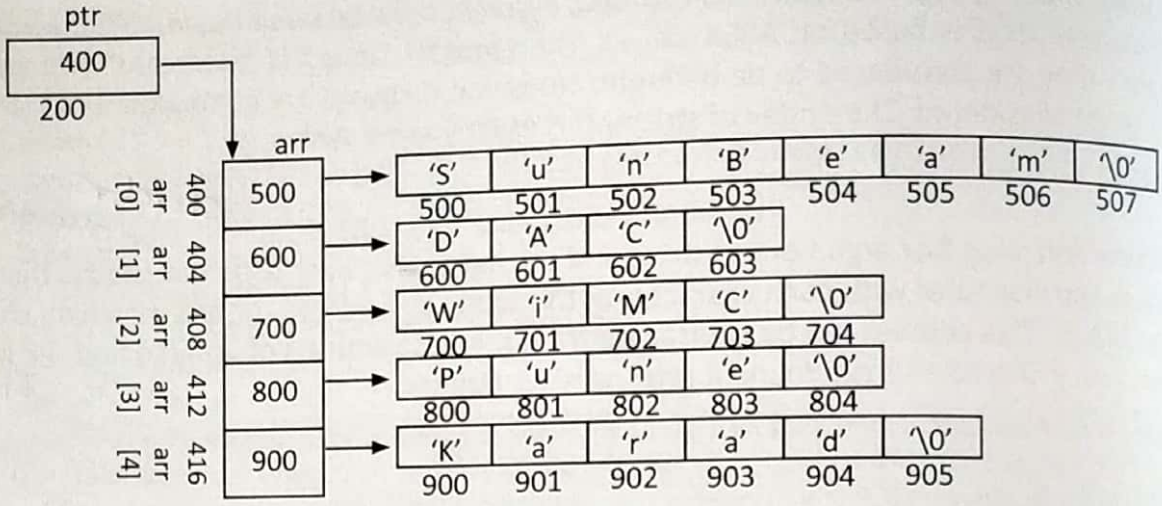


Figure 7.6 : Array of Pointers

Command line arguments

As discussed in the functions chapter, main() can take arguments.

```
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char *envp[]);
```

Note that both the *argv* and *envp* are array of character pointers. In this case, *envp* is environment parameters of the operating system. It contains information about type of operating system, system directory, path variable, etc.

argv contains the address of array of char pointers which stores information that is passed on command line while executing the program. *argv[0]* always contains the address of program name string as shown in figure 7.7. Also last address in this array of pointers is NULL. The *argc* represents number of arguments passed on command line including name of the program. The program 7.15 shows simplest example of command line arguments. Assuming that name of executable is *app.exe*, if program is run on command line as :

```
app.exe DAC WiMC SunBeam Karad
```

Then, *argc* will be 4 and array of char pointers pointed by *argv* contains total 5 elements including last NULL pointer. When program runs, all four strings including program name will be printed on screen. The figure 7.7 below shows the command line arguments in the example given.

Program 7.15 : Command Line Arguments

```
/*app.c*
/#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for (i=0; i <argc; i++)
       puts(argv[i]);
    return 0;
}
```


Scanned by CamScanner

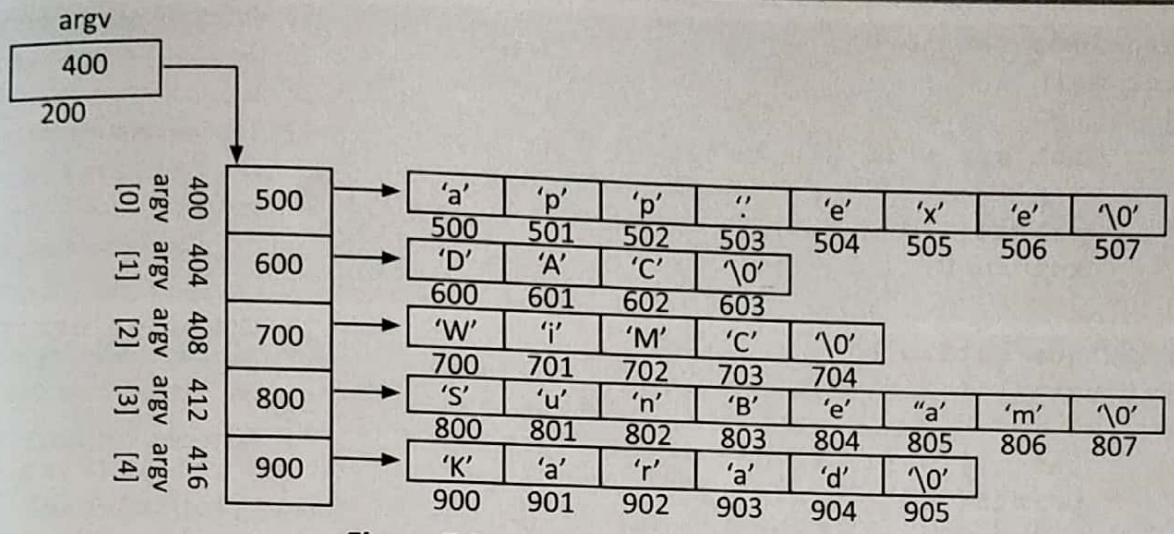


Figure 7.7 : Command Line Arguments

Lab Exercise

Q. 1. Write functions to input an array of integers and display and the same.

Q. 2. Write a function to find sum and average of elements in one dimensional array.

Q. 3. Write a function to find minimum and maximum value in one dimensional array.

Q. 4. Write a function to reverse an array of characters.

Q. 5. Write user defined string functions to simulate following library functions using array notation.

a. `int strlen(const char *s);`

b. `char *strcpy(char *destination, const char *source);`

c. `char *strchr(const char *s, int c) ;`

Q. 6. Rewrite functions in question 5 using pointer notation.

Q. 7. Write a program to accept a binary number in the form of string and convert in to integer.

Q. 8. Write a program to convert integer into string showing its hexadecimal representation.

Q. 9. Write a program to display all command line arguments except program name.

Q. 10. Write a program to implement four function calculator using command line arguments.

Objective Questions

Q. 1.

```
#include <stdio.h>
int main()
{
    int a[] = {1, 2, 3};
    printf("%d, %d, ", sizeof (a), sizeof (a[-1]));
    print_size(a);
    return 0;
}
int print_size (int a[])
{
printf("%d, %d, ", sizeof (a), sizeof (a[3]));
return 0;
}
```

1 6, 2, 6, 2
2) Compiler error
3) 12, 4, 4, 4
4) 6, 1, 6, 4

Scanned by CamScanner

Q. 2.
```
#include <stdio.h>
int main()
{
	int a[] = {1, 2, 3};
	++a;
	printf("%d", a[2]);
	return 0;
}
```
1) Compiler error
2) 3
3) 2
4) Linker error

Q. 3.
```
#include <stdio.h>
int main()
{
	int a[] = {1, 2, 3};
	printf("%d", a[a[1]]);
	return 0;
}
```
1) Compiler error
2) 3
3) 2
4) Linker error

Q. 4.
```
#include <stdio.h>
int main()
{
	int a[] = {1, 2, 3};
	f(a);
	return 0;
}
int f(int a[])
{
	++a; printf("%d", a[-1]);
	return 0;
}
```
1) 2
2) Compiler error
3) 0
4) 1

Q. 5.
```
#include <stdio.h>
#include <string.h>
int main()
{
	char *s = "SunBeam"; int i; char * p = s;
	for(i = 0; i <strlen(s); ++i, ++p)
		printf("%c", p[-i]);
	return 0;
}
```
1) maeBnuS
2) SSSSSSS
3) SunBeam
4) Snem

Q. 6.
```
#include <stdio.h>
#include <string.h>
int main()
{
	char *s = "SunBeam"; int i; char * p = s;
	for(i = 0; i <strlen(p); ++i, ++p)
		printf("%c", *p++);
	return 0;
}
```
1) SunBeam
2) Sne
3) Snem
4) SunB

Scanned by CamScanner

Q. 7.
```
#include <stdio.h>
int main()
{
  char *s = "SunBeam";char p[7] = "SunBeam";
  printf("%d %d, %d, %d", sizeof(s), sizeof(p),
          sizeof(*p), sizeof("sunbeam"));
  return 0;
}
```
1) 7, 7, 7, 7
2) 7, 7, 1, 8
3) 6, 7, 1, 8
4) 4, 7, 1, 8

Q. 8.
```
#include <stdio.h>
int main(int argv, char *argc[])
{
  printf("%d, %d, %d," *argc[argv],
  *argv[argc],*(argc + argv), *(argv + argc));
  return 0;
}
```
1) 0, 0, 0, 0
2) Compiler error
3) none
4) depends on arguments

Q. 9.
```
#include <stdio.h>
int main()
{
     char *arr[] = {"SunBeam", "DAC",
          "WiMC", "Pune", "Karad"};
     char **ptr = arr + 2;
     printf("%s", ++ptr[arr-ptr] - 1);
     return 0;
}
```
1) Karad
2) SunBeam
3) DAC
4) WIMC

Q. 10.
```
#include <stdio.h>
int main()
{
     char *arr[5] = {"SunBeam"};
     char **ptr = arr + 2;
     printf("%d", *ptr);
     return 0;
}
```
1) garbage
2) 117
3) 0
4) 110

Q. 11.
```
#include <stdio.h>
int main()
{
     static char *s = "SunBeam"; char ch = *s;
     if(*s){++s; main(); printf("%c", ch); }
     return 0;
}
```
1) SunBeam
2) maeBnuS
3) Compiler error
4) SSSSSSS

Scanned by CamScanner

Chapter 8 : Multidimensional Arrays

Last chapter has done detailed discussion on 1–D arrays of different types. However, array can have more than one dimension. The multidimensional arrays can be used to combine group of similar arrays. For example, marks of 6 subjects can be stored in 1–D array of *int* type. However, such collection of marks of top 5 students of one classroom can be combined into a two dimensional array of order 5x6. And such collection data of students in 3 classrooms can be combined in a three dimensional array of order 3x5x6. Theoretically array can have any number of dimensions. However, array with more than two dimensions is rarely used. This chapter discuss 2–D array in enough depth while introduction to 3–D array has been covered.

2–D Array

The array with two dimensions can be called as 2–D array or matrix. This array can be considered to be made of rows and columns. Like a matrix, array of 2x3 has two rows and three columns. The logical representation of 2–D array is shown in figure 8.1.

	Col 0	Col 1	Col 2
row 0	1.1	2.2	3.3
row 1	4.4	5.5	6.6

Figure 8.1 :2–D Array Logical Layout

Assuming that *arr* is name of array, the elements can be accessed in the form *arr[row][col]*. Here row represents *row* index and *col* represents column index. The indexing always starts from zero like 1–D array. Thus arr[0][2] represents the element in 0th row and 2nd column i.e. 3.3.

However, the diagram does not show the real picture of 2–D array in memory. To understand memory layout of the array, one must be comfortable with the definition of array. Array is defined as collection of *similar* data elements in *consecutive* memory locations. These data elements can be any built in type or any user defined type. Even these elements can be arrays. 2–D array is defined as array of 1–D arrays of some type. Figure 8.2 show array of *double* data type, which contains two 1–D arrays of three double elements. Both these arrays are shown as arr[0] (first element of 2–D array) and arr[1] (second element of 2–D array) and their addresses are 400 and 424 respectively. The arr[0] itself is an array of three *double* elements, its elements are shown as arr[0][0] (first element of arr[0]), arr[0][1] (second element of arr[0]) and arr[0][2] (third element of arr[0]) and their addresses are 400, 408 and 416 respectively.

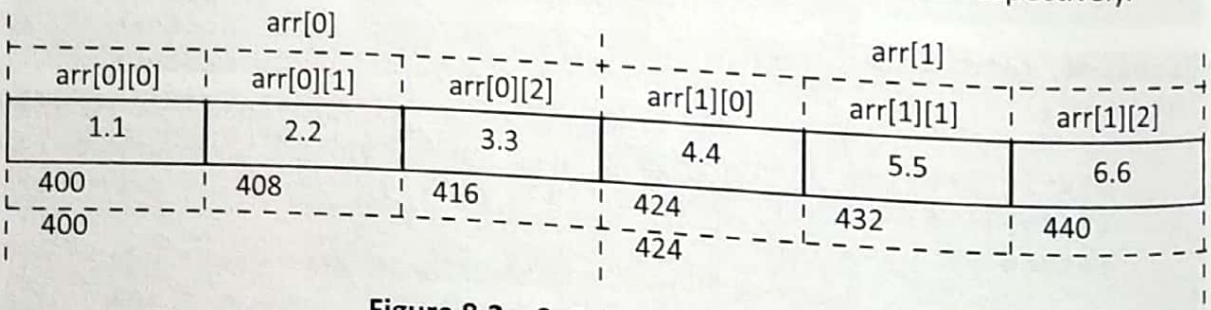


Figure 8.2 : 2–D Array Physical Layout

2–D array basics

As discussed, 2–D array is used to group 1–D arrays. The following example will help to understand

Scanned by CamScanner

declaration, initialization and accessing the elements of a 2–D array.

Problem : Take the marks of 6 subjects per student from the user. Store data of 4 students and finally display marks for all subjects per student along with its sum per student.

Program 8.1 : 2–D Array

```
#include <stdio.h>
int main()
{
    int marks[4][6];    /*marks for 4 students x 6 subjects*/
    int sum;
    int i, j;
    /*scan marks for all students*/
    for(i=0; i<4; i++)
    {
        printf("student %d\n enter the marks : ", i+1);
        for(j=0; j<6; j++)
            scanf("%d", &marks[i][j]);
    }
    /*print marks for all students*/
    for(i=0; i<4; i++)
    {
        sum = 0;
        printf("\n student %d :: marks : ", i+1);
        for(j=0; j<6; j++)
        {
            printf("%6d", marks[i][j]);
            sum = sum + marks[i][j];        /*sum marks per student*/
        }
        printf(" = %d", sum);
    }
    return 0;
}
```

Array declaration

Syntactically two–dimensional array is declared as :

data–type variable–name[rows][cols];

The above declaration means, the array with name *variable–name* can store *rows*cols* number of elements of type *data–type* in consecutive memory locations. In above example array *marks* can store marks of 4 students. For each student marks of 6 subjects are stored. Since array is declared locally and it is not initialized, all elements will by default get garbage value.

Array initialization

Array can be initialized at the point of declaration just like 1–D array. For example :

```
int a[2][3] = {11, 22, 33, 44, 55, 66};
int b[][3] = {11, 22, 33, 44, 55, 66};
int c[2][3] = {{11, 22, 33}, {44, 55, 66}};
int d[2][3] = {{11, 22}, {33, 44}};
```


Scanned by CamScanner

The array *a* is declared of order 2x3. The elements are given in curly braces. This initializes each element in the array from left to right fashion. For array *b* number of rows is not declared, However, 6 elements are given in curly braces and number of columns are given three, so number of rows in b is considered as 2. Thus if 2–D array is initialized at the point of declaration, giving number of rows of array is optional; in this case it will be automatically calculated by the compiler. However, giving number of columns is mandatory, as it is internally used for pointer arithmetic while accessing its elements.

The array *c* is declared similar to array *a*. But elements for each row are included into additional pair curly braces to improve the readability.

The size of array *should be* declared as 2x3, but only 4 elements are initialized. In fact, one element in each row is missing. In this case last element of each will be initialized to zero. Thus is array is initialized partially at its point of declaration, remaining elements are initialized to zero. Also array can be initialized by initializing each member of the array. In above example, all elements are initialized by scanning values from the user using scanf(). The use of nested loops is done to access all elements of the array one by one. Outer loop variable *i* represents row, while inner loop variable *j* represents column. Note that *& marks[i][j]* represents address of individual element in the array.

Accessing and printing array elements

It is very common practice to use nested loops to access multidimensional array. As discussed just now, for 2–D arrays two loops are used, in which outer loop variable *i* represents row, while inner loop variable *j* represents column. In above example, similar nested loop is used for accessing array elements *marks[i][j]* while summing it and in the same loop, all the elements are printed using printf() statement. This method will access array elements in row wise fashion i.e. elements of 0th row are read first, then 1st row and so on.

2–D array and pointers

2–D arrays are simple array of 1–D array. So just like 1–D array, to understand 2–D arrays well, one must have enough knowledge of pointers.

Pointer to Array

The pointer to array means the pointer variable storing the address of very first element (index 0) of the array. However, element arr[0] itself is an 1–D array (program 8.2). So the pointer variable must be declared so that it points to entire 1–D array.

Program 8.2 : Pointer to 2–D array

```
#include <stdio.h>
int main()
{
  double arr[2][3] = {{1.1, 2.2, 3.3},
                      {4.4, 5.5, 6.6}}; /*array declaration    */
  double (*ptr)[3];                     /*pointer declaration   */
  ptr = arr;                            /*pointer initialization */
  printf("%u %u %u\n", ptr, ptr+1, *(ptr+1));          /*prints 400 424 424*/
  printf("%u %.11f\n", *(ptr+1)+2, *(*(ptr+1)+2));     /*prints 440 6.6*/
  return 0;
}
```

In above program double pointer *ptr* keeps the address of first element of *arr[0]*, so it can be considered as pointer to array *arr*. The diagram 8.3 is drawn to make the visual understanding of the

Scanned by CamScanner

program. The address of array and pointer are chosen arbitrarily. The expected output is written into comments of the program.

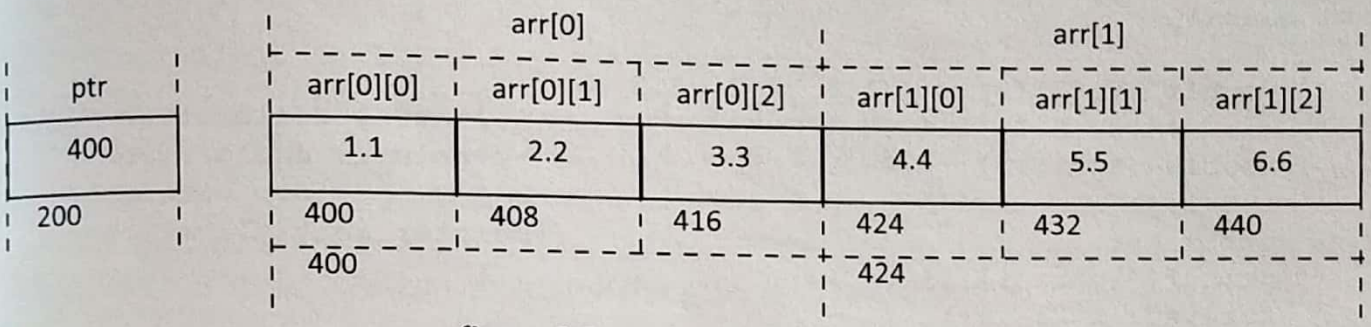


figure 8.3 : pointer to 2–d array

Note that pointer to array is declared with the complicated syntax.

*data–type (*variable–name)[cols];*

```
e.g. double (*ptr)[3];
```

Here *ptr* is pointer to the 1–D array of 3 elements of *double* type, which is 0th element of 2–D array. The size of pointer *ptr* is 4 bytes on 32–bit compilers. If this pointer is incremented it will get incremented by 24 bytes which is size of its data type (i.e. array of 3 double elements). The output of the program can be analyzed as follows :

Table 8.1 : accessing 2–D array through pointer

Expression	Description
`ptr = 400`	address of first element of 2–D array
`ptr + 1 = 424`	scale factor of the pointer is 24
`(ptr + 1) = 424`	access (index wise) 1st element of 2–D array i.e. 2nd *double* array base address
`*(ptr+1) + 2 = 440`	base address of double array + size of 2 *double* elements
`*(*(ptr+1) + 2) = 6.6`	value at the above address

Pointer Vs Base address

All the explanation given above is also applicable to array name, because it represents address of first element of the array. The previous chapter explains that :

```
arr[i] = *(arr + i)
```

This reminds that subscript notation is evaluated to pointer arithmetic expression. Extending the formula from the previous chapter for 2–D arrays new formula can be given as :

```
arr[i][j] = *(arr[i] + j) = *(*(arr + i) + j)
```

In this way array name can be used with the pointer notation and even pointer can be used with array notation. The following example program proves this :

Scanned by CamScanner

Program 8.3 : Pointer Vs Base address

```
#include <stdio.h>
int main()
{
	double arr[2][3] = {{1.1, 2.2, 3.3},
		{4.4, 5.5, 6.6}};                    /*array declaration */
	double (*ptr)[3];                        /*pointer declaration*/
	int i,j;                                 /*loop counter*/
	ptr = arr;                               /*pointer initialization*/
	printf("\narr[i][j] : ");
	for(i=0; i<2; i++)
	{
		for(j=0; j<3; j++)
			printf("%.1lf :", arr[i][j]);
	}
	printf("\n*(arr[i]+j) : ");
	for(i=0; i<2; i++)
	{
		for(j=0; j<3; j++)
			printf(":%.1lf ", *(arr[i]+j));
	}
	printf("\n*(*(arr+i)+j) : ");
	for(i=0; i<2; i++)
	{
		for(j=0; j<3; j++)
			printf("%.1lf ", *(*(arr+i)+j));
	}
	printf("\nptr[i][j] : ");
	for(i=0; i<2; i++)
	{
		for(j=0; j<3; j++)
			printf("%.1lf ", ptr[i][j]);
	}
	printf("\n*(ptr[i]+j) : ");
	for(i=0; i<2; i++)
	{
		for(j=0; j<3; j++)
			printf("%.1lf ", *(ptr[i]+j));
	}
	printf("\n*(*(ptr+i)+j) : ");
	for(i=0; i<2; i++)
	{
		for(j=0; j<3; j++)
			printf("%.1lf ", *(*(ptr+i)+j));
	}
	return 0;
}
```


Scanned by CamScanner

The output of the above program prints the array repeatedly as :

```
arr[i][j]   : 1.1 2.2 3.3 4.4 5.5 6.6
*(arr[i]+j)   : 1.1 2.2 3.3 4.4 5.5 6.6
*(*(arr+i)+j)   : 1.1 2.2 3.3 4.4 5.5 6.6
ptr[i][j]   : 1.1 2.2 3.3 4.4 5.5 6.6
*(ptr[i]+j)   : 1.1 2.2 3.3 4.4 5.5 6.6
*(*(ptr+i)+j)   : 1.1 2.2 3.3 4.4 5.5 6.6
```

Once again remember that, even though pointer and array can be used with alternative notations, both are not same. The difference in array and pointer is already discussed in previous chapter.

Passing 2–D array to function

Array can be passed as an argument to function. To pass the array to the function array name (i.e. base address of the array) is passed as parameter. Since it represents address of the first element, it can be accepted into the pointer. However, the formal argument can be declared in two different ways. The typical declaration of function will be one of the follows :

```
void print(double (*a)[3], int rows, int cols);
void print(double a[][3], int rows, int cols);
```

Understand that both of the above are equivalent and can be used alternatively. In both the cases *a* is a pointer that keeps the address of first element of the 2–D array only syntax is different. However, second syntax can be preferred to represent that array is passed as an argument. Using any of the declaration of the function will not make any difference in the implementation of the function.

The following program can be used to understand the syntax of passing array to the function and accessing it into body of the function.

Program 8.4 : Passing 2–D Array to Function

```
#include <stdio.h>
void input(int a[][6], in rows, int cols);
void display(int a[][6], int rows, int cols);
int main()
{
    int marks[4][6];          /*marks for 4 students x 6 subjects*/
    input(marks, 4, 6);            /*scan marks for all students*/
    display(marks, 4, 6);         /*print marks for all students*/

    return 0;
}
void input(int a[][6], int rows, int cols)
{
    int i, j;
    for(i=0; i<rows; i++)
    {
        printf("\n student %d\n enter the marks : ", i+1);
        for(j=0; j<cols; j++)
            scanf("%d", &a[i][j]);
```


Scanned by CamScanner

```
    }
}
void display(int a[][6], int rows, int cols)
{
    int i, j;
    for(i=0; i<rows; i++)
    {
        printf("\n student %d :: marks : ", i+1);
        for(j=0; j<cols; j++)
        {
            printf(":%6d", a[i][j]);
        }
    }
}
```

Note that, even though argument *a* of the function is a pointer, it is accessed using array notation. This is possible due to the fact that, subscript notation can be used on pointer to the array. The same program can modified to use pointer notation, However, array notation is much readable as compared to pointer notation.

3–D Arrays

The array of 2–D arrays is called as 3–D arrays. In other words, 3–D array is finite collection number of 2–D arrays in consecutive memory locations. The 3–D arrays are rarely used. The simple example of 3–D array is discussed here just to introduce its syntax and concept.

Program 8.5 : 3–D Array

```
#include <stdio.h>
int main()
{
    int a[2][3][2] = {
        {   /*0th 2-D array*/
            {1, 2},   /*0th 1-D array in 0th 2-D array*/
            {3, 4},   /*1st 1-D array in 0th 2-D array*/
            {5, 6},   /*2nd 1-D array in 0th 2-D array*/
        },
        {   /*0th 2-D array*/
            {7, 8},   /*0th 1-D array in 1st 2-D array*/
            {9, 10},  /*1st 1-D array in 1st 2-D array*/
            {11, 12}, /*2nd 1-D array in 1st 2-D array*/
        }
    };
  int i, j, k;
  for(i=0; i<2; i++)    /*a[i] is 2-D array within 1-D array*/
  {
    for(j=0; j<3; j++) /*a[i][j] is 1-D array within 2-D array a[i]*/
    {
```


Scanned by CamScanner

```
            for(k=0; k<2; k++)/*a[i][j][k] is int with 1-D array a[i][j]*/
            {
                printf("%5d", a[i][j][k]);
            }
        }
        printf("\n");
    }
    return 0;
}
```

The output of the above program will be printed as :

```
1   2   3   4   5   6
7   8   9   10  11  12
```

The declaration of 3–D array is done as :

int a[2][3][2];

Here *a* is array of 2 two dimensional array, in which each array is of size 3x2. The program can be easily understood if one can visualize the logical layout of the 3–D array as given in following figure, which try to show two 2–D arrays one after another.

	a(0)		a(1)	
0	1	2	7	8
1	3	4	9	10
2	5	6	11	12
	0	1	0	1

Figure 8.4 : 3-D array

The loop written to print all the values is self explanatory. The element a[i][j][k] represents the *int* element into the array. For example a[1][2][1], first access a[l] i.e. 1st 2–D array. a[1][2] access 2nd row into the 1st 2–D array. Finally a[1][2][1] access 1st element into that 2nd row. Note that all this explanation is given keeping zero based index in mind.

Finally since arrays are closely related with the pointers, the 3–D array can be accessed using the pointer notation on the array name. The same concept has been discussed in much depth with 1–D and 2–D arrays.

So the basic formula *a [i] = *(a+i)* can extended for 3–D arrays as follows :

```
a[i][j][k]
= *( a[i][j] + k )
= *( *( a[i] + j ) + k )
= *(*(*( a + i )+ j )+ k )
```

Thus replacing a [i] [j] [k] with *(*(*(a + i) + j) + k) in above example should give the same result. Now the same concept can be further extended for array of any dimensions.

Lab Exercise

Q. 1. Write functions to accept and display 2 dimensional arrays.

Q. 2. Write a function to display row wise and column wise sum of dimensional array.

Q. 3. Write a function to find minimum and maximum value in two dimensional array.

Q. 4. Write a function to accept 5 names and display the same.

Q. 5. Write a function to transpose a matrix.

Scanned by CamScanner

Objective Questions

Q. 1.

```
#include <stdio.h>
int main()
{
  int a[][]={{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};
  printf("%d", sizeof(a));
  return 0;
}
```

1) 3
2) Compiler error
3) 6
4) 9

Q 2.

```
#include <stdio.h>
int main()
{
  int a[][2]={ 1, 2, 3, 4, 5, 6};
  printf("%d", a[a[0][0]][a[0][0]]);
  return 0;
}
```

1) 4
2) Compiler error
3) 1
4) 2

Q 3.

```
#include <stdio.h>
int main()
{
  int a[][2]={ 1, 2, 3, 4, 5, 6};int *p = (int *)a;
  printf("%d", p[5] - p[1]);
  return 0;
}
```

1) 2
2) Compiler error
3) 4
4) 5

Q 4.

```
#include <stdio.h>
int main()
{
  int a[][2]={ 1, 9, 3, 4, 5, 6};int *p = (int *)a;
  int **pp = &p;
  *++*pp--; ++pp;
  printf("\n%u", *p - **pp);
  return 0;
}
```

1) 4
2) Garbage value
3) 8
4) 0

Q 5.

```
#include <stdio.h>
int main()
{
  char s[3][128] = {"SunBeam", "DAC", "WIMC"};
  char *p = s[1];
  s[0]=p;
  return 0; printf("%s",*p);
}
```

1) DAC
2) Compiler error
3) SunBeam
4) WIMC

Scanned by CamScanner

Q. 6.
```
#include <stdio.h>
int main()
{
  char s[3][128] = {"SunBeam", "DAC", "WIMC"};
  char *p = (char *)s[0];
  printf("%s", s[++p - s]);
  return 0;
}
```
1) DAC
2) Compiler error
3) SunBeam
4) WIMC

Q. 7.
```
#include <stdio.h>
int main()
{
  char s[3][128] = {"SunBeam", "DAC", "WIMC"};
  char *p = (char *)s[2];
  printf("%c%s", *p + *(p-1) - *p, ++p);
  return 0;
}
```
1) DAC
2) SDW
3) WIMC
4) SIMC

Q. 8.
```
#include <stdio.h>
int main()
{
  char s[3][128] = {"SunBeam", "DAC", "W I M C"};
  char *p = (char *)s[2];
  printf("%s", s[strlen(p) - strlen(s)]);
  return 0;
}
```
1) DAC
2) W I M C
3) SIMC
4) SunBeam

Q. 9.
```
#include <stdio.h>
int main()
{
  char s[3][128] = {"SunBeam", "DAC", "W I M C"};
  char *p = (char *)s[2] + 2;
  printf("%d", --*++p);
  return 0;
}
```
1) 72
2) 86
3) 76
4) 31

Q. 10.
```
#include <stdio.h>
int main()
{
  char s[3][9] = {"SunBeam", "DAC", "WIMC"};
  printf("%d, %d, %d", sizeof(s), sizeof(s[0]),
         sizeof(s[2][1]));
  return 0;
}
```
1) 27, 3, 1
2) 27, 2, 2
3) 27, 9, 1
4) 27, 9, 2

Scanned by CamScanner

Chapter 9 : Structures and Unions

As discussed in chapter 2, C has variety of data types. It includes not only built in data types, but also user defined data types. There are three user defined data types i.e. structures, unions and enumeration. This chapter explains structures and union. The enumerations are explained in the last chapter.

Structures

Many times there is requirement of keeping logically related data of more than one data type together. These types can be same or different. For example, book information contains name of book (string), name of author (string), price (double), number of pages (integer), etc. It is expected to keep entire book data in a single unit. Structure is a user defined type, which allows combining one or more logically related elements of same or different data types.

The syntax for structure will be clear from the following example :

```
struct book
{
        char name[20];
        char author[20];
        double price;
        short pages;
};
```

Note that structure is a data type. In this example "struct book" is a data type. Being a data type it should be used just as any other data type. For example, syntax for creating the variable as :

<data_type><varriable_name>;

So variable of structure can be created as :

```
struct book b1;
```

The size of any variable is decided by its data type. In case of structures each member of structure occupies certain memory and size of structure will be sum of size of each member. Thus size of the example book structure will be 50 bytes. Also note that structure variable is placed in consecutive memory locations (just like any other variable). The following diagram will help to understand memory organization of structure variable :

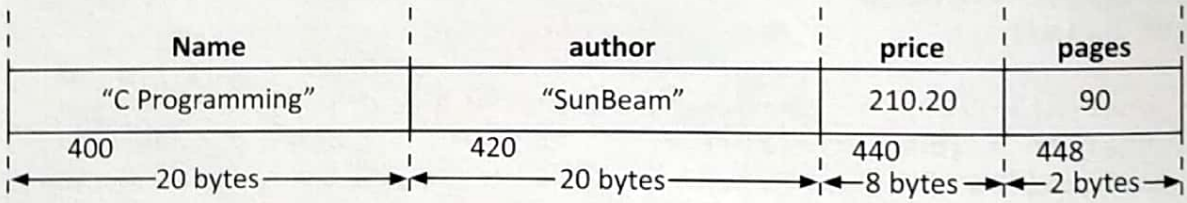


Figure 9.1 : Memory Layout of Structure variable

Declaring a structure variable was merely an example. But creating pointers, arrays of structures, passing structure by value and by address and almost all the syntaxes of structures will be exactly same as any other data type. All examples hereafter will be proving this thing only. Thing to understand is that, structure should be treated as any other data type.

The only new concept in structure is to define a new data type, accessing its members using dot(.) and arrow (–>) operator, nesting the structures and bitfields. Rest of all syntaxes are just a revision of all the things that are discussed previously.

The following program will cover the basic syntax of declaring, initializing, accessing members of structure :

Scanned by CamScanner

Program 9.1 : Introduction to Structure

```
#include <stdio.h>
#include <string.h>
/*declaring structure data type*/
struct book
{
   char name[20];
   char author[20];
   double price;
   short pages;
};
int main()
{
   struct book b1, b3;   /*declaring structure variables*/

   /*initializing structure variable while declaring it*/
   struct book b2 = {"c Programming", "Nilesh", 210.20, 90};

   /*accessing structure members and printing them*/
   printf("%s %s %.2lf %d\n", b2.name, b2.author, b2.price, b2.pages);

   /*scanning structure members and printing it back*/
   scanf("%s %s %lf %d", b1.name, b1.author, &b1.price, &b1.pages);
   printf("%s %s %.2lf %d\n", b1.name, b1.author, b1.price, b1.pages);

   /*initializing structure member wise and printing it back*/
   strcpy(b3.name, "c++ Programming");
   strcpy(b3.author, "Sandeep");
   b3.price = 200.40;
   b3.pages = 70;
   printf("%s %s %.2lf %d\n", b3.name, b3.author, b3.price, b3.pages);
   return 0;
}
```

The above program starts from the declaration of the structure. The declaration of any structure can be written either globally or within the function during declaration of the variables. Note that if structure is declared locally within the function, it cannot be accessed by any other function; so most of the times structures are declared globally. If using multiple files, it is the best practice to write the declaration into the header file and include the header file wherever needed.

Structure variables b1, b2 and b3 are declared in main(). Note that defining structure variables globally is not advised due to drawbacks of global variables discussed earlier in the chapter of storage classes. Also b2 variable is initialized at the point of declaration, where multiple values for the members are given in curly braces in the order of their declaration into the structure type.

Each structure member can be accessed using dot operator on its variable. The syntax for accessing structure member can be given as :

variable–name.member–name;

Scanned by CamScanner

From the above example program 9.1 it will be clear that accessing the variable for reading or assigning is done using the same syntax. Also note that, while scanning the structure member, address of operator is used except for the string types. Obviously string types do not need the address of operator, as name of array itself represent its base address.

While executing some of the structure programs in this chapter on few compilers like Turbo C, the runtime error may occur "Floating point formats not linked." This error is due to the problems that the compiler connot detect the use of floating point functions and hence they do not link the floating point library. This is typical case, if %f or % lf is used in printf() or scanf() function's format string.

Now to solve this problems there are multiple options as given below:

- Force compiler to link with floating point library by adding some dummy function into the source code

```
void dummy ()
{
float f= 1.0;
float *fp = &f;
}
```

- On Turbo C ++ 3.0 compiler, some pragma directive is available that can be used in the beginning source code file as :

```
extern unsigned_floatconvert;
#pragma extref_floatconvert
```

- The solution that can work on all the compilers is that, instead of directly scanning the values into float structure member, scan it into a local float variable and then assign to structure member.

The next program gives demonstration of declaring pointers, passing structure to function by value and by reference.

Program 9.2 : Passing struct to the Function

```
#include <stdio.h>
#include <string.h>
struct book
{
     char name[20];
     char author[20];
     double price;
     short pages;
};
void input(struct book *p);
void display(struct book b);
int main ()
{
     struct book b2, b1 = {"c Programming"};     /*initializing first member*/
     struct book *p = &b1; /*declaring and assigning structure variable*/
     /*assigning other members*/
     strcpy(p->author, "Nilesh");
     p->price = 210.9;
     p->pages = 90;
     printf("%s %s %.2lf %d\n", p->name, p->author, p->price, p->pages);
```


Scanned by CamScanner

```
    input(&b2);     /*pass by address*/
    display(b2);    /*pass by value*/
    return 0;
}
void input(struct book *p)
{
    scanf("%s %s %lf %d", p->name, p->author, &p->price, &p->pages);
}
void display(struct book b)
{
    printf("%s %s %.2lf %d\n", b.name, b.author, b.price, b.pages);
}
```

In the above example, pointer 'p' has been declared and assigned to a structure variable. Note that members of structures can be accessed through pointer using arrow operator (–>). Also structure variables can be passed by value or address to the function as shown in the above example.

> Passing the structure by value will create another copy of structure variable and this will need additional space on stack. This space can be significant if structure size is big enough (this is the usual case). Hence, even though structure can be passed by value or by address, it is always advised to be passed by address.

The following program shows how to declare and initialize array of structure and how to pass it to the structure :

Program 9.3 : Array of structures

```
#include <stdio.h>
struct book
{
    char name[20];
    int pages;
    double price;
};

void input(struct book *p);
void display(struct book *p);
void sort(struct book a[]);
int main()
{
    struct book arr[5]; /*declaring array of structure*/
    int i;
    printf("enter information of 5 books:");
    /*scanning all elements of array*/
    for(i=0; i<5; i++)
        input(&arr[i]);
    /*passing array to function*/
```


Scanned by CamScanner

```
    sort(arr);
    /*printing all elements of array*/
    for(i=0; i<5; i++)
        display(&arr[i]);
    return 0;
}
void input(struct book *p)
{
    scanf("%s %d %lf", p→name, &p→pages, &p→.price);
}
void display(struct book *p)
{
    printf("%s %d %.2lf", p→name, p→pages, p→price);
}
void sort(struct book a[])
{
    int i, j;
    struct book t;
    for(i=0; i<5; i++)
    {
        for(j=i+1; j<5; j++)
        {
            /*sorting on the basis of number of pages*/
            if(a[i].pages > a[j].pages)
            {
                /*swap a[i] and a[j]*/
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

The array of 5 structure variables is declared as *struct book arr[5];*. Now each element is structure i.e. *arr[i]* will be a structure variable of type *struct book*. Each member of this variable can be accessed by using dot operator on *arr[i]* or it can be passed to function by address as shown in above example. Also note that array of structure can be passed to function and received into function just like array of any other type. It means that, the base address of structure represented by array name *arr* is passed as actual argument and is received in a formal argument of pointer type *a*. As discussed in chapter 7, the formal pointer argument can be written in pointer notation or array notation (array notation is used in above program).

Scanned by CamScanner

Nested structures

Until now, in the example structure members in the example are either built-in type or derived type like strings. But actually structure can have member of user defined type i.e. struct or union. The following example will show how to declare structure variable of one type into other structure.

Program 9.4 : Nested structures

```
#include <stdio.h>
#include <string.h>
struct date
{
   short day, month, year;
};
struct student
{
   short roll;
      char name[20];
   struct date birth;
};
int main()
{
   struct student s1 = {424, "Nilesh", {28, 9, 1983}};
   struct student s2, s3;

   printf("%d %s", s1.roll, s1.name);
   printf(" %d-%d-%d\n", s1.birth.day, s1.birth.month, s1.birth.year);

   s2.roll = 32;
   strcpy(s2.name, "Ganesh");
   s2.birth.day = 12;
   s2.birth.month = 1;
   s2.birth.year = 1979;
   printf("%d %s", s2.roll, s2.name);
   printf(" %d-%d-%d\n", s2.birth.day, s2.birth.month, s2.birth.year);

   scanf("%d %s %d %d %d", &s3.roll, s3.name,
      &s3.birth.day, &s3.birth.month, &s3.birth.year);
   printf("%d %s", s3.roll, s3.name);
   printf(" %d-%d-%d\n", s3.birth.day, s3.birth.month, s3.birth.year);
   return 0;
}
```

In this example variable of struct date is used in struct 'student.' Figure 9.2 diagram shows how the memory is given for struct student variable. From the diagram it can be seen that, struct student has three members (roll, name, birth), where third member i.e. birth itself has three members (day, month, year). Thus total size of struct student will be 28 bytes.

Scanned by CamScanner

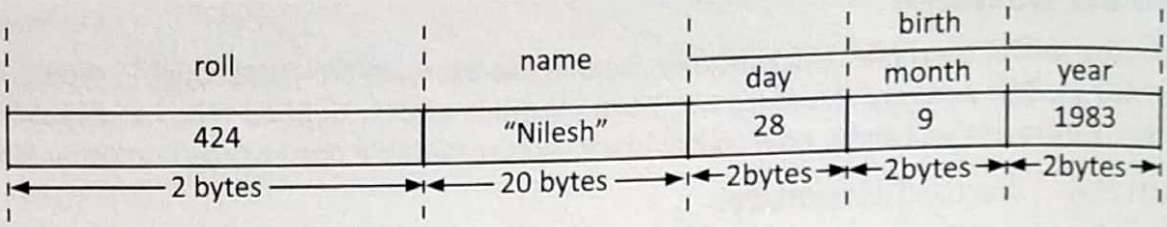


Figure 9.2 : Nested structures

Also note that structure variable of date is declared within structure of student just as other members i.e. no special syntax is used. However, there can be other way of declaring the same structures given below, where date structure is nested in student structure and its variable is declared along with declaration of structure itself.

```
struct student
{
      short roll;
      char name[20];
      struct date
      {
            short day, month, year;
      }birth;
};
```

While accessing nested structures, consecutive dot operators can be used. In the example, typical syntax used is

```
s1.birth.day = 12;
```

There is nothing great, as s1.birth access the birth member within s1 variable and s1.birth.day access day member within that birth. To access members via pointer to structure, arrow operator should be used at appropriate place. For example, accessing the day member through pointer to student structure can be done as :

```
p->birth.day = 12;
```

Anonymous Structures

```
struct
{
  char name[30];
     int age;
}a, *p;
```

It is possible to declare structure type without any name. In this case all variables or pointers of that structure type should be declared while declaring structure. The structure type is limited to scope of declaration. So such local structures cannot be accessed outside the block of function. In the example, *a* is declared as variable of anonymous structure and *p* is declared as pointer of anonymous structure type. Note that structure members can be accessed using structure variable *a* and pointer *p* with the similar syntax previously discussed.

Bitfields

The members of structure can have built-in type or user-defined type variables. Sometimes there can be need to store data using minimum memory as possible. This is generally requirement where program has to be ported on some kind of embedded system, where memory shortage is a common problem. To save the memory bitfields are used, where size of member can be given required number of bits instead of declaring as int or char. For example, age of a person can be stored as short int with the range up to 32767, while storing age needs maximum 7 bits with the range up to 127. Using a unsigned bitfield of 7 bits will be much memory efficient than using a short int of 16 bits.

Bitfields can be signed or unsigned. The range of numbers stored in unsigned bitfield of 'n' bits can be given as 0 to 2^n-1. For example : 7 bits will give range up to $2^7-1 = 127$. As expected, In case signed bitfield one bit is reserved for sign, hence its range can be given as -2^{n-1} to $+2^{n-1}-1$.

Scanned by CamScanner

The syntax for signed and unsigned bitfield is given below :

```
struct struct_name
{
     unsigned variable_name1:n;     /*n is number of bits*/
     signed variable_name2:n; /*n is number of bits*/
};
```

The following example stores information of person :

Program 9.5 : Bitfields

```
#include <stdio.h>
#include <string.h>
struct person
{
   char name[20];                                    /*store name*/
   unsigned age:7;                    /*store unsigned int 0 to 127*/
   unsigned gender:1; /*store boolean value 0 (female) or 1 (male)*/
};
int main()
{
   int age;
   char gender;
   struct person p1;
   printf("size of struct : %d\n", sizeof(p1));    /*size of struct : 21*/
     printf("enter person info name, age and gender (m/f):");
     scanf("%s", p1.name);                          /*scan name normally*/
     scanf("%d", &age);                  /*scan age in a variable and*/
     p1.age = age;                                  /*assigned to member*/

     fflush(stdin);
     scanf("%c", &gender);                           /*scan gender and*/
     p1.gender = (gender == 'm');                   /*assign to member*/
     /*print the information*/
     printf("%s %d ", p1.name, p1.age);
     p1.gender? printf("male") : printf("female");
     return 0;
}
```

The above program is more memory efficient; However, some complexity is increased due to limitations of bitfields. These limitations are listed below :

- The bitfields can be used only for integral data types; no provision is made from float or double.
- The array of bitfields cannot be created.
- The address of operator (&) cannot be used with bitfield member, so it cannot be directly scanned using scanf. However, it can be scanned in some other integer variable and then assign to the member as shown in the example.

Scanned by CamScanner

Unions

Union is another user defined data type, whose syntax is much similar to structure. Union is a user defined data type that allows combining more than one data types of same or different data type. The main difference between structure and union is their memory organization. The size of structure is sum of size of each member, while size of union is size of largest element. For the union, memory is shared among all the members of the union. Due to this any change made in one value of the member may affect the change into values of other members.

Unions are always used for specialized programming situations as follows :

The value need to be stored in one of the data types containing in union, as per the requirement. For example, result can be stored as average (double) for higher standards or as grade (char) for lower standards. The following example shows how to declare and use the union

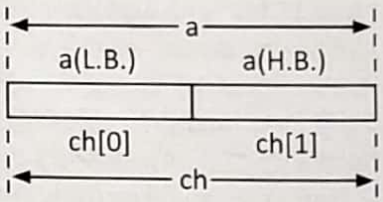
Note that, in such case one of the members of union is initialized and accessed whenever required.

Sometimes there is need to share same data in different forms. For example, some long variable can be accessed as one long or two integers or as four characters. The program 9.7 will illustrate this concept.

Program 9.6 : Union

```
#include <stdio.h>
union test
{
	short a;
	char ch[2];
};
int main()
{
	union test t;
	t.a = 65;
	t.ch[0] = 'z';
	printf("%d %c", t.a, t.ch[0]);
	/*output will be 90 z*/
	return 0;
}
```

The diagram shows memory organization for the union test in this example. Note that two bytes can be accessed as single variable of 2 bytes (short a;) or can be accessed as two variables (char ch[2];) each of 1 byte.



This kind of feature is many times used for system programming. The typical example is accessing 16 bit register (e.g. ax, bx, etc. in 86 architecture), as a pair of 8 bit registers (e.g. ah, al, bh, bl, etc.).

It is always a misconception saying that unions are used for saving memory. Always keep in mind that use of union is not very common programming situation. Before using union, understand its concept and need properly and then only use it as required.

Lab Exercise

Q. 1. Define a structure `person`, containing members' `panid, name, and phoneno.` Write functions to accept & display the structure.

Q. 2. Rewrite the above program to accept and display information of 5 `person s.`

Q. 3. Define structure `address` within structure `person`. The structure `address` consists of

Scanned by CamScanner

two address fields (`city` and `pincode`).

Q. 4. Write a function to sort array of person structure on the field pincode.

Q. 5 Define a structure point containing Cartesian coordinates x and y. Accept two points from user and find distance between them. The distance formula is :

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Objective Questions

What will the output of the following programs?

Q. 1.

```
#include <stdio.h>
int main()
{
  struct a
  {
    int a; struct b {int a; char b;} b;
  } x[] = {1, 'a', 'x', 'A', 4, 'b'};
  printf("%d", -x[1].a + x[0].b.a + x[1].b.a);
  return 0;
}
```

1) 28
2) Compiler error
3) garbage
4) 36

Q. 2.

```
#include <stdio.h>
struct student
{
     char *name[2];
     int s;
     char b;
}s = {"Sun", "Beam", 0, 1};
int main()
{
     printf("%d",sizeof(s));
     return 0;
}
```

1) 24
2) 11
3) 13
4) Compiler error

Q. 3.

```
#include <stdio.h>
struct node
{
     int head;
     struct node *n;
} ;
int main()
{
     struct node n;
     n.n = &n; n.n->head = 6;
     printf("%d",n.head);
     return 0;
}
```

1) address of n
2) garbage
3) 6
4) Compiler error

Scanned by CamScanner

Q. 4.

```
#include <stdio.h>
struct x
{
    int i; int j;int k;
};
int main()
{
    struct x *p, arr[3];
    p = &arr[0]; ++p;
    printf("%d", &(arr[0].j) - p);
    return 0;
}
```

1) -2
2) garbage
3) 2
4) Compiler error

Q. 5.

```
#include <stdio.h>
struct s1
{
struct{ int x; struct s{ int x;} s2; struct s
*s;} s3;
}y = {10};
int main()
{
    y.s3.s = &y.s3.s2;
    printf("%d", y.s3.s->x);
    return 0;
}
```

1) 10
2) garbage
3) 0
4) Compiler error

Q. 6.

```
#include <stdio.h>
int main()
{
    struct s1 {int i; };
    struct s2 {int i; };
    struct s1 st1;
    struct s2 st2;
    st1.i =5;
    st2 = st1; st1.i*=2;
    printf(" %d " , st2.i);
    return 0;
}
```

1) 5
2) garbage
3) 10
4) compiler error

Q. 7.

```
#include <stdio.h>
union u
{
    int x[3];
    char y[5];
}u[3];
int main()
{
    printf("%d", sizeof(u));
    return 0;
}
```

1) 24
2) 15
3) 36
4) 9

Scanned by CamScanner

Q. 8.

```
#include <stdio.h>
union u
{
     int *x; int y[2];
}u;
int main()
{
     int x = 10;
     u.y[1] = &x; u.y[0] = u.y[1];
     printf("%d", *u.x);
     return 0;
}
```

1) compiler error
2) garbage
3) abnormal termination
4) 10

Q. 9

```
#include <stdio.h>
struct s
{
     int x[10];
};
union u
{
     int *x;  struct s *y[2];
}u;
main()
{    printf("%d", sizeof(u));
     return 0;
}
```

1) compiler error
2) 20
3) 6
4) 8

Scanned by CamScanner

Chapter 10 : File Handling

File is collection of data or information on the secondary storage device. The data in primary memory (RAM) is volatile i.e. it is lost when computer shuts down. It is always expected to save the data for the future use. Such data is stored on secondary storage device like hard disks, compact disks, USB drives, etc in form of file. Any programming language must have the some feature for reading such data from the file or creating a new file. C language has number of standard library functions, which are dedicated to File I/O. The main file I/O operations are :

- create new file or open existing file
- closing the file
- writing data to the file
- reading data from the file

There are functions available for every operation mentioned above. For reading and writing the data, different kind of functions are available. The following figure 11.1 gives the classification of the File I/O functions in C library. One must be clear with the concept that C library functions do not perform I/O operations directly. However, they internally give call to I/O functions given by the operating system.

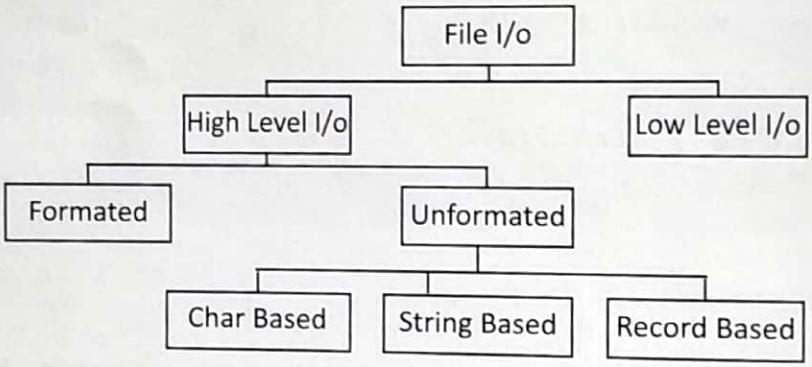


Figure 10.1 : File I/O Overview

Operating system has lots of modules inside it. The broad view of operating system split its functionality into two parts i.e. kernel and shell. The kernel of the operating system does the care functionalities of the system. These care functionalities include disk management, process management, memory management and hardware abstraction. Now the required tasks for the user (e.g. reading or writing files) of the system are exposed by the kernel in form of functions, which are called as *system calls*. The any programs running on the system including the shell, calls these functions and access kernel functionalities on the behalf of the end user. The shell provides the user interface for end user for accessing functionalities provided by the system. The shell can be text based in the operating systems like DOS, UNIX, etc. or it can be graphical in the operating systems like Windows, Linux, Mac OS, etc.

The figure 10.1 shows that File I/O can be done using C library functions (High Level functions) or system calls (Low Level functions). This book focuses on ANSI C programming, so cover only standard C library functions which are available on all the platforms.

Before starting with actual file handling functions in detail, the modes of files must be understood. The file can be opened in *text* mode or *binary* mode. The difference between both these modes is listed below :

Scanned by CamScanner

Table 10.1 : Text Mode Vs Binary Mode

Text Mode	Binary Mode
Stores the visual representation of data e.g. 12345 number occupies 5 bytes in the file (1 byte per character)	Stores memory representation of data e.g. 12345 number occupies 2 bytes in the file (short int needs 2 bytes)
The new line character is read as sequence of two characters i.e. \r\n (the way it appears on the screen)	The new line character is read as single character i.e. \n (the way it is stored in memory)
The end of file is represented by a special character with ASCII value 26 called as end of file character	The size of file is taken from file system to determine end of file, which is saved by operating system during creation of that file.

The text files are mainly used to store configuration settings or storing the data that is expected to be read from any simple text editor. The binary files are most commonly used files. All media files, executable files, etc are in binary format.

This was some theoretical discussion related to file handling. Now it is time to focus the practical aspect of file handling. The important file handling functions from C library are discussed below.

> The prototypes of file handling functions frequently includes data type const and void * is used to keep base address of string like char and const implies that this string will not be modified in the function. The void pointer is generic pointer and can take address of any data type of variable or array.

Opening and Closing File

There are two functions fopen() and fclose() for opening and closing the file respectively. The prototype of fopen() is given as :

FILE fopen(const char *filepath, const char *mode);*

The first argument of fopen() i.e. *filepath* represents path of the file to be opened or created. This path can be the absolute path or relative path. The second argument is *mode* that represents the mode for opening the file. The possible modes for opening the file are discussed in the following table.

Table 10.2 : Modes for fopen()

Mode	Description
w wb	File is opened for writing. If file does not exist, new file will be created. If file exists, all contents of the files will be deleted before opening the file. b represents binary mode.
r rb	File is opened for reading. If file exists, file is opened. If file does not exist, fopen() fails to open the file. b represents binary mode.
a ab	File is opened for writing at the end of the file. If file does not exist, new file will be created. If file exists, file is opened and new contents are appended to the existing contents in the file. b represents binary mode.
w+ wb+	File is opened for writing, also reading the data from the file is possible. Like w mode, if file does not exist, new file will be created. If file exists, all contents of the files will be deleted before opening the file. b represents binary mode.

Scanned by CamScanner

r+ rb+	File is opened for reading, also writing the data to the file is possible. Like r mode, if file exists, file is opened. If file does not exist, fopen() fails to open the file. b represents binary mode.
a+ ab+	File is opened for writing at the end of the file, also reading the data from the file is possible. Like a mode, if file does not exist, new file will be created. If file exists, file is opened and new contents are appended to the existing contents In the file. b represents binary mode.

Absolute path of any file is the full path of the file. Starting from the root of file system. For Linux it start from root/. eg./home/user/file.txt. Relative path is the path from the current directory from which the executable file is executed. Giving only file name instead of full path will cause to create file into the directory from which the executable file is executed.7

The actual file is kept into hard disk. fopen() internally call some operating system function (system call), which checks whether file exists and if required creates a new file or open the existing file. Internally, fopen() allocates some buffer in the RAM and load few contents or complete file (if it is too small in size) into this buffer. The other information of the file like mode of the file and file descriptor, etc along with base address of the buffer and current position pointer are kept in a structure called FILE structure and address of that structure is returned. This FILE pointer is required for rest of all functions of file handling. FILE structure is declared in *stdio.h* header file. The diagrammatic view of the fopen() is shown in figure 10.2. If fopen fails to open the file due to some reason (e.g. wrong file path, disk full, etc) fopen() returns NULL pointer.

As a good practice the FILE pointer opened by fopen() must be checked against NULL. If fopen() fails, this pointer will be NULL. Then appropriate action should be taken like showing error message to the user and terminate program.

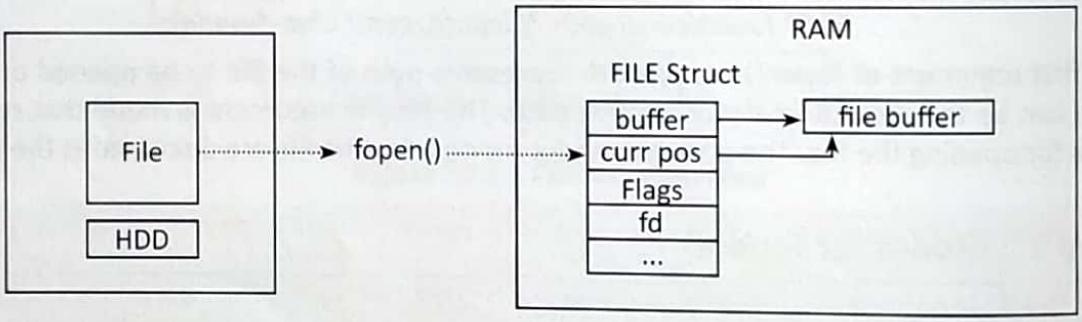


Figure 10.2 : Internals of fopen()

The file opened by fopen() is closed by fclose(). The prototype of fclose() is given as :

*int fclose(FILE *fp);*

fclose() flush any changes into the buffer associated with the FILE structure to the file on the disk and then deletes the FILE structure which is created by fopen() for that file. It returns EOF to indicate the error, otherwise it returns 0.

Reading or Writing File

The data can be written or read from the file using variety of functions. As figure 10.1 shows all

Scanned by CamScanner

the functions are classified into formatted or unformatted functions. The unformatted functions do not write the data in well formatted manner.

Character based I/O

There are mainly two functions in C library to write or read the data character by character from the file. These functions are fputc() and fgetc() which do write and read functions respectively.

The following program takes the characters from the one file using fgetc() function and then write it to another file using fputc(). Also note that source file must be opened in read mode, while destination file must be opened in write mode to write the data into the file.

Program 10.1 : Character Based I/O

```
#include <stdio.h>
int main()
{
  FILE *fpsrc, *fpdest;
  char ch, source[64], dest [64];
  /*step 1 : get name of source and destination files from user*/
  printf("enter source file name : ");
  gets(source);
  printf("enter destination file name : ");
  fflush(stdin);
  gets(dest);

  /*step 2 : open source and destination files */
  fpsrc = fopen(source, "r");  /*open source file in read mode*
  if(fpsrc==NULL)
  {
    printf("source file cannot be opened.");
    return 1;                          /*terminate program*/
  }
  fpdest = fopen(dest, "w");           /*open destination file in write mode*/
  if(fpdest==NULL)
  {
    printf("destination file cannot be opened.");
    fclose(fpsrc);                     /*close source file*/
    return 1;                          /*terminate program*/
  }

  /*step 3 : copy data from source file to destination file*/
  while((ch = fgetc(fpsrc))!=EOF)/*read the char from source file*/
  {                                    /*till end of file is not reached*/
    fputc(ch, fpdest);                 /*copy the char in destination file*/
  }
  /*step 4 : close both the files*/
  fclose(fpsrc);                       /*close source file*/
  fclose(fpdest);                      /*close destination file*/
  printf(:"file copied\n");
  return 0;
  }
```


Scanned by CamScanner

The above program is written in stepwise manner, so that it would be easy to understand it. The first step is quiet simple, as it takes path of source and destination files from the user. The second step tries to open source file, if the file cannot be opened then the program has been terminated. Note that main returns value 1 instead of 0 to represent the lack of resource. If source file is opened successfully then it tries to open destination file. If fopen() fails to open this file, then previously opened file must be closed and then program is terminated. If both files are opened successfully then source file must be read character by character mode using fgetc(). The prototype of fgetc() is :

*int fgetc(FILE*fp);*

fgetc() reads the character at the current position and returns its ASCII value. Also fgetc() increments current position pointer to the next character in the file buffer. So next call to fgetc() will read the next character from the buffer. If fgetc() reads at the end of file it returns –1 (or some negative number). This value is defined as EOF in the stdio.h. This character is written to destination file using fputc() function. Its prototype is given as :

*int fputc(int ch, FILE*fp);*

This function returns the ASCII value of the same character passed to it. Finally last step close both the files so that all changes made to destination files will be saved on the disk and also both FILE structures will be released from the memory.

String based I/O

Reading one character at a time is very flexible way of reading or writing data into the file. However, this is bit slower approach and also may be complicated in certain situations. The program given below repeats the same program but use string based I/O functions. The functions fgets() and fputs() is used to read and write a line of text respectively.

Program 10.2 : String Based I/O

```
#include <stdio.h>
int main()
{
    FILE *fpsrc, *fpdest;
    char buff[128], source[64], dest [64];
    /*step 1 : get name of source and destination files from user*/
    printf("enter source file name : ");
    gets(source);
    printf("enter destination file name : ");
    fflush(stdin);
    gets(dest);

    /*step 2 : open source and destination files */
    fpsrc = fopen(source, "r");  /*open source file in read mode*/
    if(fpsrc==NULL)
    {
       printf("source file cannot be opened.");
       return 1;                 /*terminate program*/
    }
    fpdest = fopen(dest, "w");   /*open destination file in write mode*/
    if(fpdest==NULL)
    {
```


Scanned by CamScanner

```
    printf("destination file cannot be opened.");
    fclose(fpsrc);/*close source file*/
    return 1;/*terminate program*/
  }

  /*step 3 : copy data from source file to destination file*/
  while(fgets(buff, sizeof(buff), fpsrc)!=NULL)  /*read line from
source file*/
  {                                    /*till end of file is not reached*/
  fputs(buff, fpdest);                /*copy the char in destination file*/
  }
  /*step 4 : chose both the files*/
  fclose(fpsrc);                          /*close source file*/
  fclose(fpdest);                      /*close destination file*/
  printf("file copied\n");
  return 0;
  }
```

The steps of the program can be found to be exactly same as program 10.1. Only difference is into the step 3, where functions fgets() and fputs() are used instead of fgetc() and fputc(). fgets() reads the string from the file. Its prototype is given as :

char fgets(char *buff, int size, FILE *fp);*

The first argument *buff* is string buffer of the size given as second argument *size,* where string from the file represented by *fp* will be read. It reads characters from the file till new line character is reached. However, it don't read more than the or *size–1* characters from the file. The last character in the string buffer is made null character. It returns address of string buffer *buff* where string is read. When fgets() reads a line, it moves current position pointer to the next character to be read. When fgets() reads at the end of the file, it returns NULL pointer. This string is then given to fputs() to write into destination file. The prototype of fputs() is :

*int fputs(const char *buff, FILE *fp);*

The function writes string *buff* into file associated with *fp*. It returns the ASCII value of the last character written to the file. This code works faster as compared to program 10.1.

Record based I/O :

Record based I/O is mainly used in binary files to store the structure variables into the files. The data is written into binary format, so it cannot be directly read using any text editor. The following program writes the structure data into the file using fwrite() function.

Program 10.3 : Writing Records to File

```
#include <stdio.h>
struct book
{
    char name[20];
    int pages;
    int price;
```


Scanned by CamScanner

```
};
int main()
{
      struct book data[4];
      int i;
      FILE *fp;
      /*step 1 : open the data file*/
      fp = fopen("data.dat", "wb");
      if(fp==NULL)
      {
            printf("cannot open data file");
            return 1; /*terminate the program*/
      }
      /*step 2 : input data from the user*/
      printf("enter information for 4 books : ");
      for(i=0; i<4; i++)
      {
            fflush(stdin);
            scanf("%s%d%d", data[i].name, &data[i].pages, &data[i].
price);
      }
      /*step 3 : write data to the file*/
      for(i=0; i<4; i++)
            fwrite(&data[i], sizeof(struct book), 1, fp);
      /*step 4 : close the file*/
      fclose(fp);
      printf("data written to file.\n");
      return 0;
}
```

In above program, step 1 creates the file and opens it in binary mode for writing. In step 2, information of 4 books is scanned from the user. The step 3 has code for writing the data Into the file using fwrite structure. The prototype of fwrite() is given as :

*size_t fwrite(const void *base, size_t size, size_tcount, FILE *fp);*

This function writes any block of data addressed by first argument *base*. This data can be array of *count* (third argument) number of records, where each record has size given by second argument *size*. If there is single variable count can be given as 1. The above code uses loop to write records one by one. However, this code can be replaced by the single line as :

fwrite(data, sizeof(struct book), 4, fp);

Remember that fwrite() returns the number of records written successfully into the file. So if the data is written properly, the return value of fwrite should be equal to *count*. Finally in step 4, file is closed

The data is now written into the file in binary format. This data cannot be read by using simple text editor. The following program reads this data from the file using fread() function and shows it on the screen.

Scanned by CamScanner

Program 10.4 : Reading Records from File

```
#include <stdio.h>
struct book
{
     char name[20];
     int pages;
     int price;
};
int main()
{
     struct book data[4];
     int i=0,x;
     FILE *fp;
     /*step 1 : open the data file*/
     fp = fopen("data.dat", "rb");
     if(fp==NULL)
     {
          printf("cannot open data file");
          return 1; /*terminate the program*/
     }
     /*step 2 : read data from the file*/
     while(1)
     {
          fflush(stdin);
          fread(&data[i], sizeof(struct book), 1, fp);
     if(x!=1)
          break;
     i++;
     }
     /*step 3 : write data to the file*/
     for(i=0; i<4; i++)
          printf("%s %d%d\n", data[i].name, data[i].pages, data[i].
price);
     /*step 4 : close the file*/
     fclose(fp);
     return 0;
}
```

The above program is much similar to program 10.3. Step 1 opens file in binary mode for reading. In step 2, information of 4 books is read from the file. The prototype of fread() is given as :

*size_t fread(void *base, size_t size, size_t count, FILE *fp);*

This function reads the block of data to some variable or array addressed by first argument *base*. It can read array of *count* (third argument) number of records from the file, where each record has size given by second argument *size*. To read the single record, count can be given as 1. When fread() reads a record, the current position pointer is moved to next record to be read. The above code uses loop to read data of structure variable one by one. However, this code can be replaced by the single line as :

Scanned by CamScanner

fread(data, sizeof(struct book), 4, fp);

Assuming b as variable of struct book the number of records can be read with using loop.

```
while (fread (&b) , sizeof (struct book),1, fp)==1)
{
   printf("  %s", b.name );
}
```

Remember that fread() returns the number of elements read successfully from the file. So if the data is read properly, the return value of fread() should be equal to *count*. The step 3 has code for writing the data on the console. Finally in step 4, file is closed.

Formatted I/O

The data can be stored in the file in well formatted manner. The functions fprintf() and fscanf() are used to write or read the data from the text file using format specifiers. These format specifiers are same as printf() and scanf() and are discussed in appendix @. The prototype for fprintf() and fscanf() is given as :

*int fprintf(FILE*fp, const char *format, ...);*

*int fscanf(FILE*fp, const char *format, ...);*

Note that fprintf() and fscanf() are similarly to printf() and scanf() respectively. Both of these functions take additional argument FILE *fp, which represents associated file. fprintf() returns number of characters printed on the console, while fscanf() returns number of fields scanned from file. When fscanf() read at the end of file it returns EOF. The next two programs do the same things as done in program 10.3 and 10.4. These programs use fprintf() and fscanf() instead of fwrite() and fread() respectively.

Program 10.5 : Write Record in Text File

```
#include <stdio.h>
struct book
{
    char name[20];
    int pages;
    int price;
};
int main()
{
    struct book data[4];
    int i;
    FILE *fp;
    /*step 1 : open the data file*/
    fp = fopen("data.dat", "wb");
    if(fp==NULL)
    {
      printf("cannot open data file");
      return 1;/*terminate the program*/
  }
```


Scanned by CamScanner

```
    /*step 2 : input data from the user*/
    printf("enter information for 4 books : ");
    for(i=0; i<4; i++)
    {
        fflush(stdin);
        scanf("%s%d%d", data[i].name, &data[i].pages, &data[i].price);
    }
    /*step 3 : write data to the file*/
    for(i=0; i<4; i++)
        fprintf(fp, "%20s%5d%5d%\n", data[i].name, data[i].pages,data[i].price);
    /*step 4 : close the file*/
    fclose(fp);
    return 0;
}
```

Program 10.6 : Read Records from Text File

```
#include <stdio.h>
struct book
{
    char name[20];
    int pages;
    int price;
};
int main()
{
    struct book data[4];
    int i;
    FILE *fp;
    /*step 1 : open the data file*/
    fp = fopen("data.dat", "rb");
    if(fp==NULL)
    {
      printf("cannot open data file");
      return 1; /*terminate the program*/
    }
    /*step 2 : read data from the file*/
    for(i=0; i<4; i++)
    {
      fflush(stdin);
      fscanf(fp, "%s%d%d", data[i].name, &data[i].pages, &data[i].price);
    }
    /*step 3 : write data to the file*/
    for(i=0; i<4; i++)
       printf("%s %d%d\n", data[i].name, data[i].pages, data[i].price);
    /*step 4 : close the file*/
    fclose(fp);
    return 0;
}
```

The step 2 and 3 in above program can be combined to read all records from the file as

```
while(fscanf(fp,"%s%d%d",b.name,&b.page,  &b.price)!=EOF)
    printf("%s \n",b.name);
```


Scanned by CamScanner

Random access

Many times there can be need of directly reading some record in the file. By default when any file is opened by fopen(), its current position pointer is at the beginning of the file. This position pointer is incremented internally with every call to read functions like fgetc(), fgets(), fscanf(), fread() or any write function. However, this position can be directly modified by a special function, fseek(). The prototype of fseek() is :

*int fseek(FILE*fp, long offset, int position);*

The first parameter is pointer to FILE structure *fp*, the second argument *offset* represents number of bytes the current position is to be moved from the *position* given by third argument. The *position* can take three values listed as follows.

Table 10.3 : fseek() parameter

Value	Symbolic Constant	Meaning
0	SEEK_SET	beginning of the file
1	SEEK_CUR	current position
2	SEEK_END	end of the file

The offset can be positive to move current position forward from the *position* or it can be negative to move current position backward from the *position*. One can get the offset of the current position from the start of the file using ftell() function. The few examples will help to understand the concept of fseek() better.

Table 10.4 : fseek() examples

function call	description
fseek(fp, 10L, SEEK_SET);	current position will be moved at 10 bytes from the start of file.
fseek(fp, –10L, SEEK_END);	current position will be moved at 10 bytes before the end of file.
fseek(fp, 10L, SEEK_CUR);	current position will be moved at 10 bytes ahead from the current position.
fseek(fp, –10L, SEEK_CUR);	current position will be moved at 10 bytes before the current position.

Assuming that the file contains 10 records of book structure used in above programs, the following program will give the demonstration of randomly reading the records from the file. The program and possible output is explained in comments. All output written here assumes that program is running on 32 bit compiler.

Program 10.7 : fseek () and ftell()

```
#include <stdio.h>
struct book
{
    char name[20];
    int pages;
    int price;
};
int main()
{
    FILE *fp;
    struct book b;
    long size = sizeof(struct book);
    fp = fopen("data.dat", "rb");    /*open file in binary mode*/
    if(fp==NULL)
```


Scanned by CamScanner

```
    {
        printf("error in opening file");
        return 1;/*terminate the program*/
    }
    printf("cur pos : %ld\n", ftell(fp));      /*prints 0*/
    fread(&b, size, 1, fp);     /*reads first record from file*/
    printf("cur pos : %ld\n", ftell(fp));      /*prints 28*/
    printf("%s %d%d\n", b.name, b.pages, b.price);/*prints first record*/
    fseek(fp, 4*size, SEEK_SET);/*moves current position to the start of fifth
record*/
    printf("cur pos : %ld\n", ftell(fp));      /*prints 112*/
    fread(&b, size, 1, fp);     /*reads fifth record from file*/
    printf("cur pos : %ld\n", ftell(fp));      /*prints 140*/
    printf("%s %d%d\n", b.name, b.pages, b.price);/*prints fifth record*/
    fseek(fp, -4*size, SEEK_END); /*moves current position to the start of seventh
record*/
    printf("cur pos : %ld\n", ftell(fp));      /*prints 168*/
    fread(&b, size, 1, fp);     /*reads seventh record from file*/
    printf("cur pos : %ld\n", ftell(fp));      /*prints 196*/
    printf("%s %d%d\n", b.name, b.pages, b.price); /*prints seventh record*/
    fclose(fp); /*close the file*/
    return 0;
}
```

There are many other functions available in standard C library. One line description for few functions can be found in the table below. For detailed description one should read the help document associated with the compiler.

Table 10.5 : File handling functions

function or macro	description
freopen()	Close existing file and open it again as another file.
feof()	Test for end of file for given FILE pointer
fflush()	Flushes the buffer associated with given FILE pointer
getc(), putc()	Macro implementations for fgetc() and fputc() respectively
getw(), putw()	Read or write a word value in binary mode
ferror()	Tests for the error and returns non–zero values as error codes
rewind()	Repositions the current position pointer at the beginning of the file
rename()	Renames a file
remove()	Deletes a file

Scanned by CamScanner

Standard Streams

Stream is continuous sequence of data of undetermined length. The C language is born from UNIX and in UNIX systems every device is considered as a file. These devices are accessible to C programs in form of stream. There are some standard streams that are by default open for any C program.

Table 10.6 : Standard Streams

Stream	Device (default)	Description
Stdin	keyboard	standard input
Stdout	console	standard output
Stderr	consloe	standard error

The statement fflush(stdin); used many times in this book to clear the buffer associated with standard input device. Any characters present in this buffer will be removed and then the new input values can be taken from the user.

In this chapter, the important topics related to file handling are covered. This can be the good start for any beginner. However, for more efficient file handling, one should learn low level file handling.

Lab Exercise

Q. 1. Write a program to accept a word from user and display all lines in the file containing the word along with line numbers.

Q. 2. Write a menu driven program to

a. Accept student data rollno, name, and marks from user.

b. Write structure student to a text file using formatted I/O.

c. Read structure student from a text file using formatted I/O.

Note : each time file is opened for write operation student record should be overwritten.

Q. 3. Rewrite the above program for binary files using direct I/O.

Q. 4. Modify the above program to append student records in the file.

Q. 5. Next version of above program should provide a facility to search and modify information a student on his/her roll no.

Q. 6. Write functions to read file line by line without using scanf or gets and display each line along with line no.

Objective Questions

What will the output of the following programs?

Q. 1.

```
#include <stdio.h>
int main()
{
    printf("%d, %d", NULL, EOF);
    return 0;
}
```

```
1) 0,    0
2) compiler error
3) 0,    -1
4) -1,   -1
```


Scanned by CamScanner

Q. 2.
```
void print_file(FILE *fp,char buf[BUFSIZE])
{
    while(fgets(buf,BUFSIZE,fp) !=EOF)
    {
        printf("\n%s",buf);
    }
    return 0;
}
```

1) print entire file
2) Compiler error
3) Print entire file and infinite loop
4) print last line

Q. 3. The statement
```
    fprintf(stderr, "%d", EOF);
```
prints

1) -1 on disk
2) compiler error
3) -1 on console
4) 0 on console

Q. 4. Return type for fgets and fscanf is _____ and _____

1) char *, char *
2) depends on arguments
3) int*, int
4) char *, int

Q. 5.
```
#include <stdio.h>
int main()
{
  printf("%d ", SEEK_SET|SEEK_END|SEEK_CUR);
  return 0;
}
```

1) Compiler error
2) 3
3) garbage
4) 0

Q. 6. Return type for fclose(File *fp) is _____

1) void
2) int
3) FILE
4) FILE *

Q. 7.
```
#include <stdio.h>
int main()
{
  int x;
  fprintf(stderr, "%d", fprintf(stdout,"%d");
  fscanf(stdin, "%d", &x));
  return 0;
}/* assume that user input as 369 */
```

1) Compiler error
2) Garbage and 369
3) 369, 369
4) 11

Scanned by CamScanner

Q. 8. **On success fread() returns _____**

1) Number of bytes read
2) Number of items read
3) Number of chars read
4) void

Scanned by CamScanner

Chapter 11 : Miscellaneous

After learning so many core features of C programming including control structures, function, storage classes, arrays and structures; still few more features of C are remaining to discuss. This chapter try to cover all remaining features one by one. The most of the features in this chapter are smaller size; However, they are important and cannot be ignored.

Function pointers

Function pointer is one of the most widely used concepts in system programming. Pointer to function keeps the starting address of the function in the text section and then function can be called using the pointer instead of its name. This feature is mainly required when it is expected that some other module (many times operating system) calls the function written in the source code. The *callback* functions in operating system concepts are called via function pointers.

> In typical cases, the address of some function is passed to the operating system and the operating system calls that function via address when required. This kind of function is called as *callback function*. In Win32 programming event driven mechanism and in UNIX signals are implemented in this way.

Syntactically declaration of function pointer can seem to be bit complicated. See the following example :

Program 11.1 : Function Pointers

```
#include <stdio.h>
void sum(int a, int b);
int main()
{
    int num1=12, num2=4;
    void (*ptr)(int, int);    /*declaring function pointer variable*/
    ptr = sum;                /*initializing function pointer*/
    ptr(num1, num2);          /*function call*/
    return 0;
}
void sum(int a, int b)
{
    printf("sum : %d\n", a+b); /*prints 16*/
}
```

The declaration of function pointer is based on prototype of the function, whose address has to be kept in pointer. It can be given as :

*return–type (*variable–name)(arg–type1, arg–type2, ...);*

In this *variable–name* is name of function pointer variable that keeps the address of function whose return type is *return–type* and argument types are given by *arg–type1, arg–type2,* etc.

Note that while initializing pointer only name of function is given, because function name represents starting address of that function. There should not be parenthesis after function call.

While calling the function, function pointer can be used instead of function name and the syntax

Scanned by CamScanner

of calling function remains same.

The function address can also be passed to the function as shown in the following program :

Program 11. 2 : Function Pointer as Argument

```
#include <stdio.h>
void sum(int a, int b);
void call(void (*ptr)(int, int), int x, int y);
int main()
{
     int num1=12, num2=4;
     call(sum, num1, num2);    /*pass address of function as parameter*/
     return 0;
}
void sum(int a, int b)
{
     printf("sum : %d\n", a+b);
}
/*call the function with given address*/
void call(void (*ptr)(int, int), int x, int y)
{
     ptr(x, y);      /*function call via pointer. will call sum()*/
}
```

The declaration of function pointer can be simplified by the using typedef keyword as discussed later in this chapter.

enum

Enumerations are mainly used to increase the readability of the program. They are used to give meaningful names to the integer constants. The following program shows use of enumerations.

Problem : Enter the month number from the user in the range 1 to 12 and print the name of month along with number of days in that month. Assume 28 days for February.

Program 11.3 : Enumeration

```
#include <stdio.h>
enum month
{
     JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};
int main()
{
     enum month m = JAN; /*initialization*/
     printf("enter the month (1-12) : ");
     scanf("%d", &m);
     switch(m)
     {
          case JAN : printf("jan-31"); break;
```


Scanned by CamScanner

```
        case FEB : printf("feb-28"); break;
        case MAR : printf("mar-31"); break;
        case APR : printf("apr-30"); break;
        case MAY : printf("may-31"); break;
        case JUN : printf("jun-30"); break;
        case JUL : printf("jul-31"); break;
        case AUG : printf("aug-31"); break;
        case SEP : printf("sep-30"); break;
        case OCT : printf("oct-31"); break;
        case NOV : printf("nov-30"); break;
        case DEC : printf("dec-31"); break;
        default  : printf("invalid month");
    }
    return 0;
}
```

The same program can be done without enum; However, it will not be much readable. In the declaration of *enum month*, JAN represents value 1. So JAN in the source code will be replaced with 1 by the compiler. The value of FEB, MAR, APR ... represents 2, 3, 4 ... respectively. If the value of the first element is not specified, by default zero is taken. The enumeration elements can represent positive or negative integers. It is possible that more than one element in enum represent the same integer value; However, this feature is rarely used. Note that enum variables are internally treated as int and hence can be scanned using scanf(). Size of any enum variable will be same as size of int.

typedef

typedef keyword is used to give a new name to existing data types. It is used to make portable data types and to simplify complicated declarations in C. The typical examples for typedef used in different libraries are listed below :

```
typedef int gint;   /*Linux GTK programming*/
gint* a;  /*a is int pointer*/

typedef unsigned int size_t;  /*standard C libarary*/
size_t size;   /*size is unsigned int*/

typedef unsigned long LPARAM; /*Win32 SDK programming*/
LPARAM lParam; /*lParam is unsigned long*/
```

typedef can be used with the user defined data types. The few examples are given below :

```
typedef struct student STUDENT;
STUDENT s, *p, a[3];     /*s is variable, p is pointer and a is array
of struct student*/
typedef struct book
{
char name[12];
    char author[12];
    double price;
}BOOK;
```


Scanned by CamScanner

```
BOOK b, *p, a[3];   /*b is variable, p is pointer and a is array of
struct book*/

typedef union registers
{
     unsigned short ax;
     unsigned char a[2];
}REGS;
REGS r;   /*r is variable of union registers*/

typedef enum color
{
     RED, GREEN, BLUE
}COLOR;
COLOR c;  /*c is enum color variable*/
```

However, typedef is mainly used to simplify complicated pointer declarations as follows.

```
typedef int arrtype[3];
arrtype *ptr;  /*same as int (*ptr)[3];*/

typedef int (*ptrtype)[3];
ptrtype ptr;   /*same as int (*ptr)[3]; */

typedef void fntype(int, int);
fntype *ptr;   /*same as void (*ptr)(int, int); */

typedef void (*pfntype)(int, int);
pfntype ptr;   /*same as void (*ptr)(int, int); */
pfntype arr[3];     /*same as void (*ptr[3]) (int, int); array of 3
function pointers*/
```

void pointer

The address of the variable is stored in the pointer of the same type. The *void pointer* is a special kind of the pointer whose data type is not specified. It can store address of any type of variable, so it is also called as generic pointer. It is useful while writing generic functions i.e. functions that can work on different types of data e.g. qsort(), fread(), fwrite()

Since data type is unknown, scale factor for void pointer is not defined and hence pointer arithmetic cannot be done on these pointers. Even using *value at* operator on these pointers will cause compile time error. So the pointer must be properly type cast before doing any arithmetic or using value at operator on it. The following program shows a simple demonstration of void pointer.

Program 11.4 : Void pointer

```
#include <stdio.h>
int main()
{
   int a = 3;
   char c = 'A';
```


Scanned by CamScanner

```
    void *p;  /*void pointer declaration*/
    p = &a;   /*pointer initialization with int address. no cast
required*/
    printf("%d ", *(int*)p);   /*need cast to use value at operator*/
    p = &c;     /*assigning char address. no cast required*/
    printf("%c\n", *(char*)p); /*need cast to use value at operator*/
    /*p++;*/    /*compiler error. pointer arithmetic not allowed*/
    return 0;
}
```

Dynamic Memory Allocation

In the chapter of storage classes, different types of storage were discussed. The different storage class offers different life and scope for the variables. However, life of these variables is predefined and cannot be customized programmatically. For example, local variables exist till the function execution is going on or and global variables exist throughout the program.

Also sometimes there is a need to have some array, where array size is unknown at compile time. This size will be taken from the user at runtime.

Standard C library functions like malloc() allows to allocate the memory at runtime for given number of bytes. These functions are declared in stdlib.h. This memory can be released when it is no more required using free() function. The prototype of malloc() is given as

*void*malloc(size_t size);*

Number of bytes to be allocated is passed to malloc() as its argument, *size_t size,* where size_t is typedef to *unsigned int.* Then malloc() allocates the given number of bytes (possibly with the help of operating system) and returns the starting address of that memory block as *void**. The return type of malloc() is *void** as it is generic pointer. In other words, the memory block allocated by the malloc() can be used to store any type of data. The memory allocated by malloc() is always consecutive like arrays. The following program shows typical example of malloc().

Problem : Find the sum of given integers. The number of integers will taken from the user.

Program 11.5 : Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlit.h>
int main()
{
    int i, cnt, *ptr, sum=0;
    printf("enter number of integers : ");
    scanf("%d", &cnt);
    ptr = (int*)malloc(cnt * sizeof(int));   /*allocate memory*/
    printf("enter the values : ";
    for(i=0; i<cnt; i++)
        scanf("%d", &ptr[i]);  /*access array element using subscript []*/
    for(i=0; i<cnt; i++)
        sum = sum + ptr[i];    /*access array element using value at * */
    printf("sum : %d\n", sum);
```


Scanned by CamScanner

```
    free(ptr);                    /*release memory*/
    return 0;
}
```

Note that argument passed to malloc() is *cnt*sizeof(int)*. The *sizeof* operator is used to get size of int, so that source code will be independent of compiler. The address returned by malloc() is assigned to *ptr* after casting it to avoid type mismatch. This is common way of using malloc(). For another data type, the name of data type will be used instead of *int* type given in above example.

Keeping the address of allocated memory block into a pointer is equivalent to pointer to array, as discussed in program 7.2. So such array can be accessed using array or pointer notation on the *ptr*. This illustrated in the above program while scanning and then summing array elements. This base address can be passed to functions as discussed chapter 7.

The most important thing is that allocated memory must be released after its use. To release the memory free() function is called. This function takes the address returned by malloc() as argument. The prototype of the free() is given as :

void free(void ptr);*

Note that the address returned by the malloc() is required to free this memory, so this address should not be lost by any programming mistake. Anyhow if the dynamically memory allocated is not released, then it is said to *memory leakage*. This memory is not used anywhere in the system and hence performance of the system will be affected. Most of the operating systems like UNIX or Windows, free the dynamically allocated memory by the process when process is terminated. However, for the applications that are continuously running e.g. server applications, mobile applications this can be the serious problem.

Also one must remember that once free() is called, the memory whose address was kept in the pointer is released. So now the pointer is pointing to such a memory which is not valid for the application. Such pointer is also referred as *dangling pointer*. Any attempt to access the value at this pointer can cause abnormal program termination. However, this is not the only case of *dangling pointer*. In fact, every uninitialized pointer can be a dangling pointer. It is best practice, initialize any pointer to NULL address, when it is declared or freed.

calloc() is another function in C library used to allocate the memory. Its prototype is given as :

void calloc(size_t count, size_t size);*

The first argument of calloc() i.e. count takes number of elements of the array and the second argument i.e. *size* represents size of one element of that array. The major difference between malloc() and calloc() is that, memory allocated by malloc() has garbage values and memory allocated by calloc() has all bytes initialized to zero. Similar to malloc(), calloc() returns base address of allocated array. The call of malloc() in above example can be replaced by the calloc() as :

ptr = (int)calloc(sizeof(int), cnt);*

realloc() is used to resize dynamically allocated memory block. The prototype of realloc() is given as :

void realloc(void *ptr, size_t newsize);*

It can be used to increase or decrease the size of memory block allocated by malloc() or calloc(). There can be variations in the behavior of realloc() depending on the situations.

- When realloc() is used to shrink dynamically allocated memory block, realloc() release the

Scanned by CamScanner

remaining bytes and returns the same address.

- When realloc() is used to grow the size of dynamically allocated memory block and enough contiguous memory is available to increase the size of block, the size of block is updated and realloc() returns the same address.
- When realloc() is used to grow the size of dynamically allocated memory block and enough contiguous memory is not available to increase the size of block, a new memory block is allocated of the given size, contents of old block are copied to new block, old block of memory is released and finally base address of new block is returned.
- When realloc() is used to grow the size of dynamically allocated memory block and enough contiguous memory is not available to increase the size of block and even memory of new memory block is not available, realloc() returns NULL pointer and old memory block is not released.

Finally few things for malloc(), calloc() and realloc() must be remembered are that.

- These functions return void*, so pointer returned by any of can be used to store any data type in the allocated memory block.
- These functions allocate contiguous memory only and they return NULL is the enough contiguous memory is not available.
- The memory allocated by any of these functions must be released using free() when it is no more required to avoid memory leakage.
- After releasing the memory using free(), that pointer should not be used unless it is reinitialized. Using dangling pointer can terminate the program at runtime.

The text above discusses the dynamic allocation of 1–D array. Few more examples of memory allocation can be given as :

```
/*memory allocation for single structure*/
struct book* ptr; ptr = (struct book*)malloc(sizeof(struct book));
/*memory allocation for array of structure*/
struct book* ptr; ptr = (struct book*)malloc(cnt*sizeof(struct book));
/*memory allocation for string*/
char* ptr; ptr = (char*)malloc(cnt*sizeof(char));
/*memory allocation for array of pointers*/
int** ptr; ptr = (int**)malloc(cnt*sizeof(int*));
/*memory allocation for 2-d array with three columns*/
int(*ptr)[3],  ptr = (int(*)[3]malloc(rows * sizeof(int)));
```

Function with variable argument list

The function like printf() and scanf() takes any number of arguments i.e. in different call of the same function one can pass different number of arguments. Such functions are called as functions with variable argument list. However, such function must have minimum one argument that represents count of the arguments passed to the function. For example, in case of printf() function first argument is format string and count of remaining arguments is equal to the number of format specifiers given in the format string.

The following example has a simple sum() function with variable arguments. The first argument of the function contains the count of remaining arguments. This function uses the macros from the stdarg.h.

Scanned by CamScanner

Program 11.6 : Variable Argument List

```
#include <stdio.h>
#include <stdarg.h>
int sum(int cnt, ...);
int main()
{
	int result;
	result = sum(3, 1, 2, 3);
	printf("sum : %d\n", result); /*prints sum : 6*/
	result = sum(5, 1, 2, 3, 4, 5);
	printf("sum " %d\n", result); /*prints sum : 15*/
	return 0;
}
int sum(int cnt, ...)
{
	int i, total = 0, arg;
	va_list arglist;    /*declare variable that can access argument
list*/
	va_start(arglist, cnt);  /*init arglist to the first argument*/
	for(i=0; i<cnt; i++)
	{
		arg = va_arg(arglist, int);    /*fetch next argument of type
int*/
		total = total + arg;      /*sum up that argument*/
	}
	va_end(arglist);    /*end the arglist*/
	return total;
}
```

Note that the variable argument list is represented by ... in the function prototype, *va_list* is the type declared in header files which is used to access any number of arguments of any type. It is internally typedef to void pointer. *va_start* initialize *va_list* type variable with the first argument, so that it can access consecutive arguments in argument Hst. The macro *va_arg* is used to access the next argument of the given data type. This vaiue can be collected in the variable of that data type. This macro is used number of times to access all the arguments of the function. Finally *va_list* variable is uninitialized by *va_end* macro.

const keyword

If a variable is declared as const, one cannot use any operator on that variable which can modify the value of that variable e.g. =, ++, --, +=, −=, etc. Using any of these variables on this pointer will cause compile time error.

```
const int a = 5;
```

Here a is declared as const, so any of the following statement will cause error message at compile time.

```
a++;
```


Scanned by CamScanner

```
a--;
a=5;
a+=2;
```

So variable can be used as read only variable, like

```
printf("%d", a);
b = a * 4;
```

However, in C const is not mainly used to declare such variable, rather it is used to declare some *const* pointers. These pointers as arguments ensure that their values will not be modified in the functions. Many functions like strlen(), strcpy(), strcmp(), printf(), etc take such pointers as argument. The following table lists different ways of declaring pointers with const keyword.

declaration	meaning	valid statements	invalid statements

const int *p = &b;
OR
int const *p = &b;
	cannot modify any value at the address stored in pointer *p* using *p*.	p = &c;
		p++;
		*p++;
		printf("%d", *p);
			*p = 5;
			++*p;
			(*p)++;

int *const p = &b;	cannot modify the address stored in the pointer.	*p = 5;
		++*p;
		(*p)++;
		printf("%d", *p);
			p = &c;
			P++;
			*p++;

const int * const p = &b;
OR
int const * const p = &b;
	cannot modify the value at the address stored in pointer *p* using *p* and also cannot modify the address stored in the pointer.	printf("%d", *p);	*p = 5;
			++*p;
			(*p)++;
			P = &c;
			P++;
			*P++;

volatile keywords

As discussed in chapter of storage classes, one can create more than one thread for concurrent execution of more than one function simultaneously. It is possible that global variable modified in one function is accessed in another function. Many times, if any variable is used in a loop that repeats number of times compiler optimize it for speed and assign register storage class for it. So the access for that variable become faster, However, any changes done into the variable by any other concurrently executing function may not be reflect immediately into first function. So this optimization should be turned off, which can be done by *volatile* keyword. The word *volatile* means that the value can be modified suddenly. Hence compiler does not assign register storage class for these variables. This variable is always read from its location instead of CPU register whenever required. The volatile variable can be declared as :

volatile int a = 1;

This covers all the syntactical part of C language. However, one should understand that the real use of C is done at system level programming. So even though this can be the end of grammar of C

Scanned by CamScanner

language, the real C begins from here. There are lots of things that can be discovered from C library functions and even more things can be explored into system programming. Most of the programs deal with collection of large or small amount of data. So some amount of data structure knowledge is essential for the programmer, before developing in serious application. The next chapter provides introduction to fundamental data structure concepts and their implantation using C language.

Lab Exercise

Q. 1. Write a menu driven program for four function calculator. Use enumerations to for menu choices. Use array of function pointers instead of switch statement.

Q. 2. Rewrite the program to accept data for 5 persons and display the same. Make use of typedefs. Pass pointer to display function of type pointer to constant structure.

Q. 3. Write a menu driven allocate, free, accept and display matrix of user defined size.

Objective Questions

What will the output of the following programs?

Q. 1.

```
enum color {red, green, yellow=1,dark};
int main()
{
      enum color x = green % dark;
      printf("%d", yellow + dark/x - green *
red);
      return 0;
}
```

1) 3
2) Compiler error
3) 4
4) 2

Q. 2.

```
#include <stdio.h>
#define int char
typedef char ch;
int main()
{
      int x;
      int ch;
      printf("\n%d, %d", sizeof(x), sizeof(ch));
      return 0;
}
```

1) 2, 1
2) Compiler error
3) 1, 1
4) 2, 2

Q. 3.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
      printf("%p", malloc(-1));
      return 0;
}
```

1) (nil)
2) compiler error
3) 0 0 0 0
4) Garbage

Scanned by CamScanner

Q. 4.
```
#include <stdio.h>
int main()
{
    int const *p = 5;
    printf("%d", *++p);
    return 0;
}
```
1) 5
2) Compiler error
3) 6
4) state -1

Q.5.
```
#include <stdio.h>
int main()
{
    int * const volatile p = 5;
    printf("%d", 5/2 + p);
    return 0;
}
```
2 + p
(2 × 4) + 5 = 13
Scale Factor

1) 13
2) Compiler error
3) 9
4) Garbage

Q.6.
```
#include <stdio.h>
typedef struct (int x; NODE *next;)NODE;
int main()
{
    NODE node = {100};
    node.next = &node;
    printf("%d", node.next->x);
    return 0;
}
```
1) 100
2) Compiler error
3) NULL
4) Garbage

Q.7.
```
#include <stdio.h>
int main()
{
    char *p = (char *)malloc(5);
    printf("%d", free(p));
    return 0;
}
```
1) 0
2) Compiler error
3) -1
4) Garbage

Q.8.
```
#include <stdio.h>
int main()
{
    char *const p = (char *)calloc(5, sizeo-
f(char));
    printf("%d", --*p**p);
    return 0;
}
```
1) 1
2) Compiler error
3) -1
4) Garbage

Scanned by CamScanner

ANSWER KEY FOR OBJECTIVE QUESTIONS

	Q.1.	Q.2.	Q.3.	Q.4.	Q.5.	Q.6.	Q.7.	Q.8.	Q.9.	Q.10.	Q.11.
Chapter 1	3	3	3	3	2						
Chapter 2	3	2	1	1	1	2	3	1	3	1	
Chapter 3	1	3	4	2	2	1	4	3			
Chapter 4	4	1	2	3	1	4	2	3	2	4	
Chapter 5	3	1	3	3	1	2	3				
Chapter 6	2	3	4	1	1						
Chapter 7	3	1	2	4	2	2	4	3	2	3	2
Chapter 8	2	1	3	4	2	2	3	4	4	3	
Chapter 9	4	1	3	4	3	4	3	3	4		
Chapter 10	3	3	3	4	2	2	4	2			
Chapter 11	1	3	1	4	1	2	2	1			
Chapter 12	2	4	2								

Scanned by CamScanner