UNIT 1. INTRODUCTION

1.1 ALGORITHMS

Computer solves a problem based on a set of instructions provided to it. A problem should be broken into several smaller steps for it to be transformed into a set of instructions. Such a set of statements/instructions is called as an algorithm. In other words, ***Complete, unambiguous procedure (set of instructions) for solving a problem in a finite number of steps is called as algorithm.***

Every algorithm must be complete and yield at least one output. Every statement in the algorithm must be clear and should not contain any ambiguity. For example,

Add 3 or 5 to x

The above statement is ambiguous as it is not clear whether to add 3 to x or to add 5 to x. Such statements must be avoided in the algorithm. Moreover, every algorithm must give the result in finite number of steps.

Algorithm normally consists of English-like statements. We have a bit-more structured way of writing algorithm, called as ***pseudo code.***

Example 1. Write an algorithm for finding sum of two numbers.

Algorithm:

```
Step 1. Start
Step 2. Read two values, say a and b
Step 3. Add a and b,  and store the result in another variable, say c
Step 4. Display the value of c.
Step 5. Stop
```

Pseudo code:

```
Step 1. Input a, b
Step 2. c ⟸ a + b
Step 3.  Print c
```

Example 2. Write an algorithm for finding biggest of two numbers.

Algorithm:

```
Step 1. Start
Step 2. Read two values, say a and b
Step 3. Compare a and b. Store the larger number in another variable, say big.
Step 4. Display the value of big.
Step 5. Stop
```

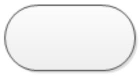
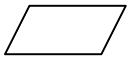
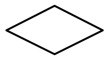
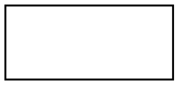
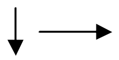
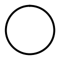
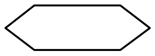
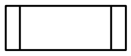
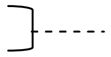
Pseudo code:

```
Step 1. Input a, b
Step 2.  if(a>b)
                big ← a
         else
                big ← b
Step 3.  Print big
```

NOTE: The student can choose either algorithm or pseudo code for solving a problem.

1.2 FLOW CHARTS

***Flow chart is a* pictorial representation of the algorithm.** Flow chart uses some geometrical shapes for representing an algorithm diagrammatically.

Symbol	Meaning	Symbol	Meaning	Symbol	Meaning
	Start/End		Input/Output		Decision
	Processing		Flow Lines		Connector
	Looping Statements		Pre-defined Process		Explanatory Notes

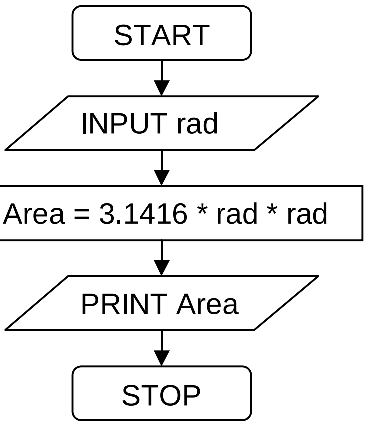
Advantages of using Flowchart:

- Flowchart acts as a blue print during program preparation phase. The program may be compared with flowchart to find if any statement is missing.
- Flowchart may be used to study different parts of a program to identify the problems and find an alternative approach.
- Flowchart helps in understanding the problem easily for any common man.
- In case a program has some logical errors, the flowchart helps in locating the error quickly and leads to easier debugging process.

Example: Write algorithm and flow chart for finding the area of a circle given the value of radius.

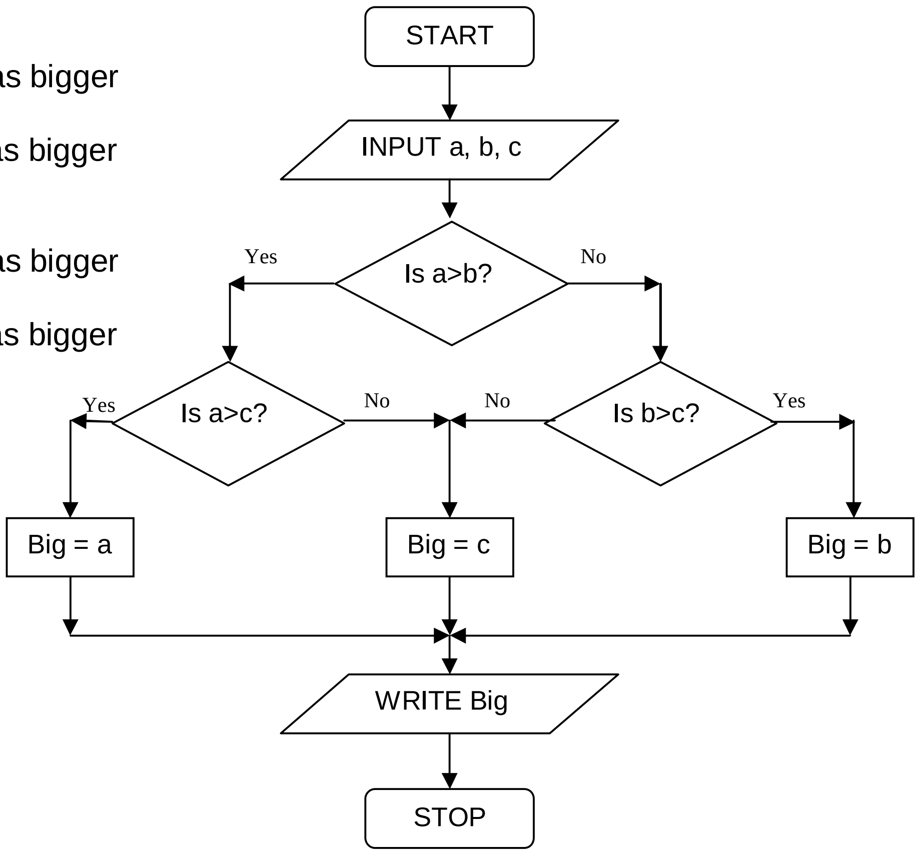
Algorithm/Pseudo Code

```
Step 1: Start
Step 2: Input the value of a radius say, rad.
Step 3: Calculate area = 3.1416 * rad * rad.
Step 4. Print the value of area.
Step 5: Stop.
```



Example: Write an algorithm and flow chart to find the biggest of three numbers.

```
Step 1: Read a, b, c
Step 2: if a>b then
                if a>c then
                        print a as bigger
                else
                        print c as bigger
        else
                if b>c then
                        print b as bigger
                else
                        print c as bigger
Step 3: Stop.
```



1.3 STRUCTURE OF C PROGRAM

The general structure of a C program can be given as -

```
#include<----->
#include<----->         }  Preprocessor Directives
#define -----------     }  Symbolic Constants/Macro Functions

Global Variable Declarations
User-defined Function Declarations/Prototypes

void main()
{
        Local Variable Declarations

        Program Statements
}

User-Defined function Definitions
```

Preprocessor is a software piece which scans the entire program before the source code is handed over to the compiler. Preprocessor directives can be used for including any header files like stdio.h, math.h etc. and also for declaring symbolic constants using #define statement. Preprocessor directives are discussed later in detail.

Global variables are the variables that can be used by all the function in the entire program. Such variable declarations can be provided after a list of preprocessor directives.

C is a modular programming language. That is, the given problem can be divided into several independent sub-problems. And each of these sub-problems can be solved independently. The solutions of all these can then be combined to get the solution for original problem. Such sub-problems can be solved with the help of **user-defined functions** (or modules). The declaration (return-type function-name (argument List)) can be done after/before global variable declarations. (Functions are discussed in later chapters in detail).

The C program execution always starts with a function main(). This function contains the declaration of local variables needed for the process inside the main() function. After local variable declaration, the main() function can contain program statements which may involve i/o operations, any process statements, function-calls etc.

User-defined functions declared earlier have to be defined with a specific set of operational statements later.

Note that, every statement in a C program is terminated by a semicolon and every block of code is enclosed within a pair of flower brackets.

1.4 CHARACTER SET

C programming language uses a finite set of symbols known as character set. A character may be printable or non-printable. The characters are useful in any language to construct identifiers and to form a valid statement. Every character is having an equivalent ASCII (American Standard Code for Information Interchange) code ranging from 0 to 255. Following is a list of printable characters in C.

Table 1.1 Character Set of C

Decimal Digits	**:** 0, 1, 2 … 9
Alphabets	**:** A to Z and a to z
Special Characters	**:** ! “ # $ % & ‘ (
	) * + - / . : ;
	< = > ? @ [ ] \
	^ _ , { } | ~

Note that ASCII codes for 0 to 9 is from 48 - 57,
for a to z is from 97 - 122
for A to Z is from 65 - 90

1.5 TOKENS

One or more characters grouped together to form a basic element in C is called as a token. A token can be a keyword, constant, variable, operator, string, special character etc.

Every word in C is classified as either identifier or keyword. Identifiers are used to name variables, functions, arrays, symbolic constants, macros etc.

1.6 KEYWORDS

Some words are used for particular purpose in C. They have a specific meaning and are reserved to do certain task. Such words are known as keywords or reserved words in C. There are 32 keywords currently defined in standard C as listed below.

Table 1.2 Keywords in C

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

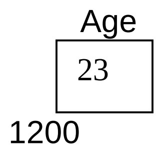
Some of these keywords are used for declaring type of variables, some for controlling the flow of statements etc. The usage of all these keywords will be understood in further discussion of the subject.

1.7 VARIABLES

A variable is a symbolic name to represent quantities in a program. In other words, variable is an identifier to name a specific memory location that can store the data. Computer identifies a memory location with an address assigned to it and each variable name in a program represents a memory location consisting of data. For example, the statement

int Age = 23;

implies that, Age is name of the variable having specific address and storing the value 23. It can represented as -



Here, Age is variable name, 1200 is the address of that variable and 23 is its value.

Note that, value of any variable is stored in binary format. That is, the value is converted into binary number and then it is stored. Also, the address will be actually in hexadecimal format and for the sake of simplicity, we will be considering in decimal format.

Rules for naming variables: To name a variable in C, certain rules have to be followed.

- Variable name should not be a C keyword.
- First character in a variable name must be any alphabet or an underscore (_).The remaining characters can be alphabets, digits or underscore.
- White (empty) space is not allowed.
- No special character other than underscore is allowed.
- The uppercase and lowercase alphabets are treated as different. For example, ***Age, age, aGe*** etc. are all treated as different variables. That is, *C is case-sensitive.*
- In most of the C compilers, the maximum number of characters allowed in a variable name is 8.

Following list gives a list of valid and invalid variable names –

Table 1.3 Example for valid and invalid variable names

Valid Names
Age
Test
Sum2
Stud_marks
_height
x2y2

Invalid Names	**Reason**
2sum	Should not start with digit
#age	Should not start with special character
height 3	Space is not allowed
Stud-marks	Hyphen (-) not allowed
$12Currency	Should not start with special character

NOTE:

- All the variables in a program must be declared with appropriate data type. The process of variable declaration assigns required number of memory locations (in bytes) for each variable based on data type.
- The data stored in a variable can be accessed just by specifying the name of that variable.
- The value of a variable can be changed by assigning new value to it or by reading from keyboard.
- The name of variable should reflect the meaning of value to be stored in it. For example, assume that age of a student is to be stored in a variable. Instead of naming it as *xyz*, it will be more meaningful if it is named as *stud_age* or simply *age*.

1.8 DATA TYPES

Basically, in C, the programmer can have four different types of data. That is, a value stored in a variable can be any one of four types viz. **int, float, char, double**. These are known as basic data types in C.

The integer (**int**) data type is used to store only integer values. The data type **float** is used store real numbers with fractional parts. Character (**char**) variables can store singe characters and **double** variables may store real numbers with higher precisions. These aspects are more elaborated in the following section.

1.9 CONSTANTS

A constant is a quantity that does not change during the program execution. Integer, floating point, character and the string are the four types of constants available in C. There are few modifiers like **short, long, signed, unsigned** etc. to provide various ranges in the values of constants. Each of the constant types are discussed here.

1.9.1 Integer Constants

These are the numeric values with no decimal point. An integer constant can be decimal, octal or hexadecimal. A decimal integer may consist of digits 0-9. An octal number may contain the digits 0-7 and is preceded by zero. Hexadecimal number consist of digits 0-9 and the alphabets *a* to *e* (or *A* to *E*). These are preceded by x or X. Following list gives some integer constants.

Table 1.4 Example for integers

Decimal	Octal	Hexadecimal
0	0	0
02	002	0x2
07	007	0x7
09	011	0x9
10	012	0xa
14	016	0xe

The range of values that can be stored in an integer variable depends on the word length of the computer. If *n* is word length of a computer, then the allowable range of numbers can be given by the formula

$$-2^{n-1} \text{ to } +2^{n-1} -1$$

For example, a 16-bit computer can store the integers in a range of -32768 to +32767. The highest bit (most significant bit) is always reserved for the sign of a number. If the most significant bit (MSB) is 1, then the number is negative, if MSB is 0, then the number is positive.

An integer data type can be either signed or unsigned.

- **Signed Integers:** The integers that may contain a sign-bit are signed-integers.
    - o **int** It is a basic integer data type and requires minimum 16 bits (2 bytes).
    - o **short int** It may be smaller than **int** or equal to **int** depending on machine. In some of the machines, **short int** will be half the size of **int.** If its size is 2 bytes, then the range of these numbers is -32768 to +32767.
    - o **long int** It requires at least 32 bits. So, the allowed range is -2,14,74,83,648 to +2,14,74,83,647 (-2^{31} to $+2^{31}-1$).

- **Unsigned Integers:** In some of the programming situations, the programmer may need only positive constants. Then instead of wasting one bit for sign-bit, the programmer can go for unsigned numbers using the keyword **unsigned**. The unsigned numbers will make use of even the bit reserved for sign. Thus, the allowable range of unsigned integers will be 0 to 65635(0 to $2^{16}-1$). The programmer may use **unsigned short int** (2 bytes), **unsigned long int** (4 bytes) etc.

1.9.2 Floating Point Constants

The numeric values having decimal point and fractional part are known as floating point constants. For example - 8.75, 0.025, 123.89 etc. Since it is difficult to represent very large and very small floating point numbers in a standard decimal form, in C, the exponential form is used as –

(Mantissa) e (Exponent)

Here, Mantissa should have at least one digit along with decimal point and exponent may be either 2 or 3 digit integer. The following table gives some examples:

Table 1.5 Example of exponential notation

Decimal	**Exponential Form**	**C Exponential Form**
7653000	7.653×10^{6}	7.653e06
350000000	3.5×10^{8}	3.5e08
1000000000	1.0×10^{9}	1.0e9
0.000012	1.2×10^{-5}	1.2e-05
-0.0000034	-3.4×10^{-7}	-3.4e-07

The decimal part and fractional part of a floating point numbers are converted into binary format separately and then stored. Thus, the operations on floating point numbers are slower than that on integers.

To store floating point constants, three data types are available in C.

- **float** A float number has 6 digits after decimal point. It requires 4 bytes of memory and the allowed range of numbers is ***-3.4E48*** to ***+3.4E48***.
- **double** This types of constants are used to increase the precision. The number of digits in fractional part is 10 (It is Operating System dependent). It requires 8 bytes of memory and the range allowed is ***-1.7E308*** to ***+1.7E308.***
- **long double** This is used to store largest floating point number. The size is 10 bytes and the range of values is ***-1.7E4932*** to ***+1.7E4932.***

1.9.3 Character Constants
A single character enclosed within single-quotes is called as a character constant. Each character has an equivalent ASCII code. The binary equivalent of this ASCII code is stored with respect to a character constant. Thus, computer stores a character constant as an integer. Examples of character constants are -

‘M’, ‘#’, ‘9’, ‘r’, ‘1’ etc.

The size required for a character constant is 1 byte and the range of values is -128 to +127. Character can be **signed** or **unsigned.** A signed character is same as **char**, but unsigned character has the range from 0 to 255.

1.9.4 String Constants
A sequence of characters enclosed within double-quotes is known as string constant. For example,

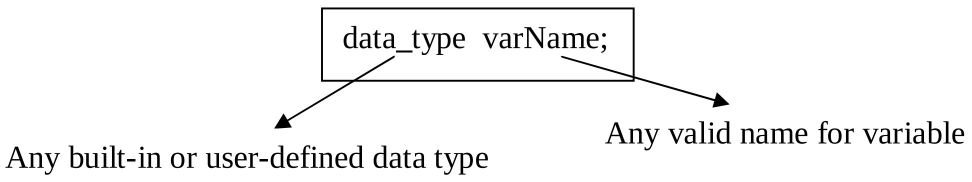
“Hello, how are you?”
“Object Oriented
Programming in
C++”
“x”
“8” etc.

To indicate the end of a string, the compiler will append a null character (‘\0’) at the end of every string. Thus, the total size of any string constant will be total number of characters in a string plus one extra byte for null character.

Note that, “x” is different from ‘x’. The former is a string composed of x and null character requiring two bytes of memory. Whereas, latter is a single character having ASCII value 120 requiring single byte of memory.

String handling is discussed later in detail.

1.10 VARIABLE DECLARATION
In C, all the variables to be used in the program must be declared initially. The syntax of variable declaration is –


data_type varName;
Any built-in or user-defined data type
Any valid name for variable

For example,

```
int age;
float marks;
unsigned char gender;       etc.
```

Variable declaration informs the compiler about the type of the data (to allocate required number of memory in bytes) and about name of a variable.

The programmer can assign any value to a variable at the time of declaration itself. It is known as initialization of a variable. For example,

```
int age=26, salary=10000;
```

This kind of statements will serve both declaration and initialization of a variable.

1.11 SYMBOLIC CONSTANTS

The process of assigning a constant value to a variable may cause a problem during the program. Unknowingly the variable may get altered through assignment statement or any such other operation. To avoid this problem, C provides Symbolic Constants which are declared using preprocessor directive *#define.* The general form of declaring a symbolic constant is -

#define TAG EXP

Here TAG is name of the symbolic constant and EXP is the value assigned to that constant.

For example -

```
#define PI 3.1416
#define MAX 5
```

Following are the rules to be followed while defining symbolic constants:

- There is no space between # and define, but there should be a space between ***define*** and *TAG* and *EXP.*
- The *#define* directive should not end with semicolon.
- Symbolic constant can not be changed later using any statement. That is,

```
#define MAX 5
……
……
MAX=3;        //error
```

- It is better to use capital letters for symbolic constants, to differentiate them from other variables of the program.
- Rules for naming symbolic constants are same as those of naming variables.

1.12 const QUALIFIER

In many programs, there are some situations where we need some value to be constant throughout the program. If we define such a value using any variable name as:

```
int size =10;
```

there is a chance that the value 10 of size may change. Because, after the initialization of size to 10, if the programmer himself unknowingly assigns some other value, say 15 as:

```
size =15;
```

size looses its original value. This may affect the logic of the program.

To prevent such changes in a constant value, we use a qualifier viz. *const*. For example:

```
const int size =10;
```

This keyword *const* assures that the value of the variable will not be changed throughout the program.

1.13 volatile QUALIFIER

This qualifier is used to tell the compiler that a variable's value may be changed, not explicitly specified by the program. For example, a global variable's address may be used to store the real time of the system. In this situation, the contents of the variable are altered without any explicit assignment statements in the program. Many of C/C++ compilers assume that a value of a variable is not changed if it does not appear at the left hand side of an assignment operator. Therefore, there is a chance that the program task is worked out without updating such global variables.

To prevent this from happening, the variable is declared as *volatile*, so that each time the external change occurs in the variable it will reflect in the program. Thus when a variable is preceded with the volatile qualifier, the compiler will not optimize the code using that variable.

It is always a good practice to declare a *volatile* variable with *const* qualifier. Because a volatile variable must not be changed by the program code.

For example:

```
volatile int disp_register;
volatile const int TIME;
```

1.14 COMMENT STATEMENT

Having comments in between a program code to indicate the purpose of a particular statement or function is a good programming style. Comment lines will not occupy space in the memory. The compiler ignores them before object code generation.

There are two types of comment delimiters in C. Whenever the comment text is more than one line, generally we use the pair /* and */. For example,

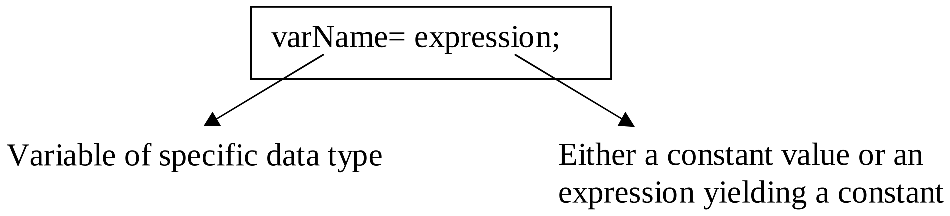
```
/* This is multi-line comment delimiter. A programmer can use
any of the comment delimiters as per programmer's
requirements. */
```

The above passage of text is ignored by the compiler when it is included in the C program, as it is written between /* and */. To comment a single line, one can use the second type of delimiter i.e. //. For example,

```
//This is single line comment.
```

1.15 ASSIGNMENT STATEMENT

It has been discussed that a variable name identifies a memory location and this memory location can store a constant value of a particular data type same as that of variable name. For any given moment of time, a variable can store a single constant value. To store a constant value into a variable, programmer has to use assignment statement. The syntax is –


varName= expression;
Variable of specific data type
Either a constant value or an expression yielding a constant

Here,

varName is known as **lvalue** and
expression is known as **rvalue**

Consider an example -

```
int a, b, c;
float x=2.5, y;
a=10;                     //constant value is assigned
b=20;
c=a+b;                    //evaluated expression is assigned
y=x;                      //value of one variable is copied to other
a=a+1;                    //value of a is increased by 1
```

1.16 INTRODUCTION TO OPERATORS

To solve any problem using computer, one may need to perform some calculations and various types of processes. C provides several types of operators to write a program involving calculations.

The symbols used to perform a specific type of operations are known as operators. The variables and/or constants upon which these operations are carried out are known as operands. An operator requiring two operands is known as **binary operator**, whereas, an operator requiring single operand is **unary operator.**

Different types of operators are discussed here under.

1.17 Arithmetic Operators

The symbols used to perform arithmetic operations are known as arithmetic operators. Following provides a list of arithmetic operators available in C.

Arithmetic operators

Type	Purpose	Operator	Syntax	Meaning
Binary Operators	Addition	+	a + b	Addition of a and b
	Subtraction	-	a – b	Subtract b from a
	Multiplication	*	a * b	Product of a and b
	Division	/	a / b	Divide a by b
	Modulus	%	a % b	Remainder after dividing a by b
Unary Operators	Positive	+	+a	Positive of a
	Negative	-	-a	Negative of a

A valid combination of variables and/or constants with arithmetic operators is known as ***arithmetic expression***. For example,

Assume, int a=10, b=4;

Then, a+b is 14
a-b is 6
a%b is 2
(a+b)/7 is 2 etc.

1.18 Increment and Decrement Operators

In many of the programming situations, the programmer has to increment and decrement value of a variable by 1. Programmer can follow traditional way like,

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

But, C provides more simple way to achieve this task in the form of increment and decrement operators, written as ++ and --respectively. These operators can be used either as a prefix or as a postfix to the variable. The syntax is –

```
var++;     } Post-increment/Post-decrement
var--;

++var;     } Pre-increment/Pre-decrement
--var;
```

The usage of these operators is illustrated below.

Examples:

- int i=5;
  i++; //value of i will be 6.

- int i=5;
  ++i; //value of i will be 6.

- int i=10;
  i--; //value of i will be 9

- int i=10;
  --i; //value of i will be 9

- int i=5,j;
  j=i++; //**i(=5) is assigned to j and then i becomes 6.**

- int i=5,j;
  j=++i; //**i becomes 6 and then i(=6) is assigned to j.**

- int i=10,j;
  j=i--; //**i(=10) is assigned to j and then i becomes 9.**

- int i=5,j;
  j=--i; //**i becomes 9 and then i(=9) is assigned to j.**

Thus, it can be easily observed that, if post/pre increment/decrement operators are used with operand independently, it will not make any difference. But, if the incremented/decremented value is assigned to some other variable, then certainly, there will be change in the value of a variable at left-hand-side of assignment operator.

1.19 Compound Assignment Operators

Sometimes, the programmer has to perform some operation on a variable and the result must be stored in the same variable. For example,

x= x*5;

a= a+4; etc.

In such situations, the assignment operator can be combined with arithmetic operators. Following is a list of compound assignment (short-hand) operators.

Compound Assignment Operators

Operator	Usage	Meaning
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Examples:

- int p=5;
  p+=3; //(p = p+3) p is now 5+3 i.e. 8.

- int q=11;
  q%=2; //(q = q%2) q is now 11%2 i.e. 1

1.20 Relational Operators

Operators used to identify the relationship between two operands are known as relational operators. The expressions involving these operators are relational expressions. Relational expressions always results in either true or false. Thus, they are also known as ***Boolean expressions.*** Following table gives the list of relational operators.

Relational Operators

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Examples:

- int a=5, b=10,c;
  c=(a>b); //a>b is false. So, 0(false) is assigned to c.

- int x=10,y=10,z;
  z=(x==y); //as x and y are equal, 1(true) is assigned to z.

1.21 Logical Operators

Some programming situations require that several relational expressions be evaluated and based on this result, the action should be taken. To combine relational expressions, C provides logical operators. Following is a list of logical operators.

Logical Operators

Type	Operator	Operation
Binary	&&	Logical AND
Binary	||	Logical OR
Unary	!	Logical NOT

The logical expressions results in either true (1) or false (0). Based on the truth-value of the operands, the expression yields the result. The truth-tables for these operators are given below:

Logical AND operation

Operand1	Operand2	Result
Zero	Zero	Zero
Zero	Non-Zero	Zero
Non-Zero	Zero	Zero
Non-Zero	Non-Zero	One

Logical OR operation

Operand1	Operand2	Result
Zero	Zero	Zero
Zero	Non-Zero	One
Non-Zero	Zero	One
Non-Zero	Non-Zero	One

Logical NOT operation

Operand	Result
Zero	One
Non-Zero	Zero

Examples:

-
  ```
  int i=10, j=20,k;
  k=i&&j;            //as i and j are non-zero values, k will be 1
  ```

-
  ```
  int i=10, j=0,k;
  k=i&&j;            //as one of the operands is zero, k will be 0
  ```

-
  ```
  int i=10, j=20, k=15, m;
  m= (i>j) && (j>k);

  /* The expression i>j is false. So, 0 is assigned to m */
  ```

-
  ```
  int i=10, j=20,k;
  k=i||j;       //as i and j are non-zero values, k will be 1
  ```

- int i=10, j=0,k;
  k=i||j; //as one of the operands is non-zero, k will be 1

- int i=10, j=20, k=15, m;
  m= (i>j) || (j>k);

  /* The expression i>j is false. But, j>k is true. As one of the operands for OR operator is true, 1 is assigned to m */

- int m=10,k;
  k=!m;

  /* 10 is true value. So, !m results to be false. So, 0 is assigned to k */

- int x=0,y;
  y=!x; //As x is false, y will be true (1).

1.22 Conditional (Ternary) Operator

When one among two situations must be opted based on some condition, the programmer can go for conditional operator. the syntax is –

```
var = (expr1)? (expr2): (expr3);
```

Here, expr1, expr2 and expr3 are any type of expressions and/or variables. expr1 is a condition resulting true or false. If expr1 results to be true, then expr2 will be evaluated and its value is assigned to var. Otherwise, evaluated result of expr3 is assigned to var. For example,

- int x=10,y=5,z;
  z=(x>y)?x:y;

  As the expression x>y is true here, value of x is assigned to z.

- float a=2.5,b=0.5,c;
  c=(a<b)?(a-b):(a+b);

  The expression a<b is false. So, (a+b) is evaluated and the result (2.5 +0.5 = 3.0) is assigned to c.

1.23 Comma Operator

To make the program compact, two or more distinct expressions can be combined to a single expression using comma operator. The syntax is –

```
(expr1, expr2, …, exprn);
```

Examples:

- int x=5,y=8, a=10,z;
  z=(++x, ++y, a);

  Here, both the expressions ++x and ++y are evaluated. So, values of x and y becomes 6 and 9 respectively. But the usage of comma operator within the pair of parentheses forces the last value to be assigned to a variable. thus, value of a (10) is assigned to z.

- int x=5,y=8, a=10,z;
  z=++x, ++y, a;

  Here also, x and y becomes 6 and 9 respectively. But, the expressions ++y and a are ignored for assignment and value of x (now, it is 6) is assigned to z.

1.24 Bitwise Operators

It has already been discussed that a computer can understand only zeros and ones (binary number format). Every program written by a programmer is translated into the form that a computer can understand and then only it gets executed. In C, programmer has a facility to operate on bits and thus having the features of low-level programming languages. The bit-wise operations provide an efficient way for interacting with the hardware and to perform some arithmetic operations in more elegant manner. Programmer can manipulate bits of variables through several bit-wise operators listed below –

Bit-wise Operators

Type	Operator	Meaning
Binary	&	Bit-wise AND
Binary	|	Bit-wise OR
Binary	^	Bit-wise XOR
Unary	~	One's complement
Binary	>>	Right-shift operator
Binary	<<	Left-shift operator

➢ **Bit-wise AND Operator:**
As the name suggests, initially, operands are converted into binary-format. Then, the AND (&) operation is performed on the corresponding bits of operands. Consider an example –

```
int x=5, y=6,z;
z= x & y;
```

Now, this operation is carried out as –

```
x  ⇨        0000 0000 0000 0101
y  ⇨   &    0000 0000 0000 0110
          ----------------------
z  ⇨        0000 0000 0000 0100
```

Thus, z will be decimal equivalent of 0000 0000 0000 0100, which is 4.

NOTE:

- In the above example, it is assumed that an integer requires two bytes of memory. Hence, 16 bit values are considered.
- Whether a number is even or odd is decided by the value of a lower order bit in the corresponding bit pattern. If the last bit of a number is 1, then the number is odd. Otherwise, it is even. To check the value of lower order bit, one can use a specially called variable viz. **mask** having a value 1. Now, the number to be checked is ANDed (&) with **mask.** If the result is one, then number is odd. Otherwise, it is even. To illustrate this fact, consider an example:

```
int x=5, mask=1,y;
y= x & mask;
```

The operation is performed as –

```
     0000 0000 0000 0011
&    0000 0000 0000 0001
   ----------------------
     0000 0000 0000 0001,   which equivalent to decimal 1.
```

Thus, y will be 1 and so, x is odd.

Consider another example:

```
int x=10, mask=1,y;
y= x & mask;
```

The operation is performed as –

```
     0000 0000 0000 1010
&    0000 0000 0000 0001
   ----------------------
     0000 0000 0000 0000
```

Thus, y will be 0 and so, x is even.

- **Bit-wise OR Operator:**
  Here, the OR (|) operations is performed on individual bits of operands. For example –

```
int x=5, y=6,z;
z= x | y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0000 0000 0101
y ⇨  |    0000 0000 0000 0110
          -------------------
z ⇨       0000 0000 0000 0111
```

Thus, z will be decimal equivalent of 0000 0111, which is 7.

- **Bit-wise XOR Operator:**
  In XOR operation, if both bits are same (either both are 1 or both 0), then the resulting bit will be 0 (false). Otherwise, the resulting bit is 1 (true). For example –

```
int x=5, y=6,z;
z= x ^ y;
```

Now, this operation is carried out as –

```
x ⇨       0000 0000 0000 0101
y ⇨  ^    0000 0000 0000 0110
          -------------------
z ⇨       0000 0000 0000 0011
```

Thus, z will be decimal equivalent of 0000 0011, which is 3.

- **One's Complement:**
  It is a unary operator. This operator changes all 1's of a binary number into 0's and vice-versa. For example –

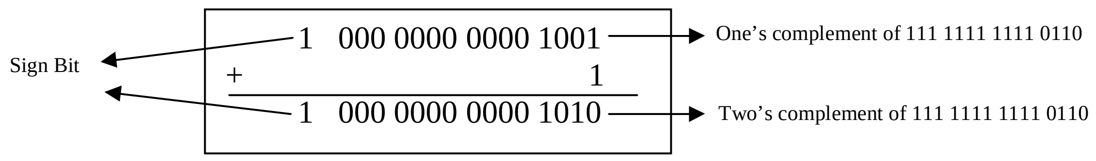
```
int x=9, y;
y= ~x;
```

```
Now,          x =   (0000 0000 0000 1001)
              y = ~(0000 0000 0000 1001)
                = 1111 1111 1111 0110
```

In binary form, we got the value of y as 1111 1111 1111 0110. It means, the sign bit (most significant bit) is 1 now. That, is the number is negative. Hence, while converting this number into decimal, the C compiler treats it as a negative number. And, negative numbers in C are represented as 2's complement. Note that,

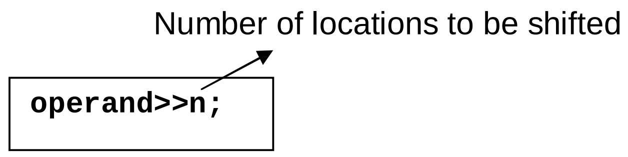
2's complement of a number = one's complement of that number + 1

Thus, in the above example, 2's complement of the number is taken by excluding sign bit. That is, 2's complement of (1111 1111 1111 0110) is calculated as –


Sign Bit
1 000 0000 0000 1001
+ 1
1 000 0000 0000 1010
One's complement of 111 1111 1111 0110
Two's complement of 111 1111 1111 0110

Now, this 2's complement is converted to decimal as -10. Hence, one's complement of +9 would be -10.

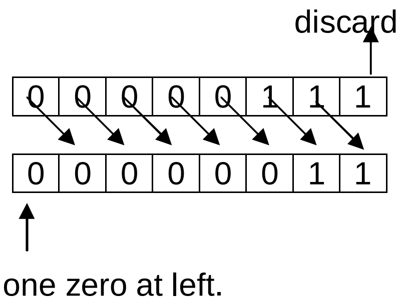
- **Right Shift Operator:**
  This operator is denoted by >> (two greater-than symbols). It will shift each bit of the operand towards right through a number of positions as specified. And the empty bit-positions at left-side must be appended by zeroes. The syntax is:



Examples:

- int x=7,y;
  y= x>>1;

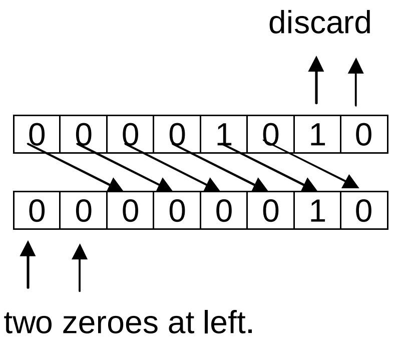
  Here, the bits of x must be shifted one position towards right as shown –



  Thus, y will be 3.

- int x=10,y;
  y= x>>2;

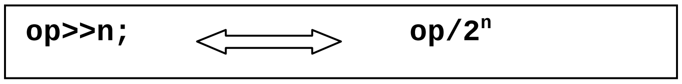
  Here, the bits of x must be shifted two positions towards right as shown –



  Thus, y will be 2 now.

NOTE:

It can be easily observed that right-shift operation on an operand is equivalent to dividing the operand by 2^n. Thus,



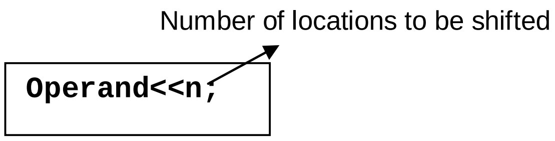
Referring to above examples –

x>>1 is nothing but $x/2^1 = 7/2 = 3$ (integer division)

and x>>2 is(when x=10) $x/2^2 = 10/4 = 2$ (integer division)

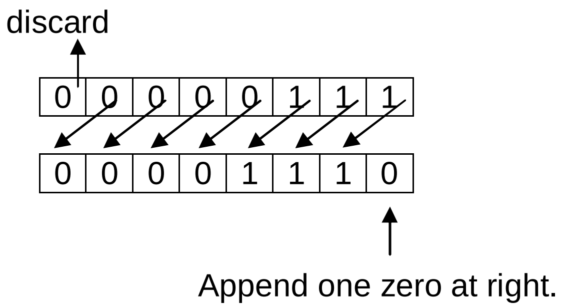
- **Left Shift Operator:**

  This operator is denoted by << (two less-than symbols). It will shift each bit of the operand towards left through a number of positions as specified. And the empty bit-positions at right-side must be appended by zeroes. The syntax is:



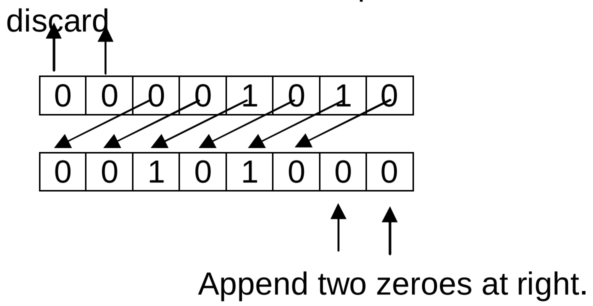
Examples:

- int x=7,y;
  y= x<<1;
  Here, the bits of x must be shifted one position towards left as shown –



  Thus, y will be 14.

- int x=10,y;
  y= x<<2;
  Here, the bits of x must be shifted two positions towards left as shown –



Thus, y will be 40 now.

NOTE:

Here also, one can observe that left-shift operation on an operand is equivalent to multiplying the operand by 2^n. Thus,

Op<<n; $\Longleftrightarrow$ **op*2^n**

Referring to above examples –

x>>1 is nothing but x*2^1 = 7*2 = 14 (integer division)

and x>>2 is(when x=10) x*2^2 = 10*4=40 (integer division)

1.25 Special Operators

There are some operators in C used for specific purpose as listed below:

- Address of Operator (&) : To extract the address of a particular variable.
- Value at the address (*) : To extract the value stored at a particular address (pointer)
- Dot operator (.) : To refer member variable of a structure/union/enumeration
- Indirectional Operaotr (->) : To refer member variable of a structure/union/enumeration through their pointers.

These operators will be discussed later in detail.

1.26 Precedence and Associativity of Operators

Computer evaluates several operations in an expression following a specific order known as precedence/hierarchy of operators. All the operators have two properties known as precedence and associativity. The associativity and precedence of various operators is listed in Table 1.12.

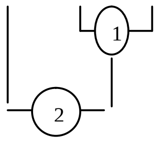
Operators with higher precedence have their operands bound or grouped to them before operators of lower precedence, regardless of the order in which they appear. For example, in the expression,

4+8*2

the multiplication (*) has higher precedence than addition (+). So, 8*2 is evaluated first and then 4 is added to 16.

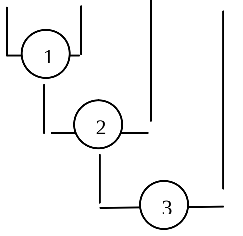
Associativity of an operator determines the direction in which the operands are associated with the operator. The association of operands with operator can be either from left to right or from right to left. The evaluation process of an expression is thus based on associativity of various operators involved in the expression. Consider the following examples:

- 12 + 6 / 3 = 14



- If two operators have equal precedence and occur one after the other, then they are evaluated in a sequence.

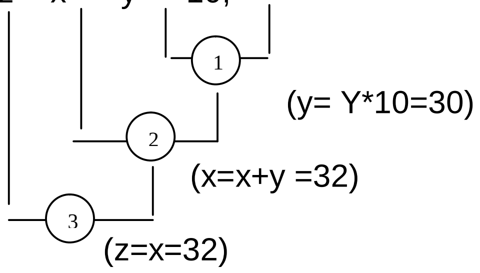
  12 / 4 * 2 + 5 = 11



  Here, / and * have equal precedence and have left to right associativity. Thus, they are evaluated in a sequence from left to right.

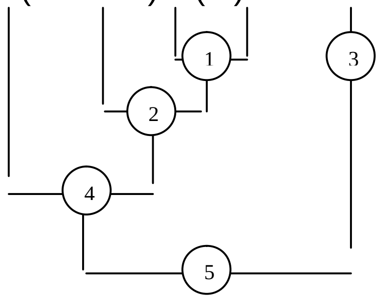
- int x=2,y=3,z;

  z = x += y *= 10;



  Here, the operators =, += and *= are all having same precedence. But, they are associated from right to left. Thus, the value of z will be 32.

- 10 * (3 + 4 / 2) + (-1) = 49



The parentheses have higher priority. So, all the operations within it are evaluated. Then the parenthesis containing unary operator – is evaluated. The next priority is the multiplication and finally addition. Note that all the operators involved here are associated from left to right.

Table 1.12 Precedence of Operators

Precedence	Operator	Operation	Associativity
1	()	Function Call/Parentheses	Left to Right
	[]	Array subscript	
2	!	Logical Negation(NOT)	Right to Left
	~	One's Complement	
	+	Unary Plus	
	-	Unary Minus	
	++	Pre/post increment	
	--	Pre/post decrement	
	&	Address of	
	*	Indirection	
	sizeof	Size of operand(in bytes)	
3	*	Dereference (pointer)	Left to Right
	->	Dereference (pointer to an object of a class or structure)	
4	*	Multiplication	Left to Right
	/	Division	
	%	Modulus (Remainder after division)	
5	+	Addition	Left to Right
	-	Subtraction	
6	<<	Left shift	Left to Right
	>>	Right shift	
7	<	Less than	Left to Right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
8	==	Equality	Left to Right
	!=	Not equal to	
9	&	Bit-wise AND	Left to Right
10	^	Bit-wise XOR	Left to Right
11	|	Bit-wise OR	Left to Right
12	&&	Logical AND	Left to Right
13	||	Logical OR	Left to Right
14	?:	Conditional (ternary) Operator	Left to Right
15	=	Assignment operator	Right to Left
	+=,*=, -=,/= etc.	Compound assignment operators	
16	,	Comma operator	Left to Right

1.27 Type Conversions

Some times it may happen that the type of the expression and the type of the variable on the left hand side of the assignment operator may not be same. In such cases, the value of the expression is promoted or demoted depending on the type of the variable on left hand side of =. Such kind of promotion/demotion of type is called as ***type conversion***. Since the compiler itself does this, it is also known as ***automatic/implicit type conversion.*** For example, consider a code segment:

```
int i;
float b;
i=3.5;
b=30;
```

Here, as *i* can not store fractional value, a floating number 3.5 is converted into integer 3. Similarly, 30 is promoted to 30.000000.

The same rule is applied for arithmetic expression also. For example,

```
float a=5.0, b=2.0;
int x;
x= a/b;
```

Here, x will get the value 2 instead of 2.500000.

Note that, always the promotion happens to the data type with more size.

Some times we need to force the compiler to explicitly convert the value of an expression to a particular data type. Consider a code segment,

```
float z;
int x=6, y=4;
z=x/y;
```

Here, as x and y are integers, the integer division is performed and x/y is evaluated as 1 instead of 1.500000. Now the integer 1 is promoted to 1.000000 and stored in the variable *z,* which is declared as float. Thus, we will not get the expected result. To avoid this problem, the programmer has to convert the type of data during evaluation of expression. This is known as ***explicit type conversion*** or ***type casting.*** The methodology is illustrated in the following code segment –

```
float z;
int x=6, y=4;
z=(float) x/y;
```

Here, the programmer is explicitly converting value of x from 6 to 6.000000. Then 4 is converted into 4.00000 implicitly by the complier. Thus, the value of expression is evaluated to be 1.500000.

1.28 Mathematical Functions

C provides some of the mathematical functions through the header file *math.h* for the programmer's usage. Following is a list showing few of the mathematical functions.

Function	**Meaning**
ceil(x)	x rounded ***up*** to the nearest integer
floor (x)	x rounded ***down*** to the nearest integer
exp(x)	e to the power of x
abs (x)	Absolute value of x
log(x)	Natural logarithm of x
log10(x)	Logarithm to base 10 of x
sqrt(x)	Square root of x
sin(x)	Sine of x
cos(x)	Cosine of x
tan (x)	Tangent of x
sinh (x)	sine hyperbolic of x
cosh (x)	Cosine hyperbolic of x
tanh (x)	Tangent hyperbolic of x

UNIT 2. I/O STATEMENTS AND CONTROL STATEMENTS

2.1 Introduction to I/O Statements

Data input to the computer is processed in accordance with the instructions in a program and the resulting information is presented in the way that is acceptable to the user. For example, to compute c=a+b, the computer should know the values of *a* and *b*. Giving the values to such variables is done in two ways:

- Assigning the values to variables in the program code
- Reading the data from the key board during run time. That is, asking the user to provide the values for the variables during run time.

The second method is efficient because, the program should be flexible to accept any valid data and provide the result accordingly. The process of feeding the data to the computer is known as input and the process of computer displaying the result/data is called as output.

Input and output can be either through console or file in C programming language. Here we will discuss, the built-in functions provided by C language for doing I/O operations through console.

Type	Function	Operation
Formatted I/O Functions	scanf()	Reads one or more values (data types: int, float, char, string) according to user's format specification
	printf()	Outputs one or more values (data types: int, float, char, string) according to user's format specification
Un-formatted I/O Functions	getch()	Reads single character (un-buffered)
	getche()	Reads single character and echos the same (un-buffered)
	putch()	Outputs single character
	getchar()	Reads single character. But enter key has to be pressed after inputting the character
	putchar()	Outputs single character
	gets()	Reads a string until enter key is pressed
	puts()	Outputs a string

The above functions are discussed in detail here under.

2.2 Formatted Console Input Function – scanf()

This function is used to input data through the standard input file viz. keyboard. Basic data types like int, float, char etc. and strings can be read using this function. The general form is –

scanf(control string, list of variables);

Here, the first argument *control string* consists of conversion specifiers. The second argument is list of variables whose values are to be input. The control string and list of variables are separated by a comma.

Example:

```
int a;
float f;
char ch;
scanf("%d%f%c", &a, &f, &ch);
```

The combination of variable name and '&' symbol gives the address of memory location where the corresponding input data is to be stored.

Control string consists of a list of conversion specifiers, wherein each conversion character is preceded by a percent symbol. Data input through the keyboard is actually the text consisting of letters, digits and punctuation symbols and is accepted in the binary form into the computer's memory. That is, if your to input 25 for an integer variable, you will actually key in the text consisting of 2 and 5. The conversion specifiers say %d, %f etc are then used to translate the input information to integer and float data types respectively. The data type order and number of specifiers in the control string should match with those of variable list.

List of Conversion Specifiers

Conversion Specifiers	**Data Types**
%c	Single character
%d or %i	Integer
%ld	Long int
%u	Unsigned int
%f	Floating point
%e	Floating point value in exponential form
%o	Octal value
%lo	Octal long
%x	Hexadecimal lower case a-f
%X	Hexadecimal upper case A-F
%s	String

2.2.1 Rules for scanf()

There are certain rules to be observed carefully while using scanf() function. Few are listed below.

- scanf() function skips white space characters, new line, tabs etc. before an integer or a float value in the input. Example:

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

  For the above statement, the user input can be either

  ```
          25 32                                       (separated by space)

  Or      25              32                          (separated by tab)

  Or      25
          32                                          (separated by new line)
  ```

- Spaces in the control string consisting of conversion specifiers for numerical data type are ignored. That is :

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

  here, space between %d and %f makes no difference.

- Ordinary characters may be placed in the control string region. This rule allows us to introduce comma between two conversion specifiers:

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

  But, now the input also should contain the data items separated by a comma symbol. That is the input should be either:

  ```
          25,  3.2                   (separated by comma and then space)
  Or      25,                 3.2    (separated by comma and then tab)
  Or      25,
          32                         (separated by comma and then new line)
  ```

- But, rules of ignoring white space characters before conversion specifiers for character type of data is different:
  - When a white space character is not included before %c in the control string, the white space character in the input is ***not skipped.***

    ```
    int a;
    char ch;
    scanf("%d%c", &a, &ch);
    ```

    Now, if the input is

    ```
            5x
    ```

    then, *a* will get the value 5 and *ch* will get x.

But, if the input is

```
5   x
```

then, *a* will get the value 5, but *ch* will get the **blank space** as a character.

- o A space before %c in the control string leads to ignoring white space characters before the character data in the input.

```
int a;
char ch;
scanf("%d  %c", &a, &ch);
```

Now, if the input is

```
5x
```

or

```
5   x
```

the variables *a* and *ch* will take the values 5 and x respectively.

- Variable name belonging to string data type (i.e. character array) need not be preceded by an ampersand (&), because name of a string itself is the address of the string.

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

The input can be

```
Ramu        25      75.4
```

2.3 The formatted output – printf()

The data can be output to a standard output device using the function printf(). The general form is –

```
printf(control string, Variable List);
```

Here, control string consists of conversion specifiers as in case of scanf(). Along with conversion specifiers, a string of characters and escape sequence characters can also be used for improved presentation of data. Variable list contains the list of variables whose values are to be output. But, these variable names are not preceded by ampersand (&), since they do not specify the memory addresses. The variable list also may contain arithmetic expressions, names of library functions and user defined functions.

2.4 The Escape Sequence

The escape sequences are used for cursor movements and for printing certain characters such as double quote, back-slash etc. Typical escape sequence used for cursor movement is the tab control character (\t), which shifts the current active position of cursor through 8 character positions horizontally. The escape sequences are very useful for displaying data according to the required format.

Escape Sequence character	**Meaning**
\a	Audio alert
\b	Back space
\f	Form feed
\n	New line
\r	Carriage Return
\t	Horizontal tab
\v	Vertical Tab
\\	Back slash (\)
\’	Single quote (‘)
\”	Double quote (“)

2.5 Space Specifications

For more meaningful input and output, it is essential to specify the number of spaces and starting locations for data items. Format specification is particularly useful for displaying floating point values according to required accuracy. This is done by specifying number of spaces for the fraction part.

The method of space specification for integer, float and string data type items are discussed here.

2.5.1 Integer Data Type

The syntax of space specification for integer data type is

%wd

where, *w* is total number of spaces required for an integer number.
Let us consider an example:

int a=359;

- Now, the statement

  printf(“%3d”, a);

  will generate the output on the monitor as –

3	5	9

  That is, without leaving any space at the top-left corner of the monitor, the number will be displayed.

- The statement

  printf("%2d", a);

  will also display on the monitor as –

3	5	9

  Since width specified is lesser than the actual width needed for the data, compiler will provide enough space.

- The statement

  printf("%5d", a);

  will generate the output on the monitor as –

		3	5	9

  Here, two additional spaces are left blank at the top-left corner of the monitor and then the number is displayed.

2.5.2 Floating point Data type

The syntax of space specification for float data type is

%w.df

where, *w* is total number of spaces required which is inclusive of the space required for decimal point and *d* is number of spaces required for fractional part.

Consider an example:

float a=45.765322;

- The statement

  printf("%8.5f", a);

  will generate the output as –

4	5	.	7	6	5	3	2

  Here, total spaces used including the dot is 8 and the number of digits after decimal point is 5.

- The statement

  printf("%5.2f", a);

  generates

4	5	.	7	7

  Here, note down the value after decimal point. The fractional part 765322 has been rounded off to two places and resulted into 77.

- The statement

  printf("%8.2f",a);

indicates the total spaces to be used is 8, but the fractional part should be of 2 digits. Hence, in the display, first 3 spaces are left blank as shown below –

			4	5	.	7	7

- The statement

  printf(“%3.3f”, a);

  indicates total spaces should be only 3. But, for the given value of variable *a,* this is not possible. Because, the integral part itself has 2 spaces (45) and the dot requires one space. Hence, the compiler will ignore this instruction partially, and generate the output as –

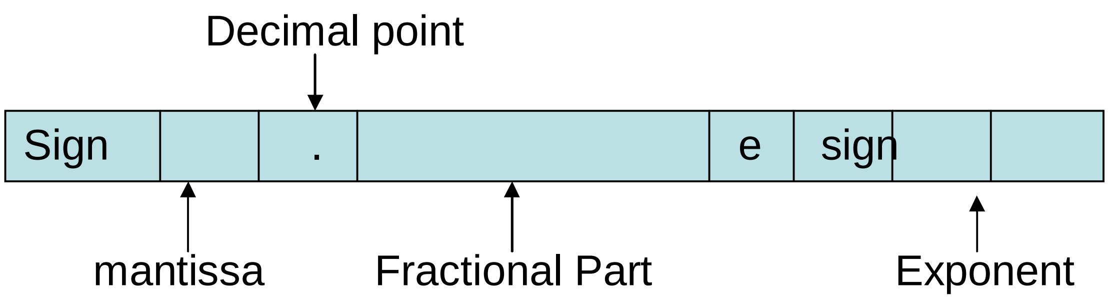
4	5	.	7	6	5

2.5.3 Floating point data with exponential form

The syntax of space specification for exponential form of input/output is

%w.de

where, *w* is total number of spaces required and *d* is number of spaces required for fraction part. The number of spaces provided for the fraction part is according to the required precision. Total number of spaces to be provided for the number will be 7 spaces required for sign of number, mantissa, decimal point, exponent symbol, sign of exponent and exponent itself plus the spaces required for the fraction part.



Let us consider an example:

float a=72457631;
printf(“%12.4e”, a);

The output would be –

		7	.	2	4	5	8	e	+	0	1

2.6 Un-formatted I/O Functions

The unformatted I/O functions are used for characters and strings. These functions require less time compared to their formatted counterparts.

2.6.1 getch(), getche() and putch()

The formatted input function scanf() is a buffered function. That is, after inputting the data, one needs to press the enter key. The input data will first be stored in the temporary memory buffer and after pressing the enter key, the data will be pushed to the actual memory location of the variable. There are two functions viz. getch() and getche() that are un-buffered. That is, these functions do not require the enter key to be pressed after inputting the character.

Example:

```
char opt;
opt=getch();
```

The above set of statements requires a single character to be input from the keyboard. The first character typed will be taken as the input. **But, the character entered by the user is not visible on the monitor.** On the other hand, C provides one more function getche() which ***echoes*** the character typed on to the monitor.

Example:

```
char opt;
opt=getche();
```

Here, the character typed will be displayed on the monitor and without the need of enter key, the typed character will be stored directly on to the physical address of the variable.

The function putch() will display one character.

Example:

```
char opt= 'Y' ;
putch(opt);                  // Y will be displayed on to the monitor.
```

2.6.2 getchar() and putchar()

The getchar() function reads one character from the key board. It is a buffered function. That is, the input character is first stored in the buffer. After pressing the enter key, the input is stored in the actual physical memory. Also, the character typed will be displayed on the monitor. This feature of getchar() facilitates the user to make correction in the input (if necessary) before pressing the enter key.

Example:

```
char opt;
opt=getchar();
```

The putchar() function displays one character on the monitor.
Example:

```
char opt= 'Y' ;
putchar(opt);                // Y will be displayed on to the monitor.
```

The putch() function is available in the header file *conio.h* and this header file is not supported by some of the compilers. Whereas the function putchar() is from *stdio.h* and hence available in all compilers.

2.6.3 gets() and puts()
The gets() function is used to read strings (array of characters). The scanf() function for reading a string (with format specifier %s), requires the string input without any white spaces. That is, for example, we can not give the input as "hello, how are you?" using scanf() function. Whereas, gets() function reads the string till the enter key is pressed.

Example:
```
char str[25];
gets(str);      //reads array of characters till enter key is pressed.
```

The puts() function will display the string on to the monitor.
Example:
```
char str[25]="Hello, How are you?";
puts(str);              // displays Hello, How are you? On to the monitor
```

2.7 Introduction to Control statements
Usually, the statements of a program are executed sequentially. That is, every statement in a program is executed one after the other in the order in which they are written. This kind of linear structure is useful when the values of variables must be generated in a sequential order. But some of the problems will be solved based on decision of a situation. So, in such programs, block of statements must be executed depending on several alternative situations. The selective structures are introduced in C to handle such situations. Thus, *the statements that can alter the sequence of execution of the program are called as control statements.*

Control statements are broadly classified into two categories viz.

i) Selective Control Structure
ii) Repetitive Control Structure (Looping statements)

2.8 Selective Control Structures
Selective control structures are also known as *branching statements.* These statements provide decision making capability to the programmer. C/C++ programming language provides conditional control structures like –

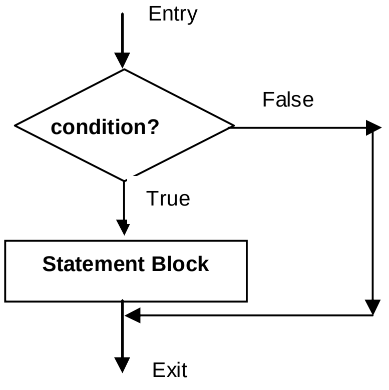
i) ***if*** statement
ii) ***if – else*** statement
iii) ***else – if*** ladder
iv) nested – ***if*** statement
v) ***switch*** statement
vi) ***go to*** statement

2.8.1 The *if* Statement

The *if* statement is one-way conditional branching statement in which a block (or set) of statements can be executed or skipped depending on whether a particular criterion is satisfied or not.

Syntax:

```
if(condition)
{
     Statement block;
}
```



Here,

condition can be a arithmetic/logical/relational expression, a variable, a constant. It can even be a combination of one or more of all these.

Statement block is a set of statements constituting a *true block* of *if* statement. There can be one or more statements. They will be executed when the **condition** is true.

Working Procedure: The execution of **if** statement begins by evaluating the **condition.** It may result in either **true (any non-zero number)** or **false (zero)**. When the value of **condition** is true, the statement – block is executed and then the program control is passed to the statement following the **if**-structure. Otherwise, the statement – block will be skipped without execution and the program control is directly passed to the next statement of **if** structure.

Following is an example showing the usage of **if** statement. Here, a variable is read from the key-board. And then it is checked for negative or positive. If the variable **p** is negative, its absolute value is calculated and printed.

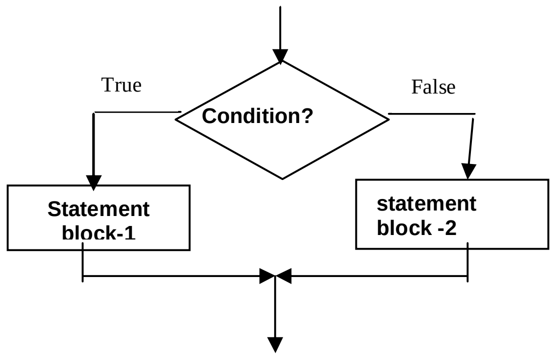
```
int p, q;
scanf(“%d”, &p);

if(p<0)
{
     printf(“p is negative”);
     q=abs(p);
     printf(“\n The absolute value is ”, q);
     exit (0);
}
printf(“p is positive”);
```

2.8.2 The *if-else* Statement

It is a two-way decision making structure, which executes any one of the two statement blocks based on the evaluated value of expression.

Syntax:

```
if(condition)
{
   statement block -1;
}
else
{
   Statement block -2;
}
```



Here,

condition can be a arithmetic/logical/relational expression, a variable, a constant. It can even be a combination of one or more of all these.

True block is a set of statements constituting a *true block* of *if* statement. There can be one or more statements. They will be executed when the **condition** is true.

Else block is also a set of statements and will be executed when the **condition** is false.

Working Procedure: The execution of **if-else** statement begins by evaluating the **condition.** When the value of **condition** is true, the statement block-1 is executed; otherwise, the statement block-2 is executed. After executing any one of these blocks, the program control is passed to the statement following the **if-else** structure.

Example:

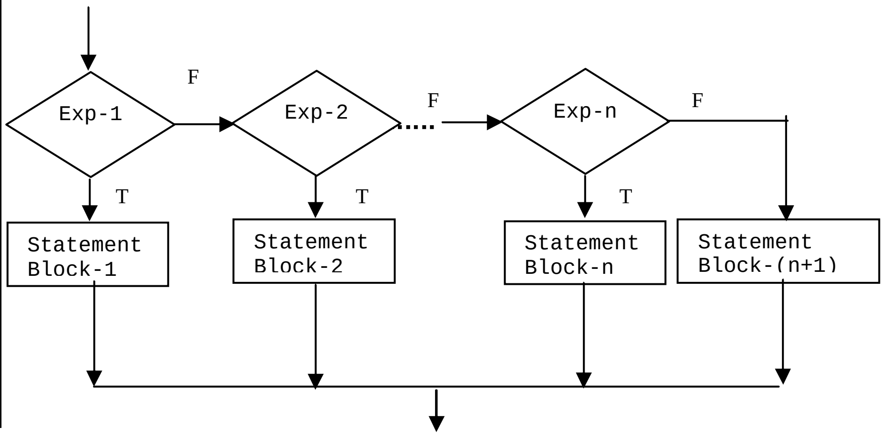
```
int p, q;
scanf(“%d”, &p);
if(p<0)
     printf(“p is negative”);
else
     printf(“p is positive”);
```

2.8.3 The *else-if* ladder

Some times, it is necessary to take one out of many alternatives. In such situations, multiple conditions are to be checked and based on one of the decisions, the block of code must be executed. C/C++ provides multiple choices with else-if known as the *else-if ladder*. This is also known as *multi-way else-if structure.*

Syntax:

```
if (Exp-1)
      Statement Block-1
else if (Exp-2)
      Statement Block-2
      |
      |
      |
      |
else if (Exp-n)
      Statement Block-2
else
      Statement Block-(n+1)
```



Working Procedure: The execution of **else-if** ladder begins by evaluating the **Exp-1.** When the value of **Exp-1** is true, the statement block-1 is executed. If it is false then the **Exp-2** is checked. If this value is true, then statement block-1 is executed. Continuing in this way, the expressions are checked until one of them evaluates true. If none of the expressions is true, then the statement block-(n+1) is executed. After executing any one of these blocks, the program control is passed to the statement following the **else-if** ladder.

Example:

```
int marks;
scanf("%d",&marks);

if (marks >= 80)
      printf("First Class with Distinction");
else if (marks >= 60 && marks < 80)
      printf("First Class");
else if (marks >= 50 && marks < 60)
      printf("Second Class");
else if (marks >= 35 && marks < 50)
      printf("Third Class");
else
      printf("Fail");
```

2.8.4 Nesting of Conditional Statements

Insertion of one or more conditional statements within another conditional statement is known as *Nesting of Conditional Statements*. This method divides the conditions into parts and reduces the complexity of the logic to be implemented. Consider some of the examples:

Example 1:

```
if (marks>=60)
{
      if (marks<70)
            printf("First Class");
```

```
                else
                        printf("Distinction");
        }
```

Here, the program control is sent to inner if statement only when the expression at outer if i.e. marks>=60 is true. When both outer and inner expressions are true, it is printed as first class. But, in case, outer expression is true and inner expression is false, then it is printed as distinction.

Example 2:

```
if (sex == 'M')
{
        if (age >= 21)
                printf("Boy, Eligible for Marriage");
        else
                printf("Boy, Not Eligible for Marriage");
}
else
{
        if (sex == 'F')
        {
                if (age >= 18)
                        printf("Girl, Eligible for Marriage");
                else
                        printf("Girl, Not Eligible for Marriage");
        }
}
```

Note that, the nested conditional structures can be replaced by else-if ladder by using relational and logical operators within expressions.

2.8.5 The *switch-case* Statement

If there are multiple conditions to be checked then the better way is to use else-if ladder. But if there is only one expression which may result into multiple values, then the switch statement is a better alternative.

Syntax:

```
switch (expression)
{
     case value-1: statement-1;
                   break;

     case value-2: statement-2;
                   break;
           |
           |
     case value-n: statement-n;
                   break;

     default:   statement-(n+1);

}
```

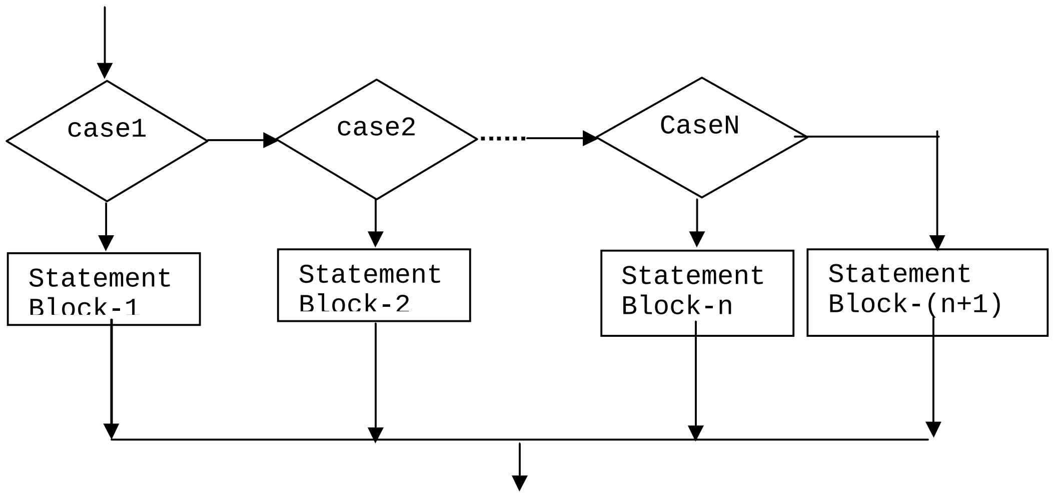
Here

switch, case, default, break are Keywords

expression which is evaluated into either an integer constant or a character constant.

value-1……value-n are the constant values with which the value returned by the expression is compared.

Flowchart:



Working Procedure: Once the control encounters the statement containing the keyword *switch*, expression following the switch is evaluated. The evaluated value is then compared with the value at first case statement. If the value of expression is equal to value-1, then the statements following the first case statement are executed until a break statement is found. Therefore, it is very essential to include a break statement at the end of each statement block. The statement containing break sends the control out of the switch.

Further, if the expression value is not equal to value-1, control directly jumps to the statement containing the next case. Now the value of expression is compared with value-2. If they are equal, the statement-2 gets executed. This procedure continues for all the case values. Thus, the expression value is compared with each case values sequentially until a match is found. Then the statements contained in that block are executed and the control is transferred outside the switch by encountering break statement. If none of the values are matched with expression value, then the statements at default are executed and by encountering closing brace (**}**), the program control comes out of the switch statement.

Note:

1. The case statement must end with a colon (**:**)
2. The default keyword should also end with a colon (**:**), which is an optional block executed only when no case values are equal to the evaluated expression value.
3. The default block should be the last block inside the switch.
4. break is also optional, but it is essential to maintain the distinction between the statement blocks of each cases.
5. There is no need of break in the default block, since default is the last block and control will be passed outside the switch automatically.
6. The values with several cases need not be in a sorted order.
7. More than one statement for a case statement need not be enclosed within braces. As the last statement is break, the program control jumps out the switch statement.
8. Nesting of switch statements is allowed. That is, it is possible to have one switch statement within a case of another switch statement.

Example:

```
int i;
printf("Enter a positive integer (1 to 3) :");
scanf("%d", &i);

switch(i)
{
        case 1:printf("One");
                break;
        case 2: printf("Two");
                break;
        case 3: printf("Three");
                break;
        default: printf("Entered value is not between 1 and 3");
}
```

2.8.6 The *goto* Statement

It is an unconditional statement used to pass the program control from one statement to any other statement in a program. The syntax is –

```
go to label;
:
:
:
label: statement;
```

Here, *label* is used with a statement to which the control is to be transferred. It can be any valid name just like a variable name. But, declaration is not necessary for labels. Consider an example:

```
        int i=0, sum=0;

        start: i++;
               sum+=i;
              if(i<=10) go to start;
```

In the above example, value of i is incremented and is added to sum until i becomes greater than 10.

Note that as the *goto* statement is unconditional, it may enter into infinite loop, if proper care is not taken. Moreover, it leads to unstructured programming. Hence, the programmer is advised to not to use it as many of other control structures are available to satisfy the needs of programming.

2.9 Repetitive Control Structures

Some of the programming statements need to be executed repeatedly. These statements may be repeated either for a pre-defined number of times or till a condition becomes false. For doing such repetitive tasks, C/C++ provides three looping structures viz.

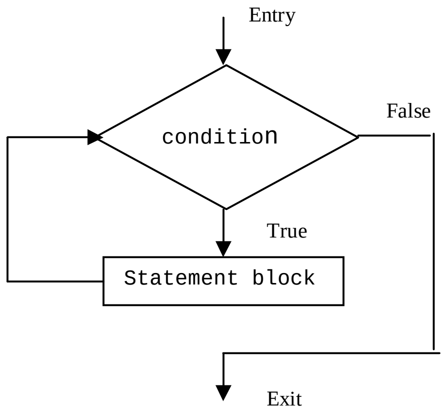
- **while loop**
- **do-while lop**
- **for loop**

2.9.1 The *while* loop

The *while* loop executes a block of statements as long as a condition is true. When the condition becomes false, the while loop terminates. The syntax and flow-charts are as given below:

Syntax:

```
while(condition)
{
     Statement Block
}
```



Working Procedure: When the program control encounters the *while* loop, it checks the condition. The condition can be any arithmetic/logical/relational expression. If the condition is true, the statement block is executed. Again the condition is checked. If it is still true, the statement block is executed for the second time. This procedure of checking the condition and executing the statement block is continued till the condition becomes false. When the condition becomes false, the control is passed to the next statement following the *while* loop. If the condition is false for the very first time, then the statement block never gets executed. Thus, *while* loop is also known as *entry-check* loop.

Example:

```
        int i=1;
        while(i<=10)
        {
              printf(“%d”, i);
              i++;
        }
```

In the above example, after initializing value of i to 1, the condition within while is checked. Since i<=10 is true, the control enters the loop. Now, value of i is printed and incremented to 2. Again the condition i<=10 is checked. Since it is true, value of i (now, 2) is printed and then incremented. This procedure is continued till i become greater than 10 (i.e. 11). When i become 11, the condition becomes false and hence the control will come-out of the loop. By the time, the values from 1 to 10 would have been printed.

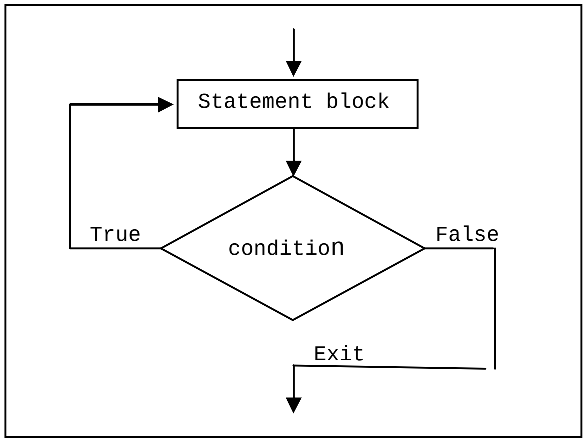
2.9.2 The *do-while* loop

In this loop, the statement block is executed initially and then the condition is checked. If the condition is true, the statement block is executed for the second time. Continuing in this manner, the statement block is executed several times until the condition becomes false. Following are syntax and flow-charts for do-while loop.

Syntax:

```
do
{
      Statement Block
} while(condition);
```

Flow Chart:



Irrespective of condition, the statement block is executed at least once in this loop as condition is checked after executing the statement block. Thus, the do-while loop is also known as *exit-check* loop.

Example:

```
        int i=1;
```

```
do
{
     printf("%d", i);
     i++;
} while(i<=10);
```

In this example, after initializing value of i to 1, it gets printed and then incremented. Now, the condition is checked whether i<=10. Since it is true, value of i gets printed for second time and then incremented. This process is continued till i become 11. By the time, values from 1 to 10 will be printed.

Note: The difference between while loop and do-while loop are listed below:

The Difference between while and do-while loops

while Loop	do-while Loop
1. The condition is checked at the beginning. So, it is entry-check loop.	1. The condition is checked after the statement block. Thus, it is exit-check.
2. As it is entry-check, if the condition is false for the first time, the statement block never gets executed.	2. Since it is, exit-check, even if the condition is false for the very first time, the statement block gets executed at least once.

2.9.3 The *for* loop

The programmer can go for while or do-while loops when the number of times the statement block to be executed is unknown at the beginning. But, if it known well-in-advance, then one can use for loop. The syntax and flow-charts of for loop are:

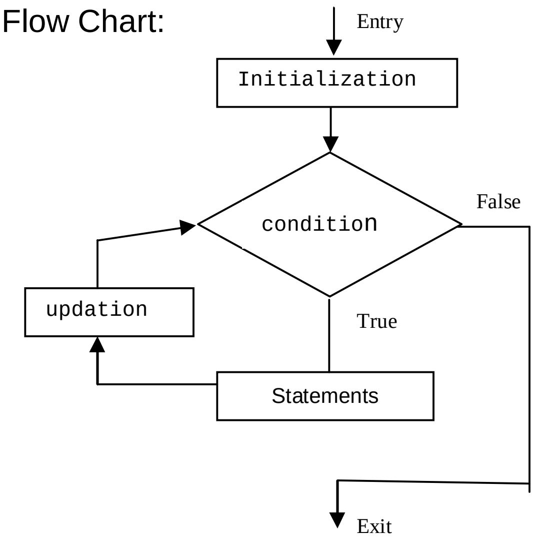
Syntax:

```
for(initialization; test-condition; updation)
{
     Statement block
}
```

Here, *initialization*	is initialization of counter variable
test-condition	is any expression resulting true/false
updation	is an expression that updates the value of counter variable

Note that, these three portions are optional, but the body of the loop should be proper so that, the program should not end-up in infinite number of loops.

Flow Chart:



Working Procedure: When the program control encounters for loop, the initialization part is executed. Then, the test-condition is checked. If it is true, then the statement block is executed. Now, update-assignment is encountered. After updation, the condition is checked once-again. If it is still true, statement block is executed for the second time. Again updation occurs and condition is checked. Continuing in this manner, the statement block is executed until the condition becomes false. Afterwards, the control will come out of the loop.

Thus, it is observed that *initialization* happens only once. The procedure of condition checking-execution of statement block-updation will happen in a sequence till the condition becomes false.

Example:

```
int i;
for(i=1;i<=10;i=i+1)
{
        printf(“%d”,i);
}
```

Within the *for* loop, value of i is assigned to 1. Now the condition i<=10 is checked. Since it is true, the control enters the loop and prints the value of i. now, the update statement i=i+1 is executed. Again the condition is checked. It is still true and hence value of i is printed once again. This procedure continued till i becomes 11 and by the time, values from 1 to 10 would have been printed.

Variations in *for* Loop:
Though the general form of *for* loop is as given at the beginning of this section, there may be several variations as per the requirement of the program. Consider following few variations (with examples):

- The updation in counter variable can be in steps of any number. For example, the following code segment will print every 3rd number starting from 1 to 50.

```
for(i=1; i<50; i=i+3)
        printf("%d", i);
```

  It will print the numbers: 1 4 7 10 etc

- The update assignment can be skipped from the *for* statement and it can be placed at appropriate position in the body of *for* loop.

```
for(i=1; i<10;)
{
        sum=sum+i;
        i++;
}
```

- An empty *for* loop with just two semicolons is executed infinite number of times. Then the program has to be terminated by using ***ctrl – break*** keys.

```
for(; ;) ;
```

- More than one value can be initialized and updated in a *for* statement. And they have to be separated by comma.

```
for(i=1, j=10; i<10; i++, j--)
        printf("%d \t %d \n", i, j);
```

  Here, the value ***i*** is initialized to 1 and incremented in every iteration. Whereas, the value ***j*** is initialized to 10 and decremented in every iteration. Hence, the output would be

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

2.9.4 Nesting of Loops

It is possible to have one loop within another loop. Even it is possible to have several types of loops one within another. This is known as nesting of loops. Consider the following example which prints the ordered pairs from (0,0) to (2,2).

```
int i,j;
for(i=0;i<=2;i++)
{
        j=0;
        while(j<=2)
        {
                printf("(%d, %d)", i, j);
                j++;
        }
}
```

The output of above code segment would be –

```
        (0,0)           (0,1)           (0,2)
        (1,0)           (1,1)           (1,2)
        (2,0)           (2,1)           (2,2)
```

In the above example, the *for* loop is encountered first. After initialization of variable i and checking the condition i<=2, the control enters the loop. Now, j is initialized and the condition of *while* loop j<=2 is checked. Then the ordered pair (0,0) is printed and j is incremented. Again the condition of *while* is checked and this process continued till j becomes 3. Note that value of i is not yet incremented. Once the program control comes out of *while* loop, value of i is incremented and condition of *for* loop is checked. Now, again j starts with 0 and *while* loop terminates when j reaches 3. This process is continued till i become 3. By the time, all the ordered pairs as shown above would have been printed.

Note that, unless the inner loop terminates, the next iteration of outer loop will not start. And for every iteration of outer loop, there will be several iterations of inner loop.

2.9.5 The *break* Statement

The program control will come out of any looping structure either normally or abnormally.

Normal Exit: Transferring the program control out of a loop when its condition is false, after executing the statement block repeatedly is known as normal exit or normal termination of the loop.

Abnormal Exit: In some situations, it is necessary to pass the control out of the loop based on some condition and not completing the required number of iterations. This is known as abnormal termination of a loop.

Abnormal exit from any loop at any point can be achieved by using *break* statement within a conditional statement. Consider the following code segment as an example:

```
int x, sum=0,i;

for(i=1;i<=10;i++)
{
```

```
                printf(”Enter a number:”);
                scanf(“%d”, &x);

                sum+=x;
                printf(“Sum = %d“,sum);
                if(sum>100)
                     break;
           }
           printf(”Over!”);
```

A sample output:

```
Enter a number: 45
Sum = 45
Enter a number: 31
Sum = 76
Enter a number: 34
Sum = 110
Over!
```

In this example, in every iteration, the user is asked to input a number and it is added to sum. Depending on the values entered, the loop will continue for maximum of 10 iterations. But, if the sum becomes more than 100 in any iteration, the loop gets terminated and control will be passed out of the loop. In a sample output shown here, after 3 iterations, the loop has been terminated as the sum crossed 100.

Note that if *break* statement is used in nested loops, only inner loop is terminated.

2.9.6 The *continue* Statement

In some of the programs, few statements within a body of a loop have to be skipped for a particular iteration and the next iteration should be started. When such statements are at the end of the statement block, they can be skipped using *continue* statement followed by a conditional statement. Then the statements after *continue* are bypassed and the control is passed to the beginning of the loop for next iteration. For example:

```
           int i,sum=0;
           for(i=1;i<=10;i++)
           {
                if(i%3 == 0)
                     continue;

                printf(“%d”, i);
           }
```

A sample output would be –

```
     1    2    4    5    7    8    10
```

The above code segment will print all the integers from 1 to 10 which are not divisible by 3. Inside *for* loop, the condition is checked whether i is divisible by 3 or not. If it is divisible, *continue* statement is used to take back the control to for loop and i get incremented. Note that, when i is divisible by 3, the statement

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

is not executed.

2.10 Programming Examples

Various programming examples are given here based on the concepts of topics discussed in this chapter. However, the programs given here are just indicative and students are advised to workout more questions. Also, every problem may have different ways of solving it. And the students are advised to write the programs using their own logic, to get the expected output.

1. Write a C program to find area of a circle given the radius.

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

void main()
{
        float rad, area;
        float PI=3.1416;
        clrscr();

        printf("Enter the value of radius:");
        scanf("%f",&rad);
        area=PI*rad*rad;
        printf("\nThe area of a circle with radius %5.2f is %5.2f",rad,area);
}
```

2. Write a C program to compute simple interest and compound interest given the Principal amount (p), Time in years (t) and Rate of interest(r).

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

void main()
{
        float P,T,R,SI,CI;
        clrscr();

        printf("Enter Principal amount:");
        scanf("%f",&P);
        printf("\nEnter Time duration:");
        scanf("%f",&T);
        printf("\nEnter Rate of interest:");
        scanf("%f",&R);

        SI=(P*T*R)/100;
        printf("\nThe Simple interest is %9.2f",SI);

        CI=P*pow(1+R/100,T)-P;
        printf("\nThe compound interest is %9.2f",CI);
    }
```

3. Write a C program to check whether a given number is even or odd using bitwise operator.

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

void main()
{
        int num,mask,result;
        clrscr();

        mask=1;

        printf("Enter a number :");
        scanf("%d",&num);

        result=num&mask;

        if(result==0)
                printf("\n%d is even",num);
        else
                printf("\n%d is odd",num);
}
```

4. Write a C program to find biggest among three numbers using ternary operator.

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

void main()
{
        int a,b,c,big;
```

```
	clrscr();

	printf("Enter three numbers:\n");
	scanf("%d%d%d",&a,&b,&c);

	big=(a>b)?((b>c)?b:c):((b>c)?b:c);
	printf("\bBiggest is %d",big);
}
```

5. Write a C program to find area of a triangle given three sides.

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

void main()
{
	float a,b,c,big,small1,small2;
	float s,area;
	clrscr();

	printf("Enter three sides:\n");
	scanf("%f%f%f",&a,&b,&c);

	big=a;
	small1=b;
	small2=c;

	if(b>big)
	{
		big=b;
		small1=c;
		small2=a;
	}
	if(c>big)
	{
		big=c;
		small1=a;
		small2=b;
	}
	if(big>=small1+small2)
	{
		printf("\nTriangle cannot be formed");
		exit(0);
	}
```

```
		else
		{
			s=(a+b+c)/2;
			area=sqrt(s*(s-a)*(s-b)*(s-c));
			printf("\nArea of triangle is %5.2f",area);
		}
	}
```

6. Write a C program to find roots of a quadratic equation.

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

void main()
{
	float a, b, c, desc,r1,r2,real,img;
	clrscr();

	printf("Enter the values of a,b,c:\n");
	scanf("%f%f%f",&a,&b,&c);

	if(a==0)
	{
		printf("\nEquation is linear and has only one root. ");
		printf("\nAnd the root is %5.2f", -c/b);
	}
	else
	{
		desc=b*b-4*a*c;
		if(desc==0)
		{
			r1= -b/(2*a);
			r2= -b/(2*a);
			printf("\nThe roots are real and equal");
			printf("\n r1 = %5.2f \t r2 = %5.2f",r1,r2);
		}
		else if(desc>0)
		{
			r1=(-b+sqrt(desc))/(2*a);
			r2=(-b-sqrt(desc))/(2*a);
			printf("\nThe roots are real and distinct");
			printf("\n r1=%5.2f\t r2=%5.2f",r1,r2);
		}
		else
		{
```

```
                    real=-b/(2*a);
                    img=sqrt(-desc)/(2*a);
                    printf("\nThe roots are imaginary");
                    printf("\n r1 = %5.2f + %5.2f i  \n”,real,img);
                    printf(“ r2 = %5.2f - %5.2f i",real,img);
          }
     }
}
```

7. Write a C program using *switch* statement to simulate a simple calculator that performs arithmetic operations.

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

void main()
{
          float a, b, result;
          char op;
          clrscr();

          printf("Enter two numbers:");
          scanf("%f%f",&a,&b);
          fflush(stdin);
          printf("\nEnter the operator(+,-,*,/):");
          scanf("%c",&op);

          switch(op)
          {
                    case '+': result=a+b;
                                break;
                    case '-': result=a-b;
                               break;
                    case '*': result=a*b;
                               break;
                    case '/': if(b==0)
                              {
                                        printf("Division by Zero!");
                                        getch();
                              }
                              else
                                        result=a/b;

                              break;
                    default : printf("Invalid Operator!");
```

```
		}
		printf("The result is %5.2f",result);
	}
```

8. Write a C program to find sum and average of first *n* natural numbers.

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

	void main()
	{
		int n, sum=0,i;
		float avg;
		clrscr();

		printf("Enter the value of n:");
		scanf("%d",&n);

		for(i=1;i<=n;i++)
			sum=sum+i;

		avg=(float)sum/n;
		printf("The sum of first %d natural numbers = %d",n,sum);
		printf("\nAnd the average is %5.2f", avg);
	  }
```

9. Write a C program to find the factorial of a given number.

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

	void main()
	{
		int n,fact=1,i;
		clrscr();

		printf("Enter any positive number:");
		scanf("%d",&n);

		if(n<0)
			printf("Factorial of a negative number can't be generated!");
		else
		{
			for(i=1;i<=n;i++)
				fact=fact*i;
```

```
                printf("%d!=%d",n,fact);
        }
}
```

10. Write a C program to generate Fibonacci sequence up to *n.*

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

void main()
{
        int n,i,fib1,fib2,fib3;
        clrscr();

        fib1=0;
        fib2=1;

        printf("Enter n:");
        scanf("%d",&n);

        if(n<=0)
                printf("\nEnter a positive number");
        else
        {
                printf("\nThe sequence is:\n");

                if(n==1)
                        printf("%d",fib1);
                else if(n==2)
                        printf("%d\t%d",fib1,fib2);
                else
                {
                        printf("%d\t%d",fib1,fib2);
                        for(i=3;i<=n;i++)
                        {
                                fib3=fib1+fib2;
                                printf("\t%d",fib3);
                                fib1=fib2;
                                fib2=fib3;
                        }
                }
        }
}
```

11. Write a C program to find GCD and LCM of two numbers.

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

void main()
{
        int a,b,m,n,gcd,lcm;
        clrscr();

        printf("Enter two numbers:");
        scanf("%d%d",&a,&b);

        if(a==0||b==0)
                printf("\nInvalid input!!");
        else
        {
                m=a;
                n=b;
                while(m!=n)
                {
                        if(n>m)
                                n=n-m;
                        else
                                m=m-n;
                }
                gcd=m;
                lcm=(a*b)/gcd;
                printf("\nGCD of %d and %d is %d",a,b,gcd);
                printf("\nLCM of %d and %d is %d",a,b,lcm);
        }
}
```

12. Write a C program to generate prime number up to the given number *n.*

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

void main()
{
        int i, n,j,flag,limit;
        clrscr();
        printf("Enter n:");
        scanf("%d",&n);

        for(i=1;i<=n;i++)
        {
                flag=1;

                for(j=2;j<i;j++)
                {
                        if(i%j==0)
                        {
                                flag=0;
                                break;
                        }
                }
                if(flag==1)
                        printf("\n%d",i);
        }
}
```

13. Write a C program to reverse a given number and to check whether it is a palindrome.

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

void main()
{
    int num,revnum=0,n,dig;
    clrscr();

    printf("Enter a number:");
    scanf("%d",&num);
    n=num;
    while(n!=0)
    {
            dig=n%10;
            revnum=revnum*10+dig;
            n=n/10;
    }

    printf("\nThe reverse of %d is %d",num,revnum);

    if(num==revnum)
            printf("\nThe number is palindrome");
```

```
        else
                printf("\nThe number is not a palindrome");

    }
```

14. Write a C program to find the sum of all the digits of a given number.

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

void main()
{
        int num,sum=0,n,dig;
        clrscr();

        printf("Enter a number:");
        scanf("%d",&num);
        n=num;
        while(n!=0)
        {
                dig=n%10;
                sum=sum+dig;
                n=n/10;
        }

        printf("\nThe sum of digits of %d is %d",num,sum);
}
```

15. Write a C program find the value of *sin(x)* using the series –

$sin(x)= x – x^3/3! + x^5/5! – x^7/7! + …$

up to n terms. Also print the value of *sin(x)* using standard library function.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define PI 3.1416

void main()
{
        int n,i;
        float x, theta,sum=0,term;
        clrscr();

        printf("Enter x:");
        scanf("%f",&x);
        printf("\nEnter number of terms in series:");
        scanf("%d",&n);

        theta = x * PI/180;
        term=theta;

        for(i=1;i<=n;i++)
        {
                sum=sum+ term;
                term=-term* theta*theta/(2*i*(2*i+1));
        }

        printf("\nsin(%f)=%f",x,sum);
        printf("\nUsing library function, sin(%f)=%f",x,sin(theta));
}
```

16. Write a C program find the value of *cos(x)* using the series –
***cos(x)*= $1 - x^2/2! + x^4/4! - x^6/6! + \ldots$**
up to n terms. Also print the value of *cos(x)* using standard library function.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define PI 3.1416

void main()
{
        int n,i;
        float x, theta,sum=0,term;
        clrscr();

        printf("Enter x:");
        scanf("%f",&x);
        printf("\nEnter number of terms in series:");
        scanf("%d",&n);

        theta = x * PI/180;
        term=1;

        for(i=1;i<=n;i++)
        {
                sum=sum+ term;
                term=-term* theta*theta/(2*i*(2*i-1));
        }
```

```
		printf("\ncos(%f)=%f",x,sum);
		printf("\nUsing library function, cos(%f)=%f",x,cos(theta));
	}
```

UNIT 3. ARRAYS AND STRINGS

3.1 INTRODUCTION
When there is a need for single data, one can declare a variable for that element and solve the problem. Suppose the marks of a student in six subjects needs to be stored. Still one can make use of six different variables. Consider a situation of storing the average marks of all the students in a class, say 60 students. Now, making use of 60 different variables is absurd. The best way to solve such kind of problems is to make use of arrays.

As observed in the above requirements, we may need to have a single variable name that may take multiple values and all are related to each other. These values will be of same data type. Therefore, ***array can be defined as the collection of related data elements stored at the contiguous memory locations.***

Depending on the arrangement of data elements, an array can be classified as one-dimensional array, two-dimensional array etc.

3.2 One Dimensional Arrays
An array in which the elements are arranged in the form of a row is known as one dimensional array or ***vector.*** For example, marks obtained by a student in 6 subjects is one dimensional array, which may be represented by a subscripted variable *marks*. Syntax for declaring one dimensional array is –

```
data_type array_name[Size];
```

Here,

data_type	is any inbuilt/user-defined data type
array_name	is any valid variable name.
Size	is an integer constant representing total number of elements in an array.

Examples:
1. int marks[6];
   marks is an array that can hold *6 elements* of *integer* type.

2. float avg_score[60];
   avg_score is an array that is capable of storing *60 elements* of *float* type.

All the elements of an array will be stored in contiguous memory locations. The total memory allocated for an array is

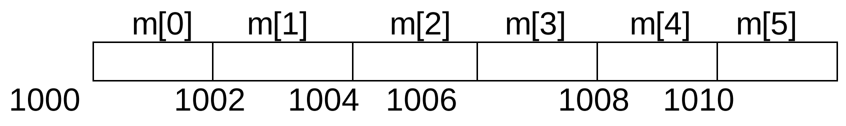
Size_of_array * size_of_data_type

For example, assuming that size of integer variable is 2 bytes and that of float is 4 bytes, the total memory allocated for the array *marks* in the above example is

6*2 =12 bytes
The size of *avg_score* is
60*4 = 240 bytes

Each element can be accessed using an integer variable known as *array index*. The ***value of array index ranges from 0 to size-1***. Consider the memory representation for the array declared as –

int m[6];



Thus, the first element of the array is *m[0],*
second element of the array is *m[1], ……*
last element of the array is *m[5]* (i.e. size-1)

One can observe that all the elements of the array have been allocated memory in contiguous locations. The address of first element of the array is known as ***base address*** of the array. In the above example, base address of the array m[] is 1000.

Note that the address of the element arr[i] in an array arr can be obtaining using the formula –

Address of ith element = base address + i * size_of_data_type

In the above example, the array m[] is of integer type. The size of an integer is 2 bytes and the base address of the array is assumed to be 1000.

Thus, the address of the element, m[3] = 1000 + 3 * 2
= 1006

3.2.1 Initialization of One Dimensional Array

When an array is declared, it will contain garbage values for all its elements. One can initialize the elements of the array at the time of its declaration itself. The syntax for array initialization is –
data_type arr_name[size]={v_1, v_2, …, v_n};

Here, v_1, v_2,…,v_n are constants or expressions yielding constant values of specified data_type.

Consider the following examples:

- int m[6] = { 80, 58, 45, 64, 68, 98};

Now, m[0] is 80,
m[1] is 58 and so on

The memory map would be –

m[0]	m[1]	m[2]	m[3]	m[4]	m[5]
80	58	45	64	68	98
1000	1002	1004	1006	1008	1010

- int m[]={75, 89, 41, 54, 90};

  When the array is initialized at the time of declaration, the size of the array need not be specified. In this example, the size of the array is 5 as the value-list contains 5 values.

- int m[5]={75, 89, 41};
  When sufficient numbers of elements are not provided as per the specified size of the array, then the last elements will take the value as 0. In this example,
  m[0]=75 m[1]=89 m[2]=41 m[3]=0 m[4]=0

3.2.2 Assigning and Accessing 1-D Array Elements
When an array is not initialized at the time of declaration, one can give the values later using the index of an array. For example,

```
int marks[6];          // Declaration of the array

marks [0] = 80;
marks [1] = 58;                and so on.
```

If the values are to be read from the keyboard, then a loop can be used to read array elements as shown –

```
int marks[6], i;       // Declaration of the array

for(i=0;i<6;i++)                  //i ranges from 0 to 5 (size-1)
        scanf("%d",&marks[i]);    //read marks[i] for each i
```

The elements of an array can be accessed in the similar manner using array index as shown –

```
for(i=0;i<6;i++)                  //i ranges from 0 to 5 (size-1)
        printf("%d", marks[i]);   //print marks[i] for each i
```

3.2.3 Programming Examples

In this section let us see some of the examples on one-dimensional arrays.

1. Write a program to read any *n* numbers and to compute average and standard deviation of those numbers.

If a[0],[1],……,a[n-1] are the elements of an array with size n, then the average is –

$$Avg = \frac{1}{n}\sum_{i=0}^{n-1} a[i]$$

The standard deviation is –

$$SD = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(a[i] * a[i]) - Avg * Avg}$$

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

void main()
{
        int a[10],i,n;
        float sum=0,sumsq=0,avg,sd;
        clrscr();

        printf("Enter size of the array:");
        scanf("%d",&n);
        printf("\nEnter array elements:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        for(i=0;i<n;i++)
        {
                sum=sum+a[i];
                sumsq=sumsq+ a[i]*a[i];
        }

        avg=sum/n;
        sd=sqrt(sumsq/n-avg*avg);

        printf("\nAverage=%5.2f",avg);
        printf("\nStandard Deviation= %5.2f",sd);
}
```

2. Write a program to display Fibonacci series up to n.

The fibonacci sequence is defined as –

F(0) = 0
F(1) = 1
F(n) = F(n-1)+F(n-2), for all n>=2

Thus, the sequence is 0, 1, 1, 2, 3, 5, 8,.........

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int fibo[20], i,n;
        clrscr();

        printf("Enter number of elements in a series:");
        scanf("%d", &n);
        fibo[0]=0;
        fibo[1]=1;
        for(i=2;i<n;i++)
                fibo[i]=fibo[i-1]+fibo[i-2];

        printf("\n The Fibonacci series:\n");
        for(i=0;i<n;i++)
                printf("%d \t",fibo[i]);
}
```

A sample output would be –

```
        Enter number of elements in a series: 10
        The Fibonacci series:
    0   1       1       2       3       5       8       13      21      34
```

3. Write a C program to sort a list of elements using bubble sort technique.

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

void main()
{
        int a[10],i,n,j,temp;
        clrscr();

        printf("Enter array size:");
        scanf("%d",&n);
```

```
	printf("\nEnter array elements:\n");
	for(i=0;i<n;i++)
		scanf("%d",&a[i]);

	for(i=1;i<n;i++)
	{
		for(j=0;j<n-i;j++)
		{
			if(a[j]>a[j+1])
			{
				temp=a[j];
				a[j]=a[j+1];
				a[j+1]=temp;
			}
		}
	}

	printf("\nSorted List:\n");
	for(i=0;i<n;i++)
		printf("%d\n",a[i]);
}
```

4. Write a C program to search for a key element in a given list using linear (sequential) search.

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

void main()
{
	int a[10],i,n,flag=0,key;
	clrscr();

	printf("Enter array size:");
	scanf("%d",&n);
	printf("\nEnter array elements:\n");
	for(i=0;i<n;i++)
		scanf("%d",&a[i]);

	printf("\nEnter key element to be searched:");
	scanf("%d",&key);

	for(i=0;i<n;i++)
	{
		if(a[i]==key)
```

```
                {
                        flag=1;
                        break;
                }
        }

        if(flag==1)
                printf("Key is found at the position %d",i+1);
        else
                printf("Key is not found");
}
```

5. **Write a C program to search for a key element in a given list using binary search.**

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

void main()
{
        int a[10],n,i,mid,low,high,key,flag=0;
        clrscr();

        printf("Enter array size:");
        scanf("%d",&n);
        printf("\nEnter array elemnts in ascending order:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        printf("\nEnter key to be searched:");
        scanf("%d",&key);

        low=0;
        high=n-1;

        while(low<=high)
        {
                for(i=0;i<n;i++)
                {
                        mid=(low+high)/2;
                        if(a[mid]==key)
                        {
                                flag=1;
                                break;
                        }
                        else if(a[mid]>key)
                                high=mid-1;
                        else
                                low=mid+1;
                }
                break;
        }

        if(flag==0)
                printf("\nKey not found");
        else
                printf("\nKey found at %d",mid+1);
    }
```

3.3 Multidimensional Arrays

In the previous section, we have discussed one dimensional array. Based on the arrangement of data, arrays can be classified as two dimensional, three dimensional etc. These are known as multidimensional arrays.

Two Dimensional Arrays:

Elements arranged in a row and column order is known as two dimensional array. For example, marks obtained by 3 students in 3 different subjects is a two dimensional array and it can be written in a row-column format as –

	Sub-0	**Sub-1**
Stud-0	75	85
Stud-1	54	62
Stud-2	92	88

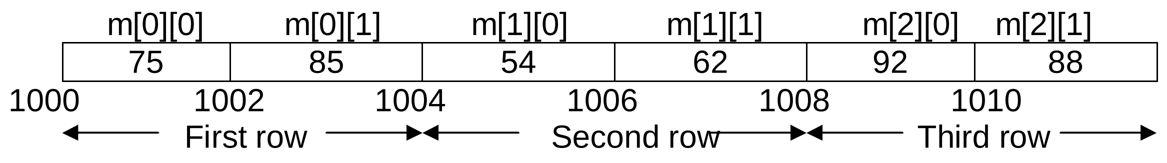
Since two dimensional array is spread into two directions, we need to specify two indices, one to represent row and the other for column. For example-

```
                int m[3][2];
```

Here, the first index 3 indicates the number of rows and second index is to indicate column. The first index (say, i) ranges from 0 to 2 and second index (say, j) takes the values 0 and 1 for each value of first index. Thus, the array elements may be given as –

I	**j**	**Array Elements**
0	0	m[0][0]
	1	m[0][1]
1	0	m[1][0]
	1	m[1][1]
2	0	m[2][0]
	1	m[2][1]

The memory allocation for above 2-D array would look like –



NOTE:

1. The total memory allocated for any 2-D array can be given as –

Size_of_first_index * Size_of_second_index * Size_of_data_type

For example, the total size of the array,

float marks[5][3];

is

5*3*4 =60 bytes (size of float being 4 bytes)

2. Let the size of a 2-D array be m X n. That is, an array is containing m rows and n columns. Then the address of an element at ith row and jth column of 2-D array can be computed using the formula -

base address + (i * n + j) * size_of_data_type

For example, assume that the base address of the following array is 1000.

float marks[5][3];

Then, the address of the element marks[4][2] is

1000 + (4 * 3 + 2)*4 =1056

(Here, the array dimensions, m = 5 and n=3. Size of float being 4 bytes)

Three Dimensional Arrays:

Consider an example of marks obtained by 5 students in 2 tests in each of 3 subjects. This type of data can be given in a tabular format as –

	Subject-0		**Subject-1**		**Subject-2**	
	Test-0	**Test-1**	**Test-0**	**Test-1**	**Test-0**	**Test-1**
Stud-0	75	58	76	90	77	75
Stud-1	54	62	52	64	55	61
Stud-2	92	88	90	88	85	82
Stud-3	75	85	77	75	73	77
Stud-4	45	48	55	52	41	44

A 3-D array for the above example can be declared as –

int m[5][2][3];

The first index represents number of students
second index represents number of tests
third index represents number of subjects

The marks obtained by 3rd student in 2nd test in 2nd subject can be accessed as –
m[2][1][1]

Discussion of 3-D and higher dimensional arrays is out of the scope of this book. In the following sections, 2-D array is discussed more in detail.

3.3.1 Initialization of 2-D Array

At the time of declaration, a 2-D array can be initialized just like 1-D array. Consider the following examples that illustrate various ways and their meaning of initializing 2-D array.

- int m[3][2] = {{56,78},{67,61},{90,85}};
  Now, m[0][0]=56, m[0][1]=78, m[1][0]=67 and so on

- int m[3][2] = {56,78,67,61,90,85};
  Now also, m[0][0]=56, m[0][1]=78, m[1][0]=67 and so on
  That is, when sufficient numbers of elements are given at the time of initialization, the inner braces are not necessary.

- If sufficient numbers of elements are not given and inner braces are removed, then the elements at the end will take the value as zero. That is,

  ```
  int m[3][2] = {56,78,67,61};
  ```

  Now, m[0][0], m[0][1], m[1][0], m[1][1] will take the values 56,78, 67 and 61 respectively. But, m[2][0] and m[2][1] will be zero.

- If sufficient numbers of elements are not given within inner braces, then the corresponding elements will be zero. For example –

  ```
  int m[3][4] = {{56,78},{67,61,85},{75}};
  ```

  is equivalent to –

  ```
  int m[3][4] = {{56, 78, 0, 0},
                 {67, 61, 85, 0},
                 {75, 0, 0, 0}
                };
  ```

- If sufficient numbers of elements are provided along with inner braces, then the first index of the array can be omitted by only specifying the second index. for example –

  ```
  int m[][2] = {{56,78},{67,61},{90,85}};
  ```

  Now, it is understood by the compiler that the number of rows is 3.

3.3.2 Assigning and Accessing 2-D Array Elements
Assigning values to the elements of 2-D are just similar to that of 1-D array. Reading the values from the keyboard for 2-D array and then printing those values is illustrated here under:

```
    int m[5][4],i,j;              //declaring 2-D array

    for(i=0;i<5;i++)              //first index ranges from 0 to 5-1=4
        for(j=0;j<4;j++)                  //second index ranges from 0 to 4-1=3
                scanf("%d", &m[i][j]);        //read elements

    for(i=0;i<5;i++)
        for(j=0;j<4;j++)
                printf("%d", m[i][j]);   //print elements
```

3.3.3 Programming Examples
In this section, we will discuss some useful programs that requires 2-D array.

1. Write a program to find trace and norm of a given matrix.
Trace of a matrix is calculated only for a square matrix. It is a sum of all the elements on a principal diagonal of a square matrix. Norm of a matrix is the square root of the sum of squares of elements of a matrix. Consider the following matrix of order 3x3 –

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Now, trace of this matrix is = $a_{00} + a_{11} + a_{22}$

Norm of a matrix with the order m x n is = $\sqrt{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left(a_{ij} * a_{ij}\right)}$

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

void main()
{
        int a[5][5],m,n,i,j,trace=0;
        float norm, sumsq=0;
        clrscr();

        printf("Enter Number of rows and columns:");
        scanf(“%d %d”, &m, &n);

        if(m!=n)
        {
                printf("\nNot a sqare matrix!!!");
                printf("\nCant find trace. But you can try Norm.");
        }

        printf("\nEnter elements:\n");
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        scanf(“%d”, &a[i][j]);

        if(m==n)
        {
                for(i=0;i<m;i++)
                {
                        for(j=0;j<n;j++)
                        {
                                if(i==j)
                                        trace+=a[i][j];
                        }
                }
                printf("\nTrace = ", trace);
        }

        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        sumsq=sumsq + a[i][j]*a[i][j];

        norm=sqrt(sumsq);
        printf("\nNorm = ", norm);
}
```

2. Write a program to add two matrices.

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

void main()
{
        int a[5][5],b[5][5],sum[5][5];
        int m,n,p,q,i,j;
        clrscr();

        printf("Enter order of first matrix:\n");
        scanf("%d%d", &m, &n);
        printf("Enter order of second matrix:\n");
        scanf("%d%d", &p, &q);

        if(m!=p || n!=q)
        {
                printf("Orders must be same. Addition not possible");
                getch();
                exit(0);
        }
        else
        {
                printf("\nEnter 1st matrix:\n");
                for(i=0;i<m;i++)
                        for(j=0;j<n;j++)
                                scanf("%d", &a[i][j]);

                printf("\nEnter 2nd matrix:\n");
                for(i=0;i<p;i++)
                        for(j=0;j<q;j++)
                                scanf("%d", &b[i][j]);

                for(i=0;i<m;i++)
                        for(j=0;j<n;j++)
                                sum[i][j]=a[i][j]+b[i][j];

                printf("\nThe addition of two matrices:\n");

                for(i=0;i<m;i++)
                {
                        for(j=0;j<n;j++)
                                printf("%d\t",sum[i][j]);
                        printf("\n");
                }
        }
}
```

3. Write a program to find transpose of a given matrix.
Transpose of a matrix is changing of its rows into columns and columns into rows. For example,

$$A = \begin{pmatrix} 4 & 5 \\ -2 & 0 \\ 7 & 3 \end{pmatrix} \qquad \text{Then, } A' = \begin{pmatrix} 4 & -2 & 7 \\ 5 & 0 & 3 \end{pmatrix}$$

Thus, if a matrix is of order m x n, then its transpose will be of order n x m.

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

void main()
{
	int a[5][5],trans[5][5],m,n,i,j;
	clrscr();

	printf("Enter the order of matrix:\n");
	scanf("%d%d", &m, &n);
	printf("\nEnter elements:\n");
	for(i=0;i<m;i++)
		for(j=0;j<n;j++)
			scanf("%d", &a[i][j]);

	for(i=0;i<n;i++)
		for(j=0;j<m;j++)
			trans[i][j]=a[j][i];

	printf("\nGiven matrix:\n");
	for(i=0;i<m;i++)
	{
		for(j=0;j<n;j++)
			printf("%d\t", a[i][j]);
		printf("\n");
	}

	printf("\nTranspose is:\n");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
			printf("%d\t", trans[i][j]);
		printf("\n");
	}
}
```

3.4 INTRODUCTION TO STRINGS

C does not provide a separate data type *string* unlike many other languages. But, an array of characters (char) can be treated as a string. The concepts of handling arrays such as array declaration, initialization and manipulation may be adopted to deal with character arrays/strings.

C provides a rich set of library functions for processing strings. A string contains one or more than one characters enclosed in double quotes. The C compiler automatically provides a *null character '\0'* at the end of a string, when it is stored as an array of characters. Though null character appears as a combination of two characters, it is actually a single character whose ASCII value is 0. This indicates the end of a string and hence very useful in tracing the characters of a string. Each character in an array occupies *one byte* of memory. So, a string with *n* characters has to be provided with *n+1* bytes of memory so as to accommodate the null character at the end.

Example:

A string "Hello" is stored in the memory as –

H	e	l	l	o	\0

3.4.1 Rules for constructing Strings

Following are certain rules to be followed while creating string (character array) variables.

- A string constant is one or more characters enclosed in double quotes. The double quotation marks serving as *delimiters* are not the part of strings.
  Examples:
  "Problem Solving Using C"
  "35"
  "Hello" etc.

- Non-graphic characters such as \n, \t etc can be part of a character array. For example:
  "\n \n Student List \n \n"
  "Name \t USN"

- A string may be continued in the next row by placing a back slash at the end of the portion of the string in the present row. For example:
  "Bangalore is \
  the capital of Karnataka"
  is same as
  "Bangalore is the capital of Karnataka"

- If the characters such as double quotes, back slash, single quote etc. are to be part of a string, they should be preceded by a back slash. For example:
  " \"What a beautiful flower!!\" "
  will yield the output as
  "What a beautiful flower!!

“I Said \’Hello\’ “
will give the output as:
I said ‘Hello’

“Division \\ Multiplication”
will be resulting as
Division \ Multiplication

3.4.2 Declaration and Initialization of Strings

A string can be declared as –

char var_name[size];

An un-initialized string contains garbage values. A string can be initialized in different ways as shown –

- char s[10]=”Hello”;
  Now, the memory allocation for s may look like –

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	e	l	l	o	‘\0’				
100	101	102	103	104	105	106	107	108	109

  Though the value of the string (“Hello”) is only 5 characters long, the string s requires 6 bytes of memory. The last byte is used for null character, which indicates end of the string. If the allocated memory for the string is more than required, then the remaining bytes will be containing garbage values. In this example, the locations s[6],s[7],s[8] and s[9] are containing garbage values.

- String can be initialized through single characters using flower brackets as we do for numeric array. In such a situation, the null character must be specified explicitly. For example –

  char s[10]={‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

  The memory allocation is same as shown in the above example.

- If string variable is initialized with sufficient number of characters, then the size of the string is optional. For example –
  char s[]=”Hello”; **OR**
  char s[]={‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

  In both the situations, the string size will be 6 bytes.

3.4.3 String Handling Functions

C provides several library functions for performing various operations on strings. These functions are included in the header file string.h. The following table lists out few of such functions.

String Handling functions

Function	Operation
strlen(str)	Returns number of characters in a string str (excluding null character)
strcpy(s1,s2)	Copies the content of s2 into s1
strcat(s1,s2)	Concatenates two strings s1 and s2. That is, contents of s2 are appended at the end of s1.
strcmp(s1,s2)	Compares two strings s1 and s2 character by character, starting from first position. The comparison is carried out till a mismatch is found or one of the strings gets exhausted, whichever is earlier. This function returns the difference between the ASCII values of first non-matching characters. If the returned value is 0, then the strings are equal. If the value is positive, then s1 is greater than s2. Otherwise, s2 is greater than s1.
strcmpi(s1,s2)	Compares two strings ignoring the case of the characters. That is, this function is case-insensitive.
strncpy(s1,s2,n)	Copies first n characters of s2 into s1, where n is an integer.
strncmp(s1,s2,n)	Compares at the most n characters in s1 and s2.

Consider the following examples –

-
  ```
  int len;
  char str[20]=”Hello”;
  len=strlen(str);          //len gets value as 5
  ```

-
  ```
  char s1[20],s2[]=”World”;
  strcpy(s1,s2);            //”World” is copied into s1
  ```

-
  ```
  char s1[20]=”Hello”, s2[]=”World;
  strcat(s1,s2);            //s1 becomes “HelloWorld”
  ```

-
  ```
  char s1[]=”Ramu”, s2[]=”Raju”;
  int n=strcmp(s1,s2);
  ```

  Now, the first non-matching characters are m and j. Difference between ASCII values of m and j is 109-106=3. Since a positive number is returned, s1 is greater than s2.

  ```
  strcmp(“cup”,”cap”);      //returns 20
  strcmp(“cap”,”cup”);      //returns -20
  strcmp(“cap”,”cap”);      //returns 0
  ```

- strcmpi(“Hello”,”hello”); //returns 0

- char s1[20], s2=”Programming”;
  strncpy(s1,s2,4);   //s1 gets value “Prog”

To find the types of characters in a given string, there are several functions provided by C through a header file ctype.h. Following table lists out few of such functions.

Functions available in ctype.h

Function	**Operation**
isalpha(ch)	Finds whether the character ch is an alphabetic character
isalnum(ch)	Checks whether alphanumeric (either alphabet or numeric) character or not
isdigit(ch)	Checks whether ch is a digit or not
islower(ch)	Checks whether ch is a lower case alphabet
isupper(ch)	Checks whether ch is an upper case alphabet
isspace(ch)	Finds whether ch is a white space character
isprint(ch)	Checks whether printable character or not
isxdigit(ch)	Finds whether hexadecimal digit
tolower(ch)	Converts ch into lower case
toupper(ch)	Converts ch into upper case
toascii(ch)	Converts ch into equivalent ASCII character

3.4.4 Programming Examples

The important programs involve solving the following issues without using inbuilt string function –

- Finding length of the string
- Concatenate two strings
- Copy one string to other
- Reversing a string and checking whether it is palindrome or not
- Comparing two strings.

Also, some problems on single characters involve:

- Counting number of vowels and consonants in a sentence
- Changing case of a letter in a sentence (upper case to lower and vice-versa)

1. Write a C program to check whether a string is palindrome or not.

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

void main()
{
	char str[15],revstr[15];
	int i,len=0,flag=0;
	clrscr();

	printf("Enter a string:\n");
	scanf("%s",str);

	for(i=0;str[i]!='\0';i++)
		len++;
	i=len;
	while(i>=0)
	{
	  revstr[len-i]=str[i-1];
	  i--;
	}
	revstr[len+1]='\0';
	printf("\nThe Reverse of %s is %s",str,revstr);

	for(i=0;i<len;i++)
	{
		if(str[i]!=revstr[i])
		{
			flag=1;
			break;
		}
	}
	if(flag==1)
		printf("\nIt is not a Palindrome");
	else
		printf("\nIt is a Palindrome");
}
```

2. Write a C program to concatenate two strings without using string functions.

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

void main()
{
	char str1[10],str2[10],str3[20];
	int i,j;
	clrscr();

	printf("Enter the first string:\n");
	scanf("%s",str1);

	printf("Enter the second string:\n");
```

```
        scanf("%s",str2);

        for(i=0;str1[i]!='\0';i++)
                str3[i]=str1[i];

        for(j=0;str2[j]!='\0';j++)
                str3[i+j]=str2[j];

        str3[i+j]='\0';

        printf("The concatinated string is: %s",str3);
    }
```

3. Write a C program to read a sentence and to count the number of vowels and consonants in that.

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

void main()
{
        char ch;
        int vowel=0,cons=0;
        clrscr();

        printf("Enter a sentence:\n");
        while((ch=getchar())!='\n')
        {
                switch(tolower(ch))
                {
                        case 'a':
                        case 'e':
                        case 'i':
                        case 'o':
                        case 'u': vowel++;
                                        break;
                        default : if((ch>='a' && ch<='z')||(ch>='A' && ch<='Z'))
                                                cons++;
                                        break;
                }
        }

        printf("\nNumber of vowels=%d",vowel);
        printf("\nNumber of consonents=%d",cons);
    }
```

4. Write a C program to read a sentence and to replace the uppercase character by lowercase and vice-versa.

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

void main()
{
        char ch;
        clrscr();

        printf("Enter a sentence:\n");

        while((ch=getchar())!='\n')
        {
                if(islower(ch))
                        ch=toupper(ch);
                else if(isupper(ch))
                        ch=tolower(ch);
                putchar(ch);
        }
    }
```

5. Write a C program to compare two strings without using string functions.

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

void main()
{
        char str1[10],str2[10];
        int i,len=0,flag=1;
        clrscr();

        printf("Enter first string:");
        scanf("%s",str1);
        printf("\nEnter second string:");
        scanf("%s",str2);

        i=0;
```

```
	while(str1[i]!='\0')
	{
		len++;
		i++;
	}

	for(i=0;i<len;i++)
	{
		if(str1[i]!=str2[i])
		{
			flag=0;
			break;
		}
	}

	if(flag==0)
		printf("\nString are not equal");
	else
		printf("\nStrings are equal");
}
```

UNITS 4 AND 5. FUNCTIONS AND POINTERS

4.1 INTRODUCTION

Transforming a larger application into a single problem leads t o several complications such as inefficient memory management, difficulty in the development of source code, debugging etc. Modularization of a program will avoid these complications. Here, the original problem is divided into several sub-problems and solved. Then the solutions of all of these sub-problems are combined to get the solution for original problem.

A sub-program is a set of instructions that can be executed under the direction of another program to get a solution of a sub-problem. Usually, subprograms are written for smaller and independent tasks that appear frequently in larger problems. In C/C++, the programmer has a concept of ***functions*** for modularization. In fact, main() is also a function from where the execution starts in every program and it is a compulsory function in any C/C++ program. A subprogram contains type declarations, statements etc. and it can not be executed by itself. It can be executed only through another subprogram. A subprogram or main() that calls another subprogram is known as ***calling function*** and the activated subprogram is known as ***called function***.

A function may be in-built or user-defined. An in-built or library function is a subprogram stored in the library of compiler and is readily available to all the programs. For example, *pow(), sqrt(), clrscr()* etc. User-defined functions are the functions developed by the programmer to solve a particular task. A calling function may supply some values known as ***arguments/actual parameters*** to a called function and the called function receives these values through a set of variables known as ***formal parameters***.

4.1.1 Advantages of Functions

The usage of functions in a program has following advantages:

- **Reduced Code Size:** Some times, the programmer has to perform a particular task repeatedly in a program. For example, one has to find factorial of a number, square root of a number, etc. at several stages in a program. Then, instead of writing statements repeatedly for these tasks, it is better to write a function. Whenever a task has to be performed, then the function may be called and thus reducing size of the code.
- **Modular Programming Approach:** Each function performing one independent task can be considered as a module. Such type of modular programming will increase readability of the program and helps in solving relatively bigger problems.
- **Easier Debugging:** A program divided into modules is always easier to debug as each function is consisting of only few set of statements.
- **Reduced Memory Size:** Every executable statement in a program requires certain amount of space in a computer memory. So, the reduction in code-size of a program will lead to reduction in memory usage too.

- **Division of Work:** A big task may need to be handled by a group of people. Then, with the help of modular programming technique, the problem may be divided and each sub-problem may be given to each individual. Thus, every person in a group will have only a small task to solve. These solutions can then be combined to get the solution of original problem. This technique will save the time.
- **Reusability of Code:** The functions written for general purpose tasks like sorting, searching, finding square root etc. can be used by other programs, whenever required. Thus, the code written once can be re-used in several applications.

4.1.2 General form of Functions

Functions that are the building blocks of C/C++ can be thought of as a user-defined operation. The general form of a function is:

```
ret_type fname(para_list)
{
     //local-variable declarations
     // statements
     return expr;
}
```

Here, ret_type	is the type of the data returned by the function
fname	is any valid name given to the function
para_list	is a list of variable names and their associated data types that receive the values of the arguments when the function is called. There may be zero or more parameters.
return	is a keyword used to return any value to the calling function.

When a function call occurs, the execution of currently active function is suspended temporarily and the program control is passed to the called function. When the execution and/or evaluations of the called function are completed, the program control returns to the calling function and the calling function resumes the execution at the point immediately following the function call.

While discussing with functions, three important concepts to be understood are: prototype of the function, argument passing technique and return value of a function. These concepts are discussed here.

4.1.3 Function Prototype

A function must be declared to the program before it is called. This function declaration consists of a single line

return_type fname(para-list);

This is known as a ***function prototype***. However, the function prototype along with function body, which is known as ***function definition***, can also serve as its declaration.

The return_type of a function can be a basic data type like int, float etc. or a compound type such as float* etc, or it may be a user defined type like enumeration, structure, union etc. A function return type may be void indicating that the function is returning nothing. The parameter list may contain zero or more parameters of different/same data types.

4.1.4 Argument Passing

Functions are allocated storage on the program's runtime stack for their use. That storage remains with the function till the function terminates. After the termination of the function, this storage memory will be available for reuse. This storage area of the function is known as ***activation record*****.** Each function parameter is given memory from this activation records according to its data type.

Argument passing is the process of initializing storage of function parameters with the values of function call arguments. There are three different ways of passing arguments to a function viz.

- **Call-by-value**
- **Call-by-address/pointer**

Each of these methods is discussed later in this chapter.

4.1.5 Returning a Value

The return statement is placed within the function body. It has two important uses:

- It causes immediate exit from the function that it is in. That is, it causes the program control to return to the calling function.
- It can be used to return a value to the calling function from the called function.

There are two different forms of return statement:

• return;	This will simply return to the calling function. Whenever it is not necessary to return any value from the called function to the calling function, the return statement is optional. Because after the last executable statement in the program, by encountering the closing bracket (the flower-bracket **}**), it will go back to the calling function. But, it is a good programming practice to use the return statement at the end of the function.
• return expr;	This will return the value of expr to the calling function. In this case the called function is used at the right-hand-side of the assignment operator in the calling function, so that the returned value of expr is stored in the variable that is at the left-hand-side of the assignment operator.

Note

1. The value of the returning *expression* may be of any basic data type like int, float, double, char etc. Moreover it can be an address i.e. a pointer to any of these data types or any derived data types like arrays, structures, unions or any of the user defined data types.
2. To indicate that the function is not returning anything, void can be used as the return type for function declaration.

main() is known to be is the compulsory function in any C program. This function returns an integer to the calling process i.e., the operating system. Returning a value from the main() is equivalent to calling the exit() function with the same value.

4.2 CATEGORIES OF FUNCTIONS

Though the general syntax of a function includes return type, parameter list and return statement, these are optional. That is a function may or may not contain these parts. Based on this aspect, one can have following four formats of functions:

- **Function without parameters and returning nothing**
- **Function with parameters and returning nothing**
- **Function without parameters but returning a value**
- **Function with parameters and returning a value**

These formats are discussed as here-under:

- **Function without parameters and returning nothing:** A function may not receive any parameter from and also it may not return any value to calling function. Consider the following example:

```
//A Function without parameters returning nothing
#include<stdio.h>

void sum()
{
        int a, b, c;
        printf(”Enter Two values:”);
        scanf(“%d%d”,&a, &b);
        c=a+b;
        printf(“\nSum is : %d”, c);
}

void main()
{
        sum();
}
```

In this example, the main() function is used just to call the function sum().

- **Function with parameters and returning nothing:** A function may take parameters from the calling function but may not return any value to it. For example:

//A Function with parameters returning nothing

```
#include<stdio.h>
void sum(int a, int b)
{
        int c;
        c=a+b;
        printf(”\nSum is : %d”, c);
}

void main()
{
        int x, y;
        printf(”Enter Two values:”);
        scanf("%d%d" , &x, &y);
        sum(x, y);
}
```

- **Function without parameters but returning a value:** Sometimes, a function may not receive any parameter from the calling function. But it may send some value. For example:

//A Function without parameters returning a value

```
#include<stdio.h>
int sum()
{
        int a,b,c;
        printf(“Enter Two values:”);
        scanf(“%d%d”, &a, &b);
        c=a+b;
        return c;
}
void main()
{
        int x;
        x=sum();
          printf(“Sum is : %d”, x);
}
```

- **Function with parameters and returning a value:** A function receiving parameters from the calling function and returning a value to it is used for general-purpose tasks.

```
//A Function with parameters returning a value
#include<stdio.h>
int sum(int a, int b)
{
        return (a+b);
}
void main()
{
        int x, y, z;
        printf(”Enter Two values:”);
        scanf("%d%d", &x, &y);
        z=sum(x,y);
        printf(“Sum is %d”, z);
}
```

4.3 STORAGE CLASSES IN C

To understand the behavior of variables inside the function /block of code, we should understand the scope of variables. There is a concept called as *storage class* in C/C++, which will explain the storage area of the variable, default value of the variable, scope of the variable and life time of the variable. C programming language has four storage classes viz.

– Automatic storage Class
– Register storage class
– Static storage class
– External storage class

If user has not specified the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used. Usually, the default storage class will be *automatic storage class.* Following table briefly illustrates various storage classes.

Storage Class	**Storage Area**	**Default Value**	**Scope**	**Life-time**
Automatic (auto)	Memory	Garbage	Local to the block in which the variable is defined	Till the end of program
Register (register)	CPU Register	Garbage	- Do -	- Do -
Static (static)	Memory	Zero	- Do -	Value of the variable persists between function calls
External (extern)	Memory	Zero	Global	Till the end of program

Automatic storage class
The variables having this storage class is declared using the keyword 'auto'. Variables of this type are stored in the memory and their scope will be local to the block in which they are defined. The default initial value will be any garbage. These variables will alive till the end of the block in which they are defined.

Example:

```
int x =20;
void display()
{
        auto int a = 10;
        printf("%d",a);

        {
                auto int b =20;
                printf("%d",b);
        }
        printf("%d%d",a,x);
        printf("%d", b);      //this line generate error
}
```

Here, *a* is local to the function display(). That is, it can be accessed anywhere inside the function. But *b* is local to the block in which it is declared. So, it can not be accessed outside that block. The variable *x* is a global variable in this example, and it can be accessed anywhere in the program.

Note that, the keyword auto is optional. Even if we declare a variable inside the function as:

```
int a=10;
```

then, the compiler will treat that variable as auto only.

Register Storage Class
The declaration of register variables is done using the keyword register. Such variables are stored in CPU registers. They have the initial default value as any garbage value. The scope of these variables is being local to the block of their declaration. Life time is till the end of the block in which they are defined.

The number of CPU registers is limited for any system. Register variables may be treated as automatic variables when all the registers are busy doing some other task. Usually, the CPU registers in a microcomputer are of 16 bits, they can not hold float or double value. In this case too, the variables declared as register will be treated as auto. Thus, the declaration of variables as register is just a *request* to the compiler but not the *order.*

The variables declared as register will be accessed faster. It is customary to declare the most-frequently-used-variables as register. For example:

```
void disp()
{
        register int i;
        for(i=1; i<=10; i++)
                printf("%d",i);
}
```

The above function is used to display the numbers from 1 to 10. Here, the variable 'i' is declared as 'register'. So, the variable 'i' is going to be stored in CPU register, if available.

Static Storage Class

The static variables are declared using the keyword static. They are stored in memory and initial default value is zero. The scope of the static variables is local to the block but the value of these variables persist between different function calls. For example:

```
void increment()
{
      /* here i is declared as static variable.  So, value of i is initialized when the
      function is called for the first time. Then for next calls, instead of re-declaration
      and re-initialization, the previous value is referred.*/

        static int i=0;
        int j =0;
        i++;
        printf("%d \t %d", i, j);
}
void main()
{
        increment();
        increment();
        increment();
}
```

The output would be:

```
i=1     j=1
i=2     j=1
i=3     j=1
```

When we call the function increment() for the first time, obviously the value of i is printed as 1. However when we call the function for the second time, memory for the variable i is *not allocated* as it is declared as static. Thus, the previous value exists and is incremented.

Hence we will get the output as 2. We get the output as 3 when the function gets called from the third time. But, the variable j is an auto variable. for each function call, memory for j will be re-allocated and initialized to 0. Thus, every time it will be zero only.

External Storage Class
Uses the keyword 'extern' for declaration. Default value being zero and stored in memory. Scope of these variables is global and so accessible for all the functions. The life of these variables exists till the end of the program. External variables can be used in two different contexts.

- In a single file containing C source code, an external variable can be treated as a normal global variable. for example:

```
#include<stdio.h>
int x=10;
void main()
{
        extern int x;
        printf("%d",x);
}
```

Here, the variable x is declared globally. So, when the declaration statement using *extern* keyword within function main() is encountered, the compiler will not allocate memory for x, but it uses the global variable x. Thus, external variable is just like a global variable in this situation.

- Consider one more situation now. Assume that there are two files of which one is containing just a variable declaration. And the other is containing source code and is required to use the variable declared in first file. This is shown below –

```
//file1.c
int x=10;
```

```
//file2.c
#include<stdio.h>
#include "file1.c"
void main()
{
        extern int x;
        printf("%d",x);
}
The output would be – 10
```

Here, the file **file1.c** must be compiled first. Then **file2.c** must be compiled and executed. Memory will be allocated for variable x only in the first file. When the compiler encounters the declaration statement with extern keyword in **file2.c**, then it searches for the variable x in all the included files. When it finds, the same will be used. If the programmer forgets to include the file **file1.c** in **file2.c**, then the compiler will generate the error.

NOTE:

- The usage of different storage classes should be made keeping the following issues in mind –
  - Economic usage of memory space
  - Improvement in speed of execution.

- Considering these two facts, user can use appropriate storage classes as under –
  - If there is a necessity of keeping values of variable between different function calls, use static.
  - If a particular variable, for ex., a counter variable in a loop, is used very often in the program, use register.
  - If a variable is used by many functions, use extern. Note that unnecessary usage of extern will lead huge wastage of memory.
  - If there is no need of above three situations, use auto.

4.4 CALL BY VALUE METHOD

The formal parameters behave like local variables of that function. Thus, in the default initialization method of argument passing, where the values of the arguments are just copied into formal parameters, the change in formal parameters does not reflect the arguments. This is known as ***call-by-value*.** To illustrate this method, consider a program:

Program for addition of two numbers

```
#include<stdio.h>
int sum(int, int);      //function prototype or declaration
void main()
{
        int a, b, c;

        printf("Enter two values:");
        scanf("%d%d", &a, &b);
        c=sum(a, b);          //function call
        printf("\nSum is %d ", c);
}

int sum(int x, int y)          //function definition
{
        int z ;
        z= x+y ;
        return z ;                    //returning the result
}
```

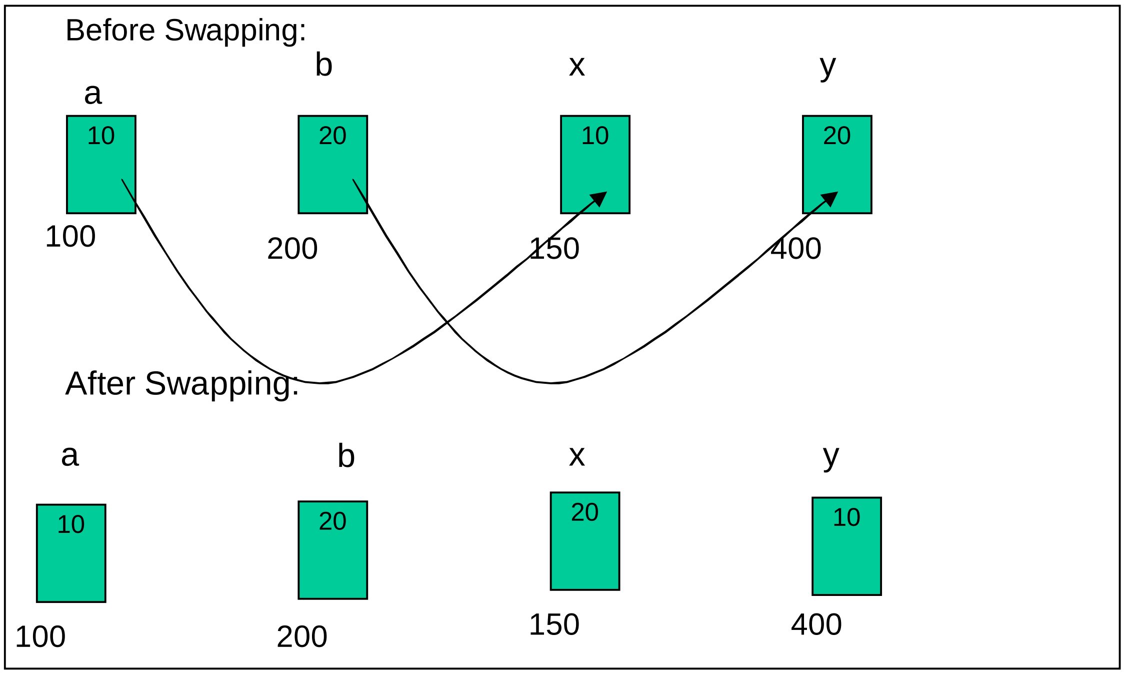
The output may be:

```
Enter two values: 4  8
Sum is 12
```

Since the values of the arguments are copied into formal parameters, they will act as a local copy of the function. As the formal parameters use run-time-stack memory and are destroyed upon the exit of the function, the changes or the manipulations in formal parameters will not affect the actual arguments. Thus when we want to modify the arguments, the call-by-value method is not suitable. Moreover, when a large list of variables must be passed to a function, the time and space costs more to allocate them a memory space in the stack. So, it is not suitable for real world application. To understand this concept, consider following example:

```
//Program that uses call-by-value method
#include<stdio.h>
void swapv(int, int);

void main()
{
        int a = 10, b=25;
        printf("Before Swapping:\n");
        printf("%d \t %d", a, b);
        swapv (a,b);
        printf("\n After swappv() function call :");
        printf("%d \t %d", a, b);
}
swapv(int x, int y)
{
        int temp;
        temp=x;
        x= y;
        y= temp;
        printf("Within function:");
        printf("%d \t %d", x, y);
}
```

In the preceding example, the function **main()** declared and initialized two integers **a** and **b**, and then invoked the function **swapv()** by passing **a** and **b** as arguments to the function **swapv()**. The function **swapv()** receives the arguments **a** and **b** into its parameters **x** and **y**. In fact, the function **swapv()** receives only a copy of the values of **a** and **b** into its parameters. Now, the values of **x** and **y** are swapped. But, the original values at **a** and **b** remains same. This can better be understood using the memory maps of variables as shown in the following figure.



Thus, the modification made to formal parameters within the function will not affect the original arguments, in case of *call-by-value* method. If we need the modification to get affected, we should use *call-by-address/reference* which will be discussed later.

4.5 RECURSION

Recursion is a technique of defining something in terms of itself. In the field of mathematics and computer science, many concepts can be explained using recursion. In computer terminology, if a function calls itself then it is known as recursive function. If the function calls itself directly, then it is known as ***direct recursion***. For example:

```
void myfun()
{
        -----
        -----
        myfun();
        -----
}
```

Here, the function myfun() is calling itself directly. So, we are making use of *direct recursion.* If the function calls itself through another function, then it is known as *indirect recursion.* For example:

```
void myfun()
{
        -----
        -----
        fun();
        -----
        -----
    }

    void fun()
    {
        ------
        ------
        myfun();
        -------
        -------
    }
```

Here, the function myfun() is calling the function fun(), which in turn calls myfun(). Such a recursive call is known as indirect recursion.

Some well known and frequently used recursive functions are described here.

Factorial Function: Factorial of a non-negative integer n is defined as the product of all integers between n and 1. That is:

n! = n(n-1)(n-2)…3.2.1 , for all n>=1 and
0! = 1.

By definition we have, n!= n(n-1)!

Thus, factorial is a recursive function.

Consider one example of finding 5! –

```
5! = 5*(5-1)!
   = 5*4!
   = 5*4*3!
   = 5*4*3*2!
   = 5*4*3*2*1!
   = 5*4*3*2*1*0!
   = 5*4*3*2*1*1
   = 120
```

This procedure can be implemented through program given below:

Factorial of a number

```
#include<stdio.h>

int fact(int n)
{
    if (n==0)
        return 1;
    return n* fact(n-1);
}
```

```
void main()
{
        int n;

        printf("Enter the value of n");
        scanf("%d", &n);

        if(n<0)
                printf("Invalid input");
        else
                printf("\nFactorial of %d is %d ", n, fact(n)) ;
}
```

The output would be:

```
        Enter the value of n: 6
        The factorial of 6 is 720
```

Fibonacci Sequence: A sequence of integers is called Fibonacci sequence if an element of a sequence is the sum of its immediate two predecessors. That is:

f(x)=0	for x=1
f(x)=1	for x=2
f(x)=f(x-2)+f(x-1)	for x>2

Thus the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, ….

Consider an example of finding 6th Fibonacci number.

$$\begin{aligned} f(6) &= f(4) + f(5) \\ &= \underline{f(2) + f(3)} + \underline{f(3) + f(4)} \\ &= 1 + \underline{f(1) + f(2)} + \underline{f(1) + f(2)} + \underline{f(2)+ f(3)} \\ &= 1 + 0 + 1 + 0 + 1 + 1 + f(1) +(2) \\ &= 4 + 0 + 1 \\ &= 5 \end{aligned}$$

The above procedure can be illustrated through the program given below:

Finding nth Fibonacci number

```
#include<stdio.h>
int fibo(int n)
{
        if (n==1)
                return 0;
        else if (n==2)
                return 1;
        return fibo(n-1) + fibo (n-2);
}
```

```
void main()
{
	int n;

	printf("Enter value of n");
	scanf("%d", &n);

	if(n<=0)
		printf("invalid input");
	else
		printf("%d th fibonacci number is %d ", n, fibo(n));
}
```

The output would be:

```
	Enter the value of n : 10
	10th fibonacci number is 34
```

Note

When a function calls itself, a new set of local variables and parameters are being allocated, memory in the stack. Then the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function, only the values being operated upon are new. When each recursive call returns, the old local variables and parameter are removed from the stack and the execution resumes at the point of the function call inside the function.

Some facts about recursive functions are:

1. Recursive functions do not usually reduce the code size.
2. They do not improve memory utilization compared to iterative functions.
3. Many of the recursive functions may execute a bit slower compared to iterative functions because of the overhead of repeated function calls.
4. They may cause a stack overrun, as each new call to the recursive function creates a new copy of variables and parameters and puts them into the stack.

The advantage of using recursive functions is to create clearer and simpler programs. Moreover, some of the algorithms do require recursive functions as the iterative versions of them are quite difficult to write and implement.

While writing a recursive function, a conditional statement has to be included which will terminate the recursive function without using a recursive function call. Otherwise, the recursive function will enter into an infinite loop and will not terminate at all. Any recursive function should satisfy the following conditions:

- In each and every call, the function must be nearer to the solution.
- There should be a non-recursive exit.

4.6 POINTERS

Pointer is a variable that holds the address of another variable. Pointers are used for the indirect manipulation of the variable. Every pointer has an associated data type. The pointer data type indicates the type of the variable whose address the pointer stores. This type is useful in interpreting the memory at a particular address. Pointers are declared using * operator.

General Form:

```
datatype *ptr;
```

Here, *datatype* can by any basic/derived/user defined data type. And *ptr* is name of the pointer variable.

For example, consider the following declarations and statements.

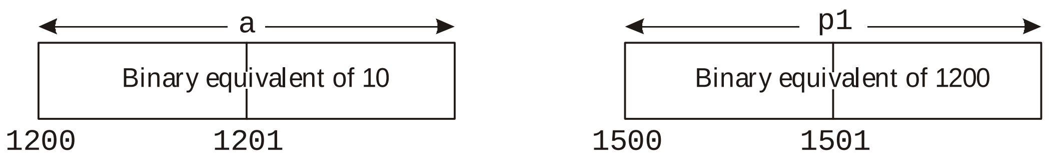
```
int     a=10, *p1;
float   b=4.5, *p2;

p1= &a;
p2=&b;
```

Here, p1 is a pointer variable which holds the address of an integer variable *a*

and p2 is a pointer to hold the address of a floating point variable b.

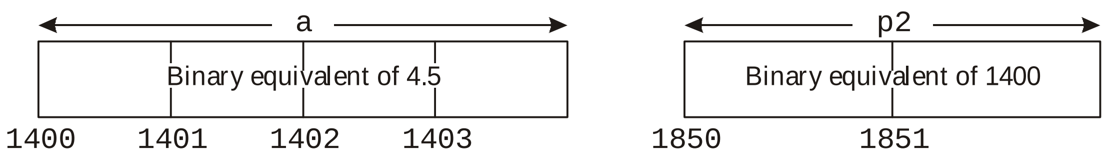
The operator ***** is known as ***dereference operator*** and means ***value at the address***. The operator **&** is known as ***address of operator***.

Now, the memory for *a* and *p1* may be allocated as follows:



The variable *a* occupies two bytes of memory as it is an integer. Suppose that the address of *a* is 1200 (the address is system-dependent and it may vary even for several runs of the same program on the same system). Now, as p1 is assigned the address of *a*, it will have the value as 1200. As p1 stores the address, and the address being an integer, p1 occupies two bytes of memory having its own address.

Similarly for the variables b and p2 memory allocation will be:



Here, the variable b occupies four bytes as it is a floating point variable. Assume that the address of b is 1400. Now, p2 is assigned the address of b. So, p2 will have the value 1400. Note that 1400 is an integer value. Hence, the space occupied by p2 is only two bytes.

The above fact indicates that a pointer variable always occupies only two bytes even if it is pointing to any of the basic data types like int, float, char or any of the derived data types like structures, unions etc.

As a pointer variable is also having an address, we can have a pointer to store the address of another pointer. This is called as a pointer to a pointer. For example, with the above given set of declarations, we can declare two more variables like:

```
int     **ptr1;
float   **ptr2;

ptr1 = &p1;
ptr2 = &p2;
```

Now, ptr1 is called as *a pointer to a pointer to an integer*. This will be having the address of p1 i.e. here, 1500. And ptr2 is called as *a pointer to a pointer to a float*. ptr2 will be having the address of p2 i.e. here, 1850.

Proceeding in the same manner, we can have a pointer to a pointer to a pointer etc.

Consider a simple example to illustrate the usage of pointers.

```
#include<stdio.h>
void main()
{
        int i, x, *pi;
        float f, *pf;

        pi=&i;
        pf=&f;
        printf(”Enter i:”);
        scanf(”%d”, &i);
        x = *pi + 2;
        *pf=4.6;
        printf(“Address of i =%u”, pi);
        printf(“\nAddress of f =%u”, &f);
        printf(“\n Values: i= %d,  f= %f,  x=%d”, *pi, *pf, x);
}
```

The output would be -

```
Enter i:12
Address of i: 0x8fa4fff4      (Note: address may be different for every execution)
Address of f: 0x8fa4ffee
Values:  i=12,  f= 4.6   x= 14
```

Observed in the above example that, the address of any variable can be printed using either the pointer name or variable name with & symbol (that is, pi and &f both can be used to print the address). Similarly, to print the value of any variable, one can use either just name of the variable or pointer name preceded with symbol * (That is, f or *pf are same).

4.6.1 Initialization of Pointers

It is known from the discussion about variables in Chapter 1 that, when a variable is declared it will be containing some garbage value unless it is initialized. Similarly, an un-initialized pointer contains a garbage address. An un-initialized pointer containing garbage address is known as ***dangling pointer.*** Dangling pointer may contain any address which might have been already referenced by some other variable of any program or OS. An un-intentional manipulation of such pointer may cause the working of other programs or even OS. Thus, it is always advised to initialize any pointer at the time of its declaration as –

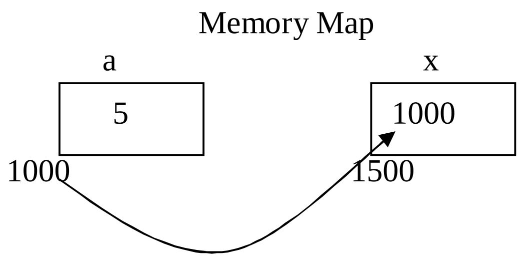
```
data_type *p = NULL;
```

Here, data_type is any basic/derived data type. NULL indicates that the pointer is *pointing no-where in the memory.*

4.7 POINTER AS FUNCTION ARGUMENTS

Pointer can be passed as an argument to a function, and this methodology is known as *call-by-address/reference*. We will now see one simple example for passing a pointer to function.

```
#include<stdio.h>
void fun(int *x)
{
        *x=5;
}

void main()
{
        int a;
        fun(&a);        //address of a is being passed
        printf("a= %d", a);
}
```



In the above example, initially, the variable *a* contains garbage value. Then, the address of *a* is being passed to the function *fun()* which has the parameter as *int *x.* The internal meaning of this parameter passing is actually means,

```
int *x=&a;
```

Hence, a pointer *x* is created which stores the address of *a.* Now, inside *fun()*, the statement

```
*x = 5;
```

Indicates

```
value at the address(x) =5
```

or value at the address(1000) =5
Hence, the variable *a* gets the value 5.

4.7.1 Call by Reference/Address

We have discussed the *call-by-value* method of passing parameters to functions in Section 4.4 and the inconveniences are discussed. To overcome these problems, ***call-by-address*** technique is preferred. Here, instead of the values of actual arguments, their addresses are passed to formal parameter pointers. That is, in this case, the formal parameters must be pointers of specified type, but not the ordinary variables. In this case, the manipulation made inside the function will be stored in the memory affecting the actual arguments.

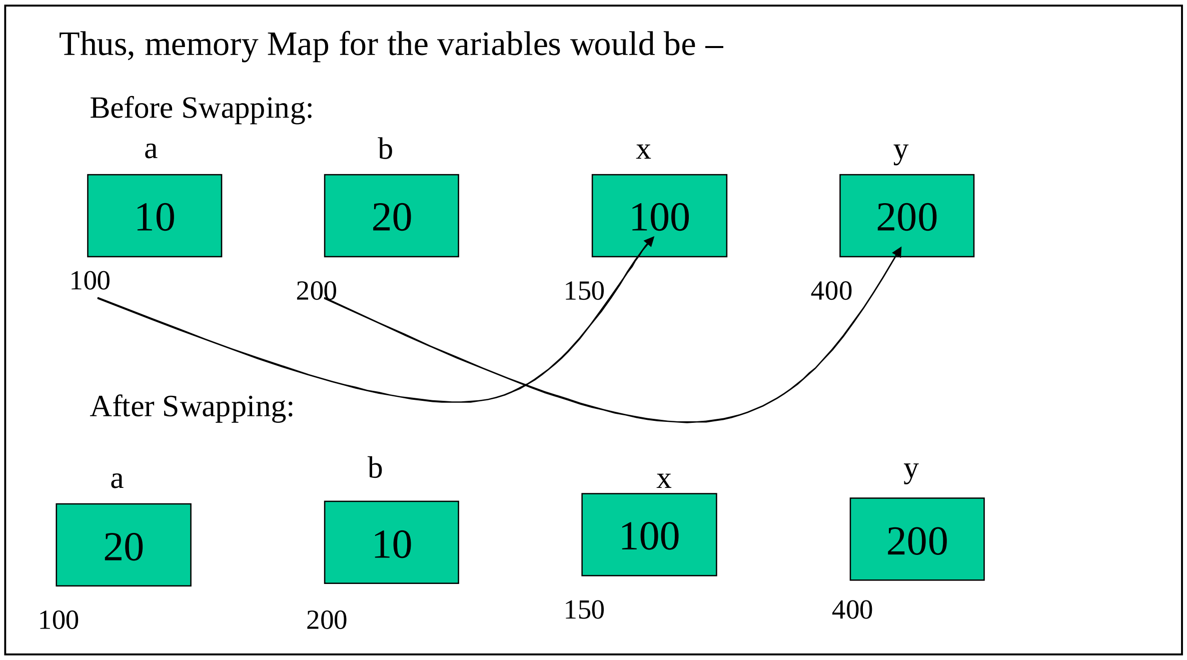
Program that uses call by address method to swap two variables

```
#include<stdio.h>

void swapr(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}

void main()
{
        int a =10, b=20;

        printf("Before swapping: a=%d   b=%d", a, b);
        swapr(&a, &b);        //addresses of a and b are passed
        printf("\nAfter swapping:  a=%d   b=%d", a, b);
}
```

The output would be:

```
a=10  b=20
a=20  b=10
```

In the above example, the function **main()** declared and initialized two integers **a** and **b**, and then invoked the function **swapr()** by passing the addresses of **a** and **b** as arguments to the function **swapr()**. The function **swapr()** receives these addresses into its pointer parameters **x** and **y**. Now, the values stored at the addresses **x** and **y** are swapped. Thus, the actual arguments have been modified. This can better be understood using the memory maps of variables as shown in following figure.



4.8 ARRAYS AND POINTERS

It has been discussed in Unit 3 about accessing array elements using array subscripts. But, there is a speciality about arrays when it comes to pointers. Name of the array itself gives the address of the array and hence allowing the programmer to play with both array and pointer together. Since array is a collection of elements, to access all the elements through pointers, we need to traverse the array. Even for this, only starting address is sufficient. The starting address of an array is known as ***base address*** of the array. The base address of an array can be assigned to a pointer as –

```
        int a[10],*p;
        p=a;
or      p=&a[0];
```

Thus, either the name of array or the address of first element can be assigned to a pointer. To access the elements of array, we need to understand the arithmetic operations on pointers.

4.8.1 Pointer Arithmetic

Arithmetic operations are possible on pointers in the following situations:

1. A pointer can be incremented or decremented. That is, if p is a pointer, then the statements like

   p++;   or   p—;

   are valid. When incremented (or decremented), the pointer will increment (or decrement) by the quantity equal to the size of the data it holds. For example,

```
int *p;
float *ptr;
p++; ptr--;
```

Now, p is incremented by 2 bytes and ptr is decremented by 4 bytes. the same rule is applied when a constant value is added to/subtracted from a pointer as –

```
int *p;
p=p+3;
```

now, p is incremented by 3 positions, that is, 6 bytes.

2. Addition or subtraction of pointers with integers is allowed. That is, if p and q are pointers of same data type, then the statements

   q = p–2;   or   q = p+5;   etc.

   are valid.
3. If two pointers are of same data type, then their subtraction is possible. For example, if p, q and r are pointers of same data type, then the statements like

   r = p – q;

   are valid.
4. Multiplication of pointers with de-referenced values is allowed. For example,

   a = (*p1) * (*p2);

   is valid, where p1 and p2 are pointers and a is a variable of the same data type as of p1 and p2.
5. Pointers can be compared with another pointer using relational operators. That is, the expressions like:

   p >= p2,   p1 != p2   etc.

   are valid.
6. Multiplication or division of a pointer with either another pointer or with an integer is not allowed. For example,

   p1/p2, p1 * p2,   p1/5,   p1*3   etc.

   are invalid.
7. Addition of two pointers is not allowed. That is,

   p1 + p2   can not be done.

If the pointer is addressed to an array element, then using pointer arithmetic like addition or subtraction, we can access the next/previous elements of the array. Consider following example:

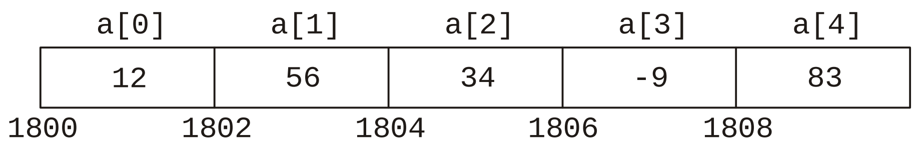
Illustration of pointer to array

```
#include<stdio.h>

void main()
{
    int a[5]={12, 56, 34, -9, 83};
    int i=0, *pa;

        /* address of the first element of the array a, that is the address of a[0] is
        stored in pa.  This is known as base address of the array a[].
        */

        pa=a;                   //or pa=&a[0];

        for(i=0;i<5;i++)
                printf("%d\n", *(pa+i));
}
```

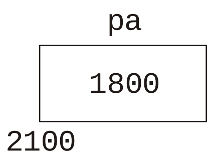
The output would be:

```
12      56      34      –9      83
```

In this program, when we declare an array a[5] and initialize it, the memory is allocated as:



And the memory allocation for the pointer pa will be:



Now, inside the *for* loop, when the statement

```
printf("%d\n", *(pa+i));
```

gets executed for the first time, value of *i* is 0. Hence, the compiler de-references the address **(pa+i)** using the *value at the address(*)* operator. So, the value stored at the address 1800 i.e. 12 is printed.

In the next iteration of the loop, *i* will be 1. But, as per pointer arithmetic, **pa+1** is 1800+2bytes. Thus, the pointer points towards the next location with the address 1802, but not 1801 as normal variable. So, the array element at the address 1802 (i.e., 56) gets printed. The process continues till all the array elements get printed.

Note

1. It can be observed that all the following notations are same:

   a[i] , *(a+i), *(i+a), i[a]

   All these expressions means ith element of an array a.

2. Consider the declarations:

```
int  a[3] = {12, 56, 78};
int *p ;
p = a;
```

Now, there is a difference between the statements:

```
printf("%d", *(p+1)); and
printf("%d", (*p+1));
```

The first expression will increment the pointer variable and so, forces the pointer to point towards next array element. Then the pointer is de-referenced and the value 56 gets printed. But the second expression will de-reference the pointer to get the value 12 and then 1 is added to it. So, the output will be 13.

So, the programmer should take care while incrementing the pointer variable and de-referencing it.

4.8.2 Passing Arrays to Functions

Arrays can be passed to the functions using base address of the array. Consider the following example for illustration:

Program for finding average of n numbers

```
#include<stdio.h>
float Average(int a[], int n)
{
        int  i, sum=0;
        for(i=0;i<n;i++)
                sum=sum+a[i];

        return (float)sum/n;
}

void main()
{
        int x[10], n, i;
        float avg;

        printf("Enter n:");
        scanf("%d", &n);

        printf("\nEnter array elements:");
        for(i=0;i<n;i++)
                scanf(“%d”, &a[i]);

        avg =Average(a, n); //passing base address of the array and value of n
        printf(“Average = %f”, avg);
}
```

In the above example, you may feel that, the pointers are not being used. But, actually, the name of the array which we are passing to the function ***Average()*** itself is the base address of the array or pointer. If you want pure pointer notation, the above program can also be written as –

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

float Average(int *p, int n)
{
        int i, sum=0;
        for(i=0;i<n;i++, p++)
                sum=sum+(*p);

        return (float)sum/n;
}

void main()
{
        int a[20], *pa, n, i;
        float avg;
        clrscr();

        printf("Enter n:");
        scanf("%d", &n);
        pa=a;
        printf("\nEnter values:\n");
        for(i=0;i<n;i++,pa++)        //pa will be pointing last element in last iteration
                scanf("%d", pa);

        pa=a;           //reset pa to base address of array
        avg=Average(pa, n);
        printf("\nAverage=%f", avg);
}
```

We can pass base address of two dimensional (or any multidimensional) array to the function. For example:

Program for reading and displaying a matrix:

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

void read(int x[5][5],int m,int n)              //address of 2-d array is received
{
        int i,j;

        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        scanf("%d",&x[i][j]);
}

void display(int x[5][5],int m,int n)
{
        int i,j;

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                        printf("%d\t",x[i][j]);
                printf("\n");
        }
}

void main()
{
        int a[5][5], m,n;
        clrscr();

        printf("Enter order of matrix:");
        scanf("%d%d",&m,&n);

        printf("\nEnter the elements of first matrix:\n");
        read(a,m,n);

        printf("\nThe matrix is:\n");
        display(a,m,n);         //base address of 2-D array is being passed
}
```

4.9 STRINGS AND POINTERS

String is an array of characters. So, it is possible to have pointer to a string almost in same manner as with arrays. Consider the following example as an illustration:

```
#include<stdio.h>
void main()
{
    char str[20]="Programming", *p;
    p=str;                  //assign base address

    /* till the character stored at p becomes null character ('\0')  print the character and
  increment pointer */

   while(*p!='\0')
   {
           printf("%c", *p);
           p++;
    }
}
```

The output is: Programming

With the help of pointers, it is possible to perform standard operations like finding length of a string, copying the strings, comparing two strings etc. Consider the following example –

Finding length of a string and copying the strings

```
#include<stdio.h>
void main()
{
        char str1[20]="Programming", str2[20],*p1,*p2;
        int len=0;

        p1=str1;
        p2=str2;

        for(len=0; *p1!='\0'; p1++);

        printf("Length = %d", len);

        p1=str1;

        //string copy
        while(*p1!='\0')
        {
          *p2=*p1;
          p1++;
          p2++;
        }
```

```
    p2='\0';        //assign null at the end of second string
    printf("\nString2 is : %s", str2);
}
```

The output is:

```
    Length of first string is: 11
    String2 is: Programming
```

In the above example, usage of both *for* loop and *while* loop is given. The student can choose any of the loops to do the program.

4.9.1 Passing strings to functions

With the help of pointers, we can pass the strings to functions, as string is nothing but a character array. Consider the following example:

```
#include<stdio.h>
void fun(char *s)
{
    printf("%s", s);
}

void main()
{
    char str[20]="hello";
    fun(str);
}
```

The output: hello

Here, in the main() function, actually, we are passing base address of the array *str* and which is being received by the parameter *s* in the function *fun().*

Passing a string to the function can be achieved in any of the two ways:

void fun(char s[ ]) or
void fun(char *s)

4.10 FUNCTIONS RETURNING POINTERS

A function can normally return any basic data type (like int, float etc) or user defined data type (like structure, union etc). Also, a function can return a derived data type like pointer. The pointer/address returned by the function can be a pointer to any data type. Consider an example:

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

```
int * fun()            //function return type is pointer to integer
{
        int a=5, *p;
        p=&a;

        return p;      //p is pointer to int, which is same as return type of function
}

void main()
{
        int *x;                //pointer to int type

        x= fun();              // x receives the address of a from fun()
        printf("%d", *x);      // value at address x is 5

}
```

The comments given in the above program clearly explains the program. In the same manner, one can return **float *, char *** etc. from the functions.

4.11 POINTERS TO FUNCTIONS

When a function returns some value, it will be available within the name of the function. But, a function name refers to a memory location. So, we can have a pointer to a function also. That is, a function can be passed to another function as an argument.

The general form of declaring a function as a pointer is :

```
ret_type (*fn_name)(argument_list)
```

Here,

ret_type	is the return type of the function
fn_name	is valid name given to a pointer to the function
argument_list	is the list of parameters of the function fn_name, if any

To illustrate this concept, consider the program for the simulation of simple calculator.

Simulation of calculator using pointer to functions

```
#include<stdio.h>
#include<conio.h>
/* function that uses a pointer to another function as one parameter and two integer
parameters.  This function is used to call different functions depending on the operator
  chosen by the user. */

float operate(float (*pf)(int, int), int a, int b)
{
        float val;

        val=(*pf)(a, b);        //call for the function using pointer to that function
        return val;
}
float add(int p, int q)         //function to add two integers
{
        return (p+q);
}

float sub(int p, int q)         //function for subtraction
{
        return (p-q);
}

float mul(int p, int q) //function for multiplication
{
        return (p*q);
}

float divide(int p, int q)      //function for division
{
        return (float)p/q;
}

void main()
{
        int x, y;
        char op;
        float result;

        printf("Enter two integers");
        scanf("%d%d", &x,&y);
        printf("\n Enter the operator");
        fflush(stdin);
        scanf("%c", &op);

        switch(op)
        {
      /* depending on the operator entered, the appropriate function is passed as a
      parameter to the operate() function */
                case '+': result = operate(add, x, y);
                            break;
                case '-': result = operate(sub, x, y);
                            break;
                case '*': result = operate(mul, x, y);
```

```
                        break;
                case ‘/’: result = operate(divide, x, y);
                        break;
                default : printf(”Invalid operator”);
                        break;
        }
        printf(”\n The result is: %f”, result);
}
```

The output would be:

```
        Enter two integers :
        5
        4
        Enter the operators : *
        The result is 20
```

In this program, operate() is a function having one of the arguments as a pointer to some other function.

UNIT 6. STRUCTURES AND UNIONS

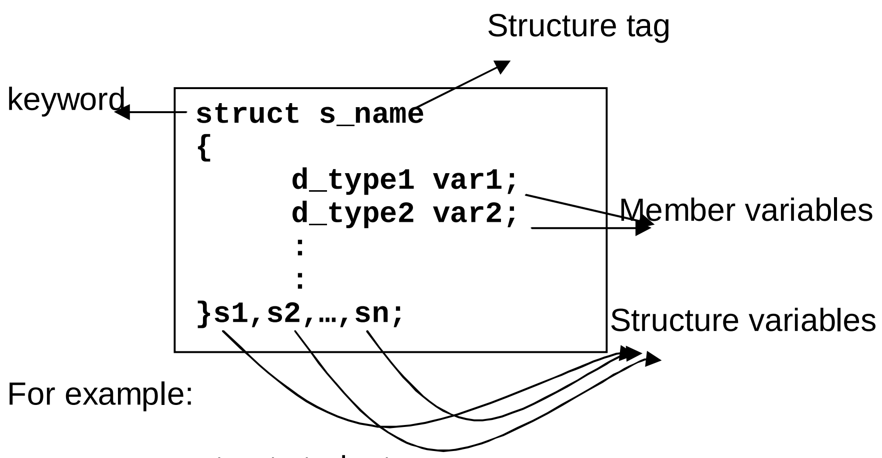
6.1 INTRODUCTION

Along with built-in data types like int, float, char, double C/C++ provides the programmer to define his/her own data-type. These are known as *user-defined data types*. A programmer can derive a new data type based on existing data types. Structures, Unions and Enumerations are considered to be major user-defined data types.

6.2 STRUCTURES

We know that an array is a collection of related elements of same data type. But, if the related elements belong to different data types, then it is not possible to form an array. Consider a situation of storing the information about students in a class. The information may include name of the student, age of the student and marks of the student. It can be easily observed that name is a character array (or string), age might be an integer and marks obtained may be a floating point number. Though name, age and marks are of same student, we can not combine them into an array. To overcome this problem with arrays, C/C++ allows the programmer to combine elements of different data types into a single entity called as a *structure.*

Structure is a collection of inter-related elements of different data types. The syntax of structure is:



```
struct s_name
{
     d_type1 var1;
     d_type2 var2;
     :
     :
}s1,s2,…,sn;
```

For example:

```
struct student
{
        int age;
        float marks;
}s1,s2;
```

Here, student is name of the structure or structure tag.
age, marks are member variables of a structure.
s1, s2 are variables of new data type *struct student*

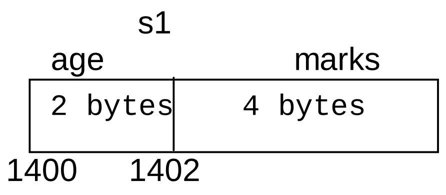
6.2.1 Declaration of Structure variables

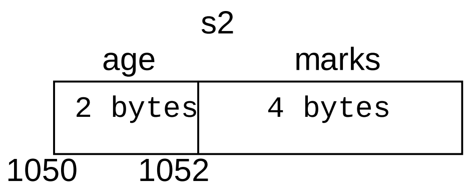
Structure declaration is considered to as defining a new data type. So, to make use of it, the programmer has to declare a variable of this new data type. There are two ways of declaring a structure variable. The programmer can declare variables of structure at the time of structure declaration itself as shown in the above example. Otherwise, one can declare them separately. For example –

```
struct student
{
        int age;
        float marks;
};

struct student s1,s2;
```

Here, struct student is name of new data type and hence using it, one can declare the variables s1 and s2.

Note that declaration of structure is just a prototype and it will not occupy any space in the memory. When a variable of structure gets declared, memory will be allocated for those variables. The memory space required for one structure variable will be the sum of the memories required for all member variables. In the above example, each of the structure variables s1 and s2 takes 6 bytes of memory (2 bytes for age and 4 bytes for marks). The memory map can be given as –





Thus, s1 and s2 are considered to be separate variables.

Note:
After declaring some of the structure variables along with the structure definition, it is possible to declare some more structure variables in a separate statement as shown below:

```
struct student
{
        int age;
        float marks;
}s1,s2;

struct student s3,s4;
```

6.2.2 Structure Initialization

Just like any variable or an array, a structure variable also can be initialized at the time of its declaration. Values for member variables are given in a comma-separated list as we do for arrays. Consider an example:

```
struct student
{
        int age;
        float marks;
}s1 = {21, 84.5};

struct student s2 = {20, 91.3};
```

NOTE:

1. Providing lesser number of values during initialization will make the rest of the variables to hold the value zero. That is,

```
struct test
{
        int a,b,c;
}t = {12, 45};
```

   Now, the values of a, b and c will be 12, 45 and 0 respectively.

2. It is not possible to give initial value to a member variable within a structure definition. Because, structure definition is just a logical entity and member variables will not be having any physical location until the structure variables are declared. Thus, following attempt is ***erroneous.***

```
struct test
{
        int a=10                        //error
        float b= 12.5;          //error
};
```

6.2.3 Accessing Member Variables

Member variables of a structure can not be accessed just by referring to their names. The reason is obvious. Refer to the memory map given in section 6.2.1**.** It can be observed that memory location for the member variable age of s1 is different from that of s2. Similarly for the member marks. Hence, it can be understood that each of the member variable is associated with a structure variable. Or in other words, every structure variable has its own copy of member variables. Thus, for accessing any member variable, its associated structure variable also should be specified. This is done with the help of ***dot (.) operator.*** The syntax is:

Structure_variable.member_variable

Note that a member variable of a structure can be used in arithmetic/logical/relational expressions. The following example illustrates the accessibility and usage of member variables.

Example for accessing member variable

```
#include<stdio.h>

struct student
{
        int age;
        float marks;
}s1, s2;

void main()
{
        s1.age=21;
        s2.age=s1.age;
        printf("Enter marks of students:");
        scanf("%f%f", &s1.marks, &s2.marks);

        printf("\nAge1= %d   Age2=%d", s1.age, s2.age);
        printf("\nMarks1= %f   Marks2=%f", s1.marks, s2.marks);

        if(s1.marks>s2.marks)
                printf("\n First Rank is Student1");
        else
                printf("\n First Rank is Student2");
}
```

The output would be:

```
Enter marks of students: 81.5      92.3
Age1=21                      Age2=21
Marks1=81.5         Marks2=92.3
First Rank is Student2
```

6.2.4 Structure Assignment

In some situations, the programmer may need two structure variables to have same values for all the member variables. Then, instead of assigning each member separately, it is possible to assign whole structure variable to another structure variable of same type. For example:

```
        struct student
        {
                int age;
                float marks;
        }s1, s2={21,92.3};
```

Now, the statement

```
s1=s2;
```

will assign values of all members of s2 to the corresponding members of s1.

6.2.5 Arrays within Structures

A member variable of a structure can be an array of any data type. Consider the following example:

Example for array as a member of structure

```
#include<stdio.h>

struct student
{
	char name[20];			//array as a member
	int age;
	float marks;
}s1, s2={“Jaggu”,22,92.3};


void main()
{
	s1.age=21;
	printf(”Enter name of student:”);
	scanf(“%s”, s1.name);

	printf(”\nEnter marks of student:”);
	scanf(“%f”, &s1.marks);

	printf(”\n Student Information:\n”);
	printf (”\n Name \t\t Age \t\t Marks \n”);
	printf (“%s \t %d \t %f \n”, s1.name, s1.age, s1.marks);
	printf (“%s \t %d \t %f \n”, s2.name, s2.age, s2.marks);
}
```

The output would be:

```
Enter name of student: Ramu
Enter marks of students: 81.5
Student Information:
Name        Age         Marks
Ramu        21          81.5
Jaggu       22          92.3
```

6.2.6 Arrays of Structures

We know that array is a collection of elements of same data type and the structure is a collection of items of different data types. Some times, it may be necessary to have several structure variables. For example, if the information (like name, age and marks) about 60 students studying in 6th Semester is required, creating 60 structure variables is absurd. Instead, one can create an array of structure variables. Memory allocation for array of structures will be in contiguous locations. This concept is illustrated in the following example:

Example for array of structures

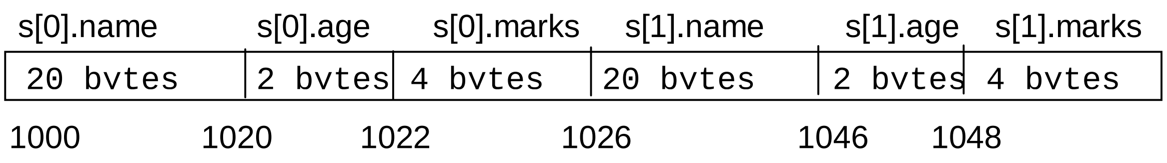
```
#include<stdio.h>

struct student
{
        char name[20];
        int age;
        float marks;
}s[2];          //array of structures

void main()
{
        int i;

        for(i=0;i<2;i++)
        {
                printf(”Enter name of student”);
                scanf(“%s”, s[i].name);

                printf(”\nEnter age of student”);
                scanf(“%d”,&s[i].age);

                printf(”\nEnter marks of student”);
                scanf(“%f”, &s[i].marks);
        }

        printf(”\n Student Information:\n”);
        printf (”\n Name \t Age \t Marks \n”);

        for(i=0;i<2;i++)
                printf(“\n%s \t %d \t %f”, s[i].name, s[i].age, s[i].marks);
}
```

The output would be:

```
        Enter name of student: Vidya
        Enter age of student: 21
        Enter marks of student: 81.5
```

```
Enter name of student: Jaggu
Enter age of student: 22
Enter marks of student: 92.3

Student Information:
Name        Age         Marks
Vidya       21          81.5
Jaggu       22          92.3
```

In the above example, memory allocation for the array s[2] might be as follows:



Note that array of structure variables can also be initialized just like a normal array. For example:

```
struct student
{
        int age;
        float marks;
}s[3] = {
                {21, 81.5},
                {22, 92.3},
                {25, 88.4}
           };
```

Then, the individual elements will be assigned the values as –

s[0].age=21	s[1].age=22	s[2].age=25
s[0].marks=81.5	s[1].marks=92.3	s[2].marks=88.4

Note also that, un-initialized members will take the value as zero.

6.2.7 Nested Structures

A member variable of a structure may be a variable of type another structure. This is known as nesting of structures. For example:

```
struct address
{
        int d_no;           ------------    2 bytes  ⎫
        char street[20];    ------------   20 bytes  ⎬42 bytes
        char city[20];      ------------  20 bytes   ⎭
};
```

```
struct employee
{
        char name[20];          ------------   20 bytes
        int num;                ------------    2 bytes    68
        float sal;              ------------    4 bytes   bytes
        struct address add;     ------------   42 bytes
}emp;
```

The total memory allocated for the variable emp would be 68 bytes. The memory map can be:

emp.name	emp.num	emp.sal	emp.add.d_no	emp.add.street	emp.add.city
20 bytes	2 bytes	4 bytes	2 bytes	20 bytes	20 bytes

1000 1020 1022 1026 1028 1048

Here, the structure address contains the members d_no, street and city. Then the structure employee contains the members name, d_no, sal and add. Here, add is a variable of structure address. To access the members of the inner structure one has to use the following type of statements:

```
emp.add.d_no
emp.add.street
emp.add.city   etc.
```

6.2.8 Passing Structure to Function

Just like a normal variable, it is possible to pass a structure variable as a function parameter either using call-by-value method or call-by-address method. Passing of structure to a function can be done in two different ways viz.

- **Member variables as arguments**: Individual members of a structure can be passed to function as arguments. In this situation, members of structure are treated like any normal variable. Consider an example –

  Structure member as a function argument

```
#include<stdio.h>

struct add
{
    int a, b;
};

int sum(int x, int y)
{
    return (x+y) ;
}

void main()
{
    struct add var ;
    int s ;

    printf("Enter two numbers :" );
    scanf("%d%d", &var.a, &var.b);
    s=sum(var.a, var.b);

    printf("Sum is: %d ",s);
}
```

The output would be –

```
Enter two numbers : 5        8
Sum is: 13
```

- **Whole structure variable as an argument:** Instead of passing each member of a structure to a function, entire structure variable can be passed. For example:

  Structure variable as a function argument

```
#include<stdio.h>

struct add
{
    int a, b;
};

int sum(struct add A)     //parameter is a structure variable
{
    return (A.x + A.y) ;
}

void main()
{
    struct add var ;
    int s ;

    printf("Enter two numbers :" );
    scanf("%d%d", &var.a, &var.b);

    s=sum(var);
    printf("Sum is: %d ",s);
}
```

The output would be –

```
Enter two numbers : 5          8
Sum is: 13
```

6.2.9 Pointer and Structures

We can create a pointer to store the address of a structure variable. To access the member variables of a structure using a pointer, we need to use either an indirectional operator -> or the combination of * and dot(.). Let us consider an example:

```
#include<stdio.h>

struct student
{
        char name[20];
        int age;
};

void main()
{
        struct student s={"Ramu", 22};
        struct student *p;

        p= &s;
        printf("Student name =%s", p ->name);
        printf("\nStudent age =%d", (*p).age);
}
```

In the above example, note the usage:

```
        (*p).age
```

Here, the pointer *p* is dereferenced first, and then dot operator is used to get the member variable *age.* Instead of two operators * and dot, we can use single indirectional operator (arrow mark) to achieve the same. That is, we can use

```
        p->age,  p->name etc.
```

We can even create a pointer to array of structures also. For example (with respect to structure declared in the above example):

```
                struct student s[10], *p;
                p = s;      // base address of the array s is assigned to p
```

Then, for accessing array members, we can use:

```
        for(i=0; i<n;i++)
                printf("%s %d", p[i]->name, p[i].age);
```

The similar approach can be used while passing structures to functions via call by address method.

6.2.10 Bit Fields

Definition: Bit fields are used in structures to compact the size of a structure variable.

As we have discussed till now, structures are normally used to combine the related data elements of different data types. Most of the times, the member variables declared inside the structure are more in size than required. For example, consider the following structure:

```
struct student
{
	int age;
	int gender;
};
```

Here, age of a student can be at the most, say, 25. To store the number 25, we require only 5 bits (equal to the value 32) space in memory. But the integer occupies 2 bytes (or 4 bytes – depending on machine) and it can store the values from – 32768 to +32767. But, what we need the maximum value is 25 (only positive value, as age cannot be negative). So, out of 16 bits (or 32 bits), we may use only 5 bits and remaining 11 bits will be wasted. Similarly, the gender may take the values '0' or '1' representing Male or Female. To represent just two values, we require only one bit space (either 0 or 1), and remaining 15 bits will be wasted. To avoid such a memory loss, the concept of bit fields had been introduced in C language, as the memory was a very crucial aspect when C has been designed. Moreover, even today, in most of the hardware applications (like chip design, micro controllers etc) one need to compact the structure size for optimal usage of memory. These requirements lead us to usage of bit fields.

Consider an example to use bit fields:

```
	struct student
	{
		int age: 5;		//indicates, only 5 bits has to used for age
		int gender: 1;		// only one bit for gender.
	};
```

Now, let us create a variable of this structure:

```
		struct student s;
```

Without bit fields, the variable *s* would require 4 bytes (2 for age and 2 for gender). But, we have specified now as *age* should use just 5 bits and *gender* should use only 1 bit. So, total will be just 6 bits. **But, that doesn't mean that *s* requires only 6 bits now.** Because, now, the size of *s* is depending of hardware architecture of the machine on which you are running the program:

- It may take one byte (if architecture is designed to identify every byte of data)

- o It may take two bytes (if architecture is designed to identify minimum of two bytes)

But, we can assure that, most of the times, bit fields usage will reduce the total size of the structure.

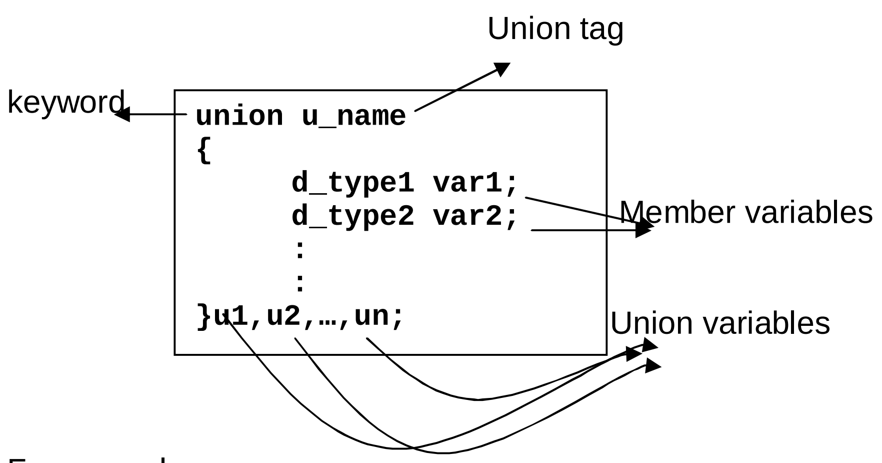
Also, due to bit pattern storage, we can't read the bit-fields from the keyboard directly using scanf() function. Either we can assign the value (hard-coded value) or take any ordinary variable to read from console, and then assign that value to a bit field.

Consider the following example to illustrate bit fields:

```
#include<stdio.h>
struct student
{
        int age:5;
        int gender: 1;
};

void main()
{
        struct student s;
        int a;

        printf("Size of a structure: %d", sizeof(s));        //may print 1 or 2

        // scanf("%d", &s.age);              //generates error!!
        scanf("%d", &a);
        s.age=a;                        //read using ordinary variable and then assign

        s.gender=0;                     //hardcoded value: 0 for Male and 1 for Female

        printf("Age =%d", s.age);
        printf("Gender =%d", s.gender);
}
```

6.3 UNIONS

Union is also a derived data type just like structures. Union can be defined as a collection of several items of different data types. The syntax of union is:



```
union u_name
{
        d_type1 var1;
        d_type2 var2;
              :
              :
}u1,u2,…,un;
```

For example:

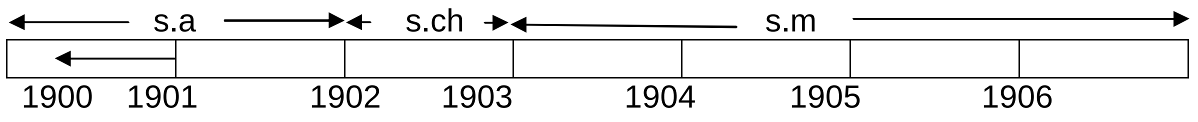
```
union student
{
        int age;
        float marks;
}u1,u2;
```

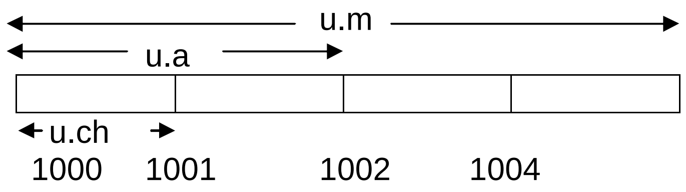
Here, student is name of the union or union tag.
age, marks are member variables of a union.
u1, u2 are variables of new data type union student

It is observed in the previous discussion that memory for member variables of a structure are allocated in contiguous locations. But, when a union variable is declared, the computer allots memory space for only one member variable that requires maximum number of memory locations. Therefore only one of its members is stored in memory for a given moment of time. Thus, we can access only one member at a time. To understand the difference between structure and union, consider the following example:

```
struct st                               union un
{                                       {
        int a;                                  int a;
        char ch;                                char ch;
        float m;                                float m;
}s;                                     }u;
```

Now, the memory for structure variable s is 7 bytes and that for union variable u is only 4 bytes. The memory map for both s and u will be like –





Memory map for structure and union

Thus, usage of unions will save the memory. So, if all the members need not be accessed at a time, then it is better to use union than structure.

Accessing member variables of union is exactly same as that with structures. It is possible to have array within union, array of union variables, union as function argument etc. Consider following program as an illustration for unions.

Example for union

```
#include<stdio.h>

union student
{
        char name[8];
        int age;
}s;

void main()
{

        printf(”Enter name of student:”);
        scanf(“%s”, s.name);
        printf(”\nName is: %s”, s.name);

        printf(”\nEnter age of student:”);
        scanf(“%d”, &s.age);
        printf(”\nAge is:”, s.age);
}
```

The output would be:

```
        Enter name of student: Jaggu
        Name is: Jaggu
```

```
Enter age of student: 21
Age is: 21
```

The differences between structure and union are elaborated as below

Difference between Structure and union

Structure	Union
1. Memory for a structure variable is sum of all the memories of its member variables.	1. Memory for a union variable is maximum memory required for any one of its member variable.
2. Uses more memory.	2. Uses Less memory.
3. All the members have separate memory locations.	3. Members will share the memory.
4. All the members can be stored and accessed at a time.	4. Only one member can be stored and accessed at a time.

NOTE: For the question on difference between structure and union, the student is supposed to give one example and the respective memory map.

6.4 ENUMERATIONS

An enumeration is a set of named integer constants that specify all the legal values a variable of that type may have. The enumerated data type gives an opportunity to define a user's data type. The general form of enum declaration would be:

```
enum  e_name {list_of_values}list_of_var;
```

Here, enum	is a keyword
e_name	is valid name given to enumerated data type
list_of_values	gives the list of all the values that this new data type can take.
list_of_var	is a list of variables of this new data type.

For example:

```
enum mar_status
{
        married, unmarried, divorced, widowed
};
enum mar_status p1, p2;
```

Here p1 and p2 are two variables of enumerated data type mar_status. Values can be given to these variables as:

```
p1 = married;
```

```
p2 = divorced;
```

No values other than those specified in the list can be given. That is:

```
p1= test;
```

is ***invalid*** assignment.

When the values are assigned to enumerated data type and if we try to print them, they will not be the string values as we assigned, instead they will be integers. To illustrate the usage of enumerated data type, consider the program given below:

Illustration of enumerators

```
#include<stdio.h>

void main()
{
        enum emp_dept
        {
                Development, Testing, Finance
        };

        enum emp_dept e1, e2;

        e1=Development;
        e2=Finance;
        printf(”e1= %d”, e1);
        printf(”e2= %d”, e2);
}
```

The output would be:

```
        e1 = 0
        e2 = 2
```

Enumeration data types are used to improve the readability of the program.

6.5 TYPE DEFINITION

The ***type definition*** mechanism provides a facility to redefine the name of an existing variable type. It gives a mnemonic synonym for a basic or user defined data type. The general form is:

```
typedef data_type new_name;
```

Here,	typedef	is a keyword
	data_type	is the basic/user-defined data type
	new_name	is the name given to data_type.

For example:

```
typedef unsigned long int USLI;
```

After this statement, USLI becomes an alias name for unsigned long int. Now, we can declare variables of type unsigned long int as:

```
USLI v1, v2;
```

Thus, typedef provides a short and meaningful way to call a data type.

UNIT 7. FILES AND DMA

7.1 INTRODUCTION TO FILES

A file is a collection of logically related information. For example:

- A student file containing information pertaining to the students like name, age, USN, marks etc.
- An employee file with employee names, designation, salary etc.
- A product file containing product name, make, batch, price etc.
- A census file containing house number, names of the members, age, gender, employment status, children etc.

Files are necessary to store the information on a disk and to use the same for future purpose. That is, reading and writing the data from and to console (i.e. console I/O operations) is not always useful in real time applications. Hence, I/O operations on files make sense in the study of any programming language.

C programming language provides a set of built-in functions to perform I/O operations on files. The simplicity of file input/output in C lies in the fact that it essentially treats a file as a stream of characters, and accordingly facilitates input/output in streams of characters. Functions are available for character-based I/O, string-based I/O, formatted I/O etc.

7.1.1 Defining a file

Every operation on files has to be done with the help of a pointer to an inbuilt structure called as FILE. The structure FILE contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors, or end of file have occurred. This structure FILE is defined in the header file stdio.h. Thus, in any file related program in C language, we must first create a pointer as shown below:

```
FILE *p = NULL;
```

Here, p is known as a file pointer, and normally initialized to NULL at the beginning.

7.1.2 Opening a file

Before a file can be read from, or written into, a file has to be opened using the library function **fopen( ).** The syntax is:

```
FILE *p = fopen(“filename”, “mode”);
```

Here,

filename	is name of the file which you are trying to open. The *filename* should consists of a valid extension and path(if necessary)
mode	is a string representing the purpose (read/write/append etc.) for which you are opening a file. This string must be one among the predefined set of modes given by C.

The function fopen( ) takes a file name as an argument, and interacts with the operating system for accessing the necessary file. And it returns a pointer of type FILE that will be used in subsequent input/output operations on that file. Note that, this pointer is the only medium through which all operations on file has to be done. Due to any reason, if file could not be opened, then *fopen()* function returns NULL and hence, file can't be handled any more.

Some of the file opening modes have been listed below:

Mode	Access	Explanation
"r"	Read only mode	Opening a file in "r" mode only allows data to be read from the file. File can be opened in "r" mode only if file exists in the specified path, and the user has permission to open that file in read mode. Otherwise, NULL will be returned.
"w"	Write only mode	If file by that name exists, its contents are overwritten. Otherwise, a new file is created and then written. If file by that name exists, but user don't have permission, then NULL is returned.
"a"	Append mode	Allows data to be appended to the end of the file, without erasing the existing contents. It is therefore useful for adding data to existing data files.
"r+"	Read + Write mode	This mode allows data to be updated in a file
"w+"	Write + Read mode	This mode works similarly to the "w" mode, except that it allows data to be read back after it is written.
"a+"	Read + Append mode	This mode allows existing data to be read, and new data to be added to the end of the file.

7.1.3 Closing a file

Every file that got opened for any purpose must be closed properly using a function ***fclose().*** This function disassociates the file pointer from the actual file. Also, all internal buffers associated with the file pointer are disassociated from it and are flushed. The content of any unwritten output buffer is written and the content of any unread input buffer is discarded. The syntax is:

```
fclose(fp);
```

Here, *fp* is a file pointer.

7.1.4 Input and Output operations

I/O operations on files can be done in many ways viz. character based I/O, string based I/O, formatted I/O etc. We will discuss each one of them here under:

Character based I/O:
In C, character-based input/output from and to files is facilitated through the functions *fgetc()* and *fputc()*. The syntax for *fgetc()* is given below:

```
char ch;
ch=fgetc(p);
```

Here, p is a file pointer pointing to a file which would have opened in a read mode. The function *fgetc()* will read single character from the file and assigns it to *ch*. Also, ***the function fgetc() will push the file pointer towards next location in the file.*** Normally, *fgetc()* function is executed till the end of file (EOF) is reached.

The syntax for *fputc()* is:

```
        fputc(ch, p);
```

Here, *ch* is a character which needs to be written into a file. And *p* is a file pointer pointing to a file which would have opened in a write/append mode. The *fputc()* function writes the character *ch* into a file and also, it pushes the file pointer towards next location in the file.

Consider a program for copying content of one file into another to illustrate fgetc() and fputc() functions.

NOTE: To execute the below program, the student should first create one file viz. *source.txt* in the edit window. Then execute the program. Again he/she should check the output file i.e. *dest.txt* at edit window itself.

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

void main()
{
        FILE *fs,*fd;
        char ch;
        clrscr();

        fs=fopen("source.txt","r");
        if(fs==NULL)
        {
                printf("File can not be opened to read");
                exit(0);
        }

        fd=fopen("dest.txt","w");
        if(fd==NULL)
        {
                printf("\nFile can not be opened to write");
                fclose(fs);                     //source.txt would have already opened
                exit(0);
        }

        while(1)
        {
                ch=fgetc(fs);

                if(ch==EOF)
                        break;

                fputc(ch,fd);
        }

        printf("File copied successfully!!!");
}
```

Line based I/O:

C provides the functions **fgets( )** and **fputs( )** for performing line input/output from/to files. The prototype declaration for fgets( ) is given below:

char* fgets(char *line, int maxline, FILE *fp);

Here,

char* line	the string into which data from the file is to be read
int maxline	the maximum length of the line in the file from which data is being read
FILE *fp	is the file pointer to the file from which data is being read

The *fgets()* function returns a string (char *) that has been read from a file. It reads the next input line (including the newline) from file fp into the character array ***line**.* At most ***maxline-1*** characters will be read. The resulting line is terminated with **'\0'**. Normally **fgets( )** returns line; on end of file, or error it returns **NULL**.

For output, the function **fputs( )** writes a string (which need not contain a newline) to a file:

int fputs(char *line, FILE *fp)

It returns **EOF** if an error occurs, and non-negative otherwise.

Following is the program to copy content of one file to other, but copy is done line by line.

NOTE: To execute the below program, the student should first create one file viz. *source.txt* in the edit window. Then execute the program. Again he/she should check the output file i.e. *target.txt* at edit window itself.

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 81                    //indicates length of one full line in editor

main( )
{
        FILE *fp1, *fp2;
        char string[MAX];

        fp1 = fopen( “source.txt”, “r”);
        fp2 = fopen( “target.txt”, “w”);

        if(fp1==NULL)
        {
                puts(“file can’t be opened”);
                exit(0);
        }

        if(fp2==NULL)
        {
                puts(“file can’t be opened”);
                fclose(fp1);
                exit(0);
        }

        while ((fgets(string, MAX, fp1)) != NULL)
                fputs (string, fp2);

        fclose(fp1);
        fclose(fp2);
}
```

Formatted I/O:

C facilitates data to be stored in a file in a format of your choice. You can read and write data from/to a file in a formatted manner using fscanf( ) and fprintf( ). These functions are similar to their console counter parts (printf() and scanf()) except that they require one additional argument – file pointer.

The syntax of *fprintf()*:

```
        fprintf(fp, “format string”, v1, v2, …);
```

Here,

fp	is a file pointer
format string	is a list of format specifiers like %d, %f etc.
v1, v2…..	are the variables to be written into a file.

The syntax of *fscanf()*:

fscanf(fp, “format string”, &v1, &v2, …);

Here,

fp	is a file pointer
format string	is a list of format specifiers like %d, %f etc.
v1, v2…..	are the variables to be read from a file.

fscanf( ) returns the value EOF upon encountering end-of-file.

Consider the following program:

Declare a structure *student* with the members *name, USN* and *marks.* Write a C program to read *n* student records and store the data into a file.

NOTE: *After executing the following program, the student must exit from the Turbo C window. Then at DOS prompt, give the command as*

edit student.txt

Now, the file student.txt will be opened and it will be containing the output.

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

struct student
{
        char name[20];
        int age;
};

void main()
{
        struct student s[10];
        FILE *fp;
        int i,n;
        clrscr();

        printf("Enter number of students:");
        scanf("%d",&n);
        printf("\nEnter Student details:\n");
        for(i=0;i<n;i++)
        {
                printf("\nEnter Name:");
                scanf("%s",s[i].name);
                printf("\nEnter age:");
                scanf("%d",&s[i].age);
        }

        fp=fopen("student.txt","w");
        if(fp==NULL)
        {
                printf("\nFile can not be opened");
                exit (0);
        }

        for(i=0;i<n;i++)
                fprintf(fp,"%s \t %d\n",s[i].name, s[i].age);
        printf("\nFile written successfully!!!");
        fclose(fp);
}
```

Now, to read the file contents using *fscanf()*, write the following program:

```
        #include<stdio.h>
        #include<conio.h>
        #include<stdlib.h>
        void main()
        {
                FILE *fp;
                char name[20];
                int age;
                clrscr();

                fp= fopen("student.txt", "r");
                if(fp==NULL)
                {
                        printf("File opening failed");
                        exit(0);
                }

                printf("NAME \t AGE\n");

                while(!feof(fp))
                {
                        fscanf(fp, "%s%d", name, &age);
                        printf("%s\t%d\n", name, age);
                }
                fclose(fp);
        }
```

Binary Files I/O:

The programs discussed till now were working on text files. But, C provides I/O operations on binary files (data is stored in binary format) also.

Moreover, while reading/writing the information stored in the structure (previous two examples) we have used individual structure members. But, C provides two more functions ***fread()*** and ***fwrite()*** to read and write entire record from/to a file. These functions are normally helpful in dealing with databases.

The syntax ***fread()*** function:

```
fread(&rec, size, number, fp);
```

Here,

rec	is record (normally, a structure variable) that needs to be read from file
size	is size (in bytes) of one record
number	indicates number of records to be read at a time
fp	is a file pointer

The *fread()* function returns the total number of records that has been read successfully. Hence, while using *fread()* function, we can't check for end-of-file, instead, we need to check till it returns true(non zero) value

The syntax ***fwrite()*** function:

```
fwrite(&rec, size, number, fp);
```

Here,

rec	is record (normally, structure variable) that needs to be written into file
size	is size (in bytes) of one record
number	indicates number of records to be written at a time
fp	is a file pointer

Below given program illustrates ***fread()*** and ***fwrite()*** functions along with binary files.

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

struct student
{
        char name[20];
        int age;
};

void main()
{
        struct student s, s1;
        FILE *fp;
        char opt='y';
        clrscr();

        fp=fopen("emp.dat", "wb");      // open a binary file in write mode
        if(fp==NULL)
        {
                printf("failed");
                exit(0);
        }

        while(opt=='y')
        {
                printf("\nEnter student name and age:");
                scanf("%s%d", s.name, &s.age);
                fwrite(&s, sizeof(s), 1, fp);

                printf("\nDo you want to add another record? (y/n):");
                fflush(stdin);
                scanf("%c", &opt);
        }
        fclose(fp);

        /*  Second half of the program for reading the data  */

        fp=fopen("emp.dat", "rb");   // open a binary file in read mode
        if(fp==NULL)
        {
                printf("failed");
                exit(0);
        }

        while(fread(&s1, sizeof(s1), 1, fp)==1)
                printf("\n%s \t%d", s1.name, s1.age);

        fclose(fp);
}
```

The above program opens a file ***emp.dat*** in a binary mode for writing (**wb**). Data for structure members are read from the keyboard first. Then, they are written as one full record using the statement –

fwrite(&s, sizeof(s), 1, fp);

Here, *s* is a structure variable (one full record containing name and age). Second argument is size of one structure variable in bytes. The third argument ***1*** indicates we are writing only one record now.

After finishing only the writing process, if you open *emp.dat* file in any text editor (notepad, wordpad, Microsoft word, DOS editor etc), you can find some jumbled characters. Because, the file is in **binary** mode, and you are trying to open using a **text** editor. Hence, there won't be any compatibility. But, you can read the data from the file *emp.dat* using ***fread()*** and display the records appropriately. For this, you need to execute, the second half of the program given above.

While reading, notice the *while* loop :

while(fread(&s1, sizeof(s1), 1, fp)==1)

The loop is executed till *fread()* successfully reads one record at a time. If no record found (or end of file is reached), then *fread()* function will return zero, and hence the loop will be terminated.

7.2 COMMAND LINE ARGUMENTS

Command line arguments are the arguments to main() function. Normally, for user defined function is called from a calling function by passing arguments (if any). But, main() function is called by the OS, and the programmer don't have any chance to call it. Hence, if the programmer wants to pass any arguments (or to give input) to the main() function, he has to do so by using command line arguments. For doing so, the main() function has to be defined with a syntax as given below:

```
void main(int argc, char* argv[])
{
        ----------------
        -------------
}
```

Here, *argc* is an integer that will receive number of arguments provided (argument count)
argv is array of strings (char*) which will hold the arguments (argument value)

Note that, the above prototype of main() function can't be modified. That is, you can not give arguments of your choice. That is, main() function can receive only two arguments, of which, the first is of type integer, and the second is array of strings.

Consider the following example to illustrate the same:

```
#include<stdio.h>
void main(int argc, char* argv[])
{
        printf("Hello %s", argv[1]);
}
```

Instructions to execute above program:

1. Save the program, say, **test.c.**
2. Compile the code
3. Execute the code, so that **test.exe** file will be created in the current working directory.
4. Quit the Turbo (or any other) IDE, and go back to command prompt. If you are running the code in Turbo, normally the command prompt may look like:
   C:\TC>
5. Now, give the command as:
   C:\TC>test Ram
6. You will get the output as **Hello Ram.**

Note that, the number of arguments will be counted from the command prompt itself. That is, in the line –

C:\TC>test Ram

The first part **C:\TC>test** is treated as first argument (argv[0]) and **Ram** is treated as second argument (argv[1]). Hence, normally in any such programs, we have to consider the values from second arguments as the real inputs to main() function.

Consider one more program:

```
#include<stdio.h>

void main(int argc, char* argv[])
{
        int i;

        printf("Total number of arguments: %d", argc);
        printf("The arguments provided are:");
        for(i=1; i<argc; i++)          // note that i started with 1 and till argc
                printf("%s\n", argv[i]);
}
```

If you provide the set of values as:

C:\TC>test hello how are you?

Now, there are totally 5 arguments, each one is separated by a blank space. But, in the program, we have started loop counter with 1, the output would be –

```
Total number of arguments: 5
The arguments provided are:
hello
how
are
you?
```

7.3 DYNAMIC MEMORY ALLOCATION

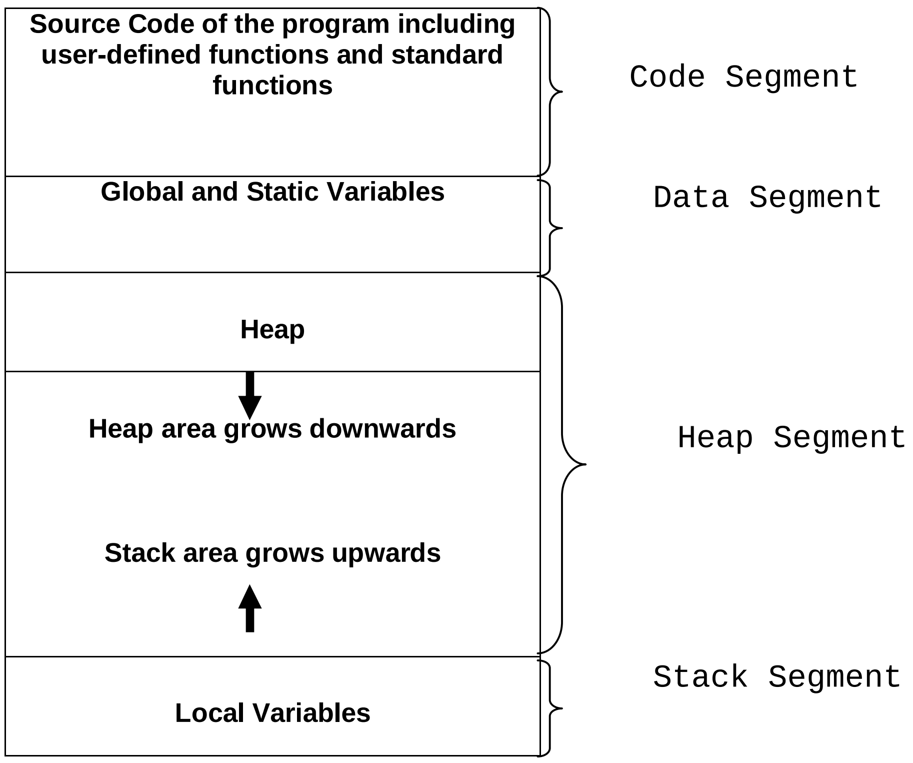
To understand the working of dynamic memory allocation, one should understand few concepts about memory segments of the computer. Also, one needs to understand static memory allocation and generic pointer concept. These are explained in the following sections.

7.3.1 Basics about memory

When a C program is compiled, the compiler translates the source code into machine code. Now the program is given a certain amount of memory to use. This memory is divided into four segments viz.

- **Code Segment**, in which the source code is stored
- **Data Segment**, that holds static and global variables
- **Heap Segment**, a free area
- **Stack Segment**, where local variables are stored

The memory organization for a C/C++ program is depicted here under–


Source Code of the program including user-defined functions and standard functions
Code Segment
Global and Static Variables
Data Segment
Heap
Heap area grows downwards
Heap Segment
Stack area grows upwards
Local Variables
Stack Segment

Consider a block of code:

```
void test()
{
        int Var_X;
        --------
        --------
}
```

When this function is compiled and linked, an executable file will be generated. The executable file contains equivalent instructions for all the statements of the function. When this executable file is executed, memory is allocated for the variable Var_X. This is called as ***Static memory allocation*** and memory for Var_X is given from ***stack segment.***

When the compiler writes instructions to allocate memory for Var_X in executable file, it also writes the instructions to de-allocate that memory at the end of the block within which Var_X is declared. So, during execution, the memory for Var_X is de-allocated when Program Control (PC) comes out the block in which it is declared. Thus, any local variable can not be accessed outside its block. This is known as ***static memory de-allocation.*** Note that, this freed memory is returned to the Operating System (OS) by the compiler.

Static memory allocation has its own pitfalls. To understand the problems, consider the following sequence of code:

```
void test()
{
        int arr[20];
        ………
        ………
}
```

Now, 40 bytes (assuming, integer requires 2 bytes) will be allocated for the array arr, and it can store 20 integers. If the number of items to be stored exceeds 20, then the array arr will not be capable to do so. On the other hand, assume that the number of items is less than array size, say, 5. Then 10 bytes are sufficient and remaining 30 bytes of memory will be wasted. This kind of problems viz. memory overflow or wastage of memory are inevitable with static memory allocation.

To overcome these problems, one should go for ***Dynamic memory allocation***. In this method, the memory for variables will be allocated during runtime *based on the requirements arising at runtime*. And the memory blocks are taken from ***Heap segment***. When such memory locations are no longer required, the same can be de-allocated and returned to OS. This is known as ***dynamic memory de-allocation***. To do this kind of memory allocation and de-allocation, the programmer has to write specific code within the program.

The differences between static and dynamic memory allocations are listed hereunder –

Static Memory Allocation	Dynamic Memory Allocation
1. Used when the number of memory locations required is known in advance.	1. Used when number of memory locations required is unknown.
2. The size of the memory to be allocated is fixed and can not be varied during run time.	2. The size of memory can be increased or decreased during run time.
3. The variables with statically allocated memory are named ones and can be manipulated directly.	3. The variables with dynamically allocated memory are unnamed and can be manipulated indirectly only with the help of pointers.
4. The allocation and de-allocation of memory for variables is handled by the compiler automatically.	4. The memory allocation and de-allocation must be explicitly performed by the programmer.
5. As memory for the variables is decided at compile time and data manipulation is done on these locations, execution of the program is faster.	5. As memory must be allocated during run time, the execution will be slower.
6. Memory for global and static variables will be allocated in *Data Segment* and for local variables in *Stack Segment.*	*6.* Memory allocation will be in *Heap Segment.*

7.3.2 Generic Pointer/Void pointer

Dynamic memory allocation/deallocation in C is done through 4 functions viz. malloc(), calloc(), realloc() and free(). To understand the working of these functions, you should know the concept of generic pointer or void pointer.

As the name suggests, generic pointer is a pointer which can store the address of any type of variable. Normally, when we don't know, address of which type of variable we are going to assign to a pointer, we can declare it as a void pointer. Later, we can assign the address of a variable to the void pointer. While accessing the data via pointer, we need to use appropriate typecasting.

For example:

```
void main()
{
        void *p;        // a void pointer or generic pointer declared
        int   a= 5;
        float b= 3.4;

        p= &a;          //address of integer variable is assigned to p
        printf("%d", *(int *)p);        //dereference after typecasting
        p= &b;          //address of float variable is assigned to p
        printf("%f", *(float *)p);      //dereference after typecasting
}
```

One can note that, the same pointer *p* has been used to store the address of an integer as well as float. But, while dereferencing those values, first we should type-cast. Consider, the statement,

```
printf("%d", *(int *)p);
```

Here, *p* is type casted to the type ***pointer to integer type*** using **int *.** Later, it is dereference using ***.**

Thus, C/C++ language provides an option to programmer to make use of single generic pointer that is capable of storing address of any data type.

Note that, the memory address returned from heap area will always be of generic type. Hence, this topic is discussed here.

7.3.3 Dynamic Memory Allocation in C

The memory allocation and de-allocation can be done in C using the functions like malloc(), calloc(), realloc() and free. Each of these functions are discussed below. The function malloc() is discussed in more elaborative manner with examples. But, similar approach/concepts will apply to calloc() and realloc() functions also.

- **malloc(size):** This function allocates a block of 'size' bytes from the heap. It will return a void pointer if the requested memory is available in the heap area. Otherwise, it returns NUL. Syntax of this function is –

  ptr = (data_type *) malloc(size);

  Here,

'ptr'	is a pointer variable of type 'data_type',
'data_type'	is any basic C data type/user defined data type
'size'	is the number of bytes required.

  For ex.-

  ```
  int *ptr;
  ptr = (int *) malloc(sizeof(int) * n);
  ```

  Suppose n=3 and size of integer is 2 bytes, then above statement will allocate 3 x 2= 6 bytes and the address of first byte is assigned to ptr.

  To understand the concept clearly, let us discuss the memory map for the following set of statements:

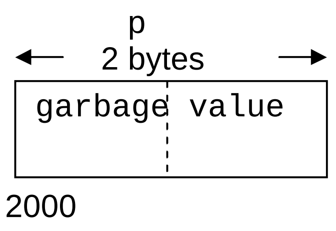
```
int *p;
p = (int *) malloc(sizeof(int));
*p= 10;
```

Now, the memory map for each of these statements may look like –

Statement:

```
int *p;
```

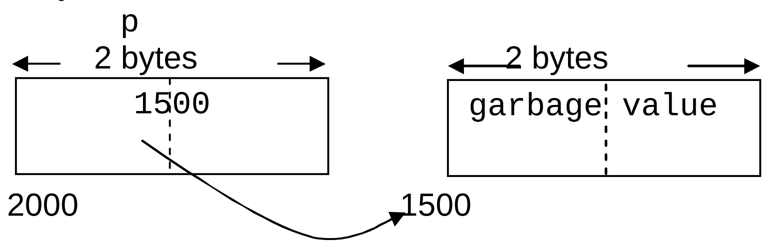
Memory Map:



Statement:

```
p = (int *) malloc(sizeof(int));
```

Memory Map:



Statement:

```
*p=10;
```

Memory Map:

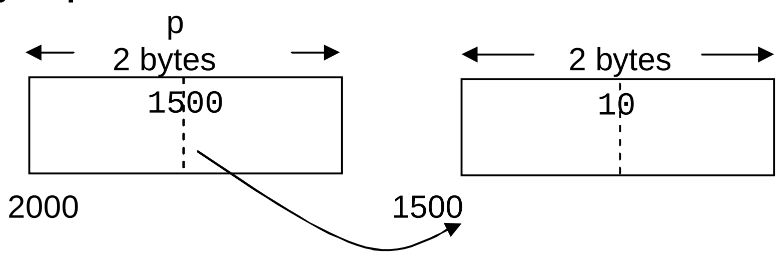


Figure 7.1 DMA memory map

It is important to note from the above figure that, the address of p viz. 2000 is allocated from the **stack segment** and the address stored at p viz. 1500 is allocated from the **heap segment.**

Note also that, if 2 bytes of memory is available in heap, then the statement,

p = (int *) malloc(sizeof(int));

assigns the base address of those memory blocks to the pointer p. Otherwise, NULL is returned. So, it is always a good practice to check whether memory is allocated or not, before using the pointer. An illustrative program for dynamic memory allocation is given below:

```
#include<stdio.h>
#include<alloc.h>
#include<stdio.h>
void main()
{
        int *p;
        p = (int *) malloc(sizeof(int));

        if (p == NULL)
        {
                printf(”Allocation Failed”);
                exit(0);
        }
        *p=10;
        printf(”value is  %d“, *p);
}
```

OUTPUT 1:

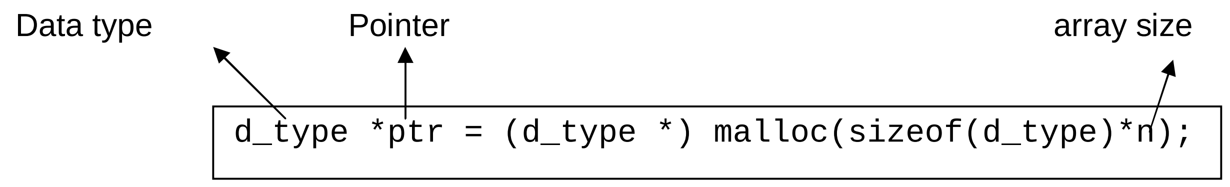
```
        Value is 10     (if heap has sufficient free space)
```

OUTPUT 2:

```
        Allocation Failed (in case of insufficient memory)
```

DMA for Arrays using malloc():

In the previous section, it has been discussed about allocating memory dynamically for a single variable. It is possible to allocate dynamic memory to arrays also. The syntax is as given below –

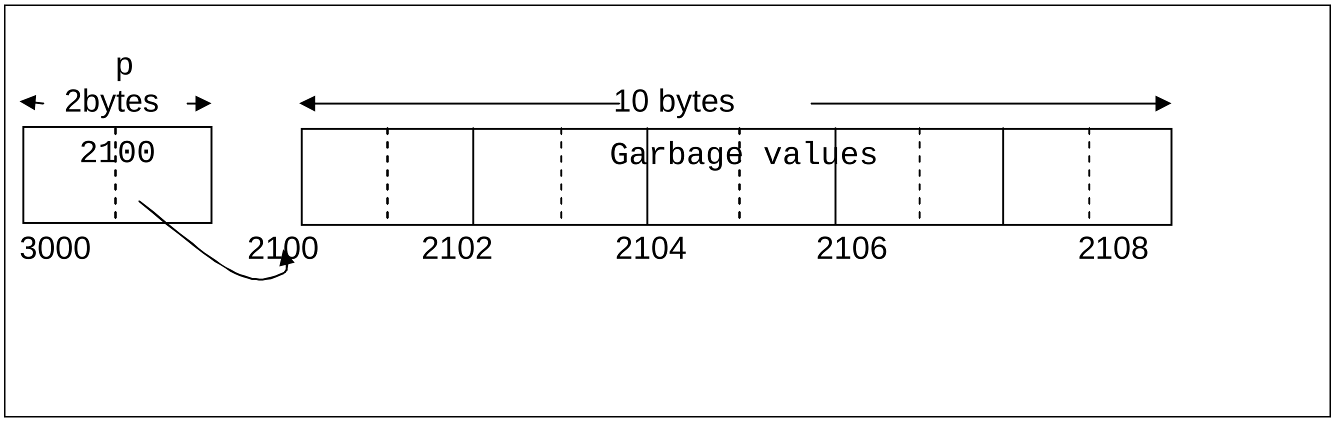


```
d_type *ptr = (d_type *) malloc(sizeof(d_type)*n);
```

For example,

```
int *p = (int *) malloc(sizeof(int) *5);
```

Now, p will hold the base address of an array of 5 integers. The memory map may look like –



The following program illustrates DMA for arrays.

```
#include<stdio.h>
#include<alloc.h>
#include<stdlib.h>
void main()
{
  int *p, i,n;
  printf(”Enter size of the array:”);
  scanf(“%d”, &n);
  p= (int *) malloc(sizeof(int)*n);

  if (p == NULL)
  {
                  printf(”Allocation Failed”);
                  exit(0);
  }
  printf(”\nEnter array elements:”);
  for(i=0;i<n;i++)
          scanf(’%d”,  (p+i));

  printf(”\nElements are:”);
  for(i=0;i<n;i++)
          printf(“%d”, *(p+i));
}
```

The output would be –

```
Enter size of the array: 5
Enter array elements: 12    31     -4      64      10
Elements are: 12      31     -4      64      10
```

- **calloc(n, size):** This function is also used to allocate the memory from heap. But, malloc() allocates a single block of contiguous storage where as calloc() allocates multiple blocks of storage, each block of the same size. More over, calloc() initializes the memory block with the value zero, which is not the case with malloc(). Syntax is give as below –

  ptr= (data_type *) calloc(n,size);

  Here, 'ptr' is a pointer of type 'data_type', 'size' is the number of bytes required and 'n' is the number of blocks to be allocated of 'size' bytes.

  Rest of the explanation for calloc() function will be same as that for malloc().

- **realloc(ptr, size):** In some situations, we need to increase of decrease the memory blocks already allocated using malloc() or calloc(). In this case, to resize the memory, we will go for re-allocation of memory using this function. The syntax is as below –

  ptr = (data_type *) realloc(ptr, size);

  Here, 'ptr' is the starting address of allocated memory obtained before by using malloc(), calloc() or realloc(). And 'size' is the number of bytes required for re-allocation.

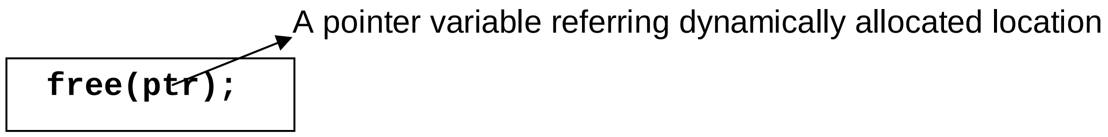
7.3.4 Dynamic Memory De-allocation in C

It has already been discussed that the dynamically allocated memory must be de-allocated and to be returned to OS. Let us see, what happens if such memory is not de-allocated, with the help of following example. Consider a block of code –

```
void test()
{
        int *ptr;
        ptr=(int *) malloc(sizeof(int));
        *ptr=10;
        ………
        ………
}
```

The memory map for ptr may be as shown in the **Figure 7.1.** As it has been discussed earlier, ptr is a local variable and the memory location 2000 is allocated from stack segment. The location with base address 1500 is allocated from heap. It is locked, and can be referred only with the help of ptr. When the program control goes out of the function test(), these two bytes of memory (2000 and 2001) will be de-allocated and returned to OS automatically. But, the location 1500, which is taken from heap, remains as an un-referenced location. Since ptr is destroyed, there is no means for referring the location 1500. As these locations (1500 and 1501) are locked, the OS can not allocate them to any other variable later, though they are no longer in use. This is known as ***memory leak.***

To overcome this problem, one has to de-allocate the memory manually in the program. This is achieved with the help of ***free()*** function. The syntax is –

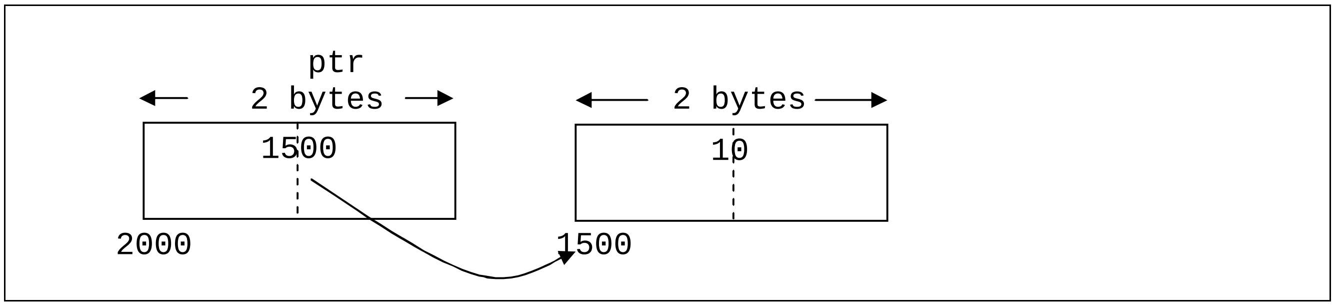


```
free(ptr);
```

For example, consider a code segment –

```
void test()
{
        int *ptr= (int *) malloc(sizeof(int)
        *ptr=10;
        printf(”Value is %d “,  *ptr);
        free(ptr);
}
```

Here, the memory allocation is as shown below.



When the last statement

```
free(ptr);
```

is executed, the location with base address 1500 is released from the hands of ptr and is returned to OS. So, this location can then be allocated to some other variable by OS, if needed, thus preventing memory leak. Then ptr (location 2000) is de-allocated and returned to OS when the program control comes out of the function test().

UNIT 8. PREPROCESSORS

The preprocessor is a program that processes the source code before it is passed to the compiler. It will offer several features called preprocessor directives, each of which will begin with a # symbol. It may be any one of

- Macro expansion
- File inclusion
- Conditional compilation
- Miscellaneous directives

8.1 MACRO EXPANSION

It starts with the statement **#define**. It may be defined either with arguments or without arguments. Consider the following program to illustrate a macro without arguments.

An example for a macro without arguments

```
#include<stdio.h>
#define PI  3.1416

void main()
{
        float rad, area;

        printf(”Enter the radius:”);
        scanf(“%f”, &rad);

        area=PI * rad * rad;
        printf(”\nArea =%f “, area);
}
```

The output would be:

```
        Enter the radius: 5
        Area = 78.540001
```

Here, the statement

```
                #define PI  3.1416
```

is called as ***macro definition*** or simply *macro.* The term PI is known as ***macro template*** and the value 3.1416 is ***macro expansion*** of PI.

When the command for compiling the program is given, before the source code passes to the compiler, it is processed by the C preprocessor for macro definition. When it encounters the #define statement, it goes through the entire program to search for macro template. Every occurrence of a macro template is replaced by the corresponding macro expansion now. After this procedure, the program is passed to the compiler.

It is customary to have a constant value used in the program as a macro definition. Because, whenever the programmer wants to change the value of a constant, it is sufficient to change only at #define statement.

Macros can have arguments just like functions. The program given below is an illustration for the macros with arguments.

Illustration for a macro with arguments

```
#include<stdio.h>
#define CUBE(x) (x*x*x)

void main()
{
        int a, b;

        printf(”Enter two numbers:”);
        scanf(“%d%d”, &a, &b);

        printf(“\nThe cube of %d is %d”, a, CUBE(a));
        printf(“\nThe cube of %d is %d”, b, CUBE(b));
}
```

The output would be:

```
        Enter two numbers:
                4       6

        The cube of 4 is 64
        The cube of 6 is 216
```

When the preprocessor finds the macro CUBE with a parameter, it replaces the macro template with the macro expansion. After such replacement for all the occurrences of CUBE, the program is transferred to the compiler. So, for the compiler, the above program looks like

```
void main()
{
        printf(”Enter two numbers:”);
        scanf(“%d%d”, &a, &b);

        printf(“\nThe cube of %d is %d”, a, (a*a*a));
        printf(“\nThe cube of %d is %d”, b, (b*b*b));
}
```

Though the macro with arguments seems to be similar to a function call, they are not same. When a function is called, the program control is passed to the called-function with necessary arguments. Then, operations are performed inside the function and a value is returned back to the calling-function. But, in case of macro, the preprocessor replaces the macro template with macro expansion. Thus, if a particular function is called several times in a program, the body of the function exists only once. But, there will be separate copies of a macro for each occurrence of macro template. This indicates that, the use of macro increases the object-code/size of the program. Therefore, it is always a good programming practice to use macros only for small tasks.

Consider one more example to illustrate how macro functions are different from ordinary functions.

```
#include<stdio.h>
#include<conio.h>
#define TEST(x, y)  x+y-2

void main()
{
        int a=5, b=10, c;

        c=TEST(a, b)*3;
        printf("%d", c);
}
```

Output of the above program will be 9, because in the statement

c= **TEST(a, b)***3;

the macro would have been replaced by preprocessor as –

c= **a+b-2** * 3;

Thus, it will first multiply 2 and 3, then the result is subtracted from the sum of a and b. Hence, the result will be 9.

But, what we expect out of above macro function (if it was an ordinary function) is –

a and b must be added and 2 is subtracted, then the result must be multiplied by 3. That is, we expect the result to be 39. The unexpected behavior is happening because, the macro function expansion is not having appropriate brackets.

Thus, the statement

#define TEST(x, y) **x+y-2**

must be written as –

#define TEST(x, y) **(x+y-2)**

This proves that macro expansion/function is just a replacement by the preprocessor, but no logical calculation is involved in it. Hence, the programmer must be careful while defining a macro function.

8.2 FILE INCLUSION

In procedural languages, a big problem is divided into several related sub-problems. Usually, the programs for such sub-problems are written in different files. Then to get the solution for the entire problem, these files must be segregated. Moreover, some of the programs may be having few functions in common. Such functions can be written in a single file and then this file can be made use of whenever required. Both of these situations can be best handled with the help of file inclusion preprocessor directive.

The file inclusion directive starts with #include statement. It can take any of the two forms:

#include<filename.h>	The file is assumed to be a project or a standard header file. And this command would look for the file filename.h in the specified list of directories only.
#include "filename.h"	This command would look for the file filename.h in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.

8.3 CONDITIONAL COMPILATION

Sometimes, the included file may itself contain a #include directive. Because of nested include files, a header file may sometimes be included several times for a single source file. Such multiple processing of a header file can be prevented using conditional directives. For example,

```
#ifndef MYFILE.H
        #include MYFILE.H
#endif
```

Here, #ifndef checks whether MYFILE.H is included or not. If it is included already, the next line is not going to be processed. Otherwise, MYFILE.H is getting included using #define directive.

Just like ordinary if-else control statement, the conditional directives #ifdef and #else can be used. This is generally used for conditional inclusion of some statements in the program depending on whether a particular preprocessor constant is defined or not. For example,

```
#define MAX 5

main()
{
        #ifdef MAX
                printf( “Constant MAX is defined”);
        #else
                printf("Constant MAX is not defined");
        #endif
        …….
        …….
}
```

In the above program, as we have defined the constant MAX using #define statement before the main() function, the output statement ‘Constant MAX is defined’ gets printed. If we remove the #define statement for MAX, then the output will be ‘Constant MAX is not defined’.

8.4 MISCELLANEOUS DIRECTIVES

Following are the two rarely used preprocessor directives.

#undef	It is used to ‘un-define’ a defined name. That is, if we do not require an already defined macro, it can be undefined using #undef statement. For example, a program has a macro like
	`#define PI 3.1416`
	After sometimes, if this macro is not needed by the program, it can be suppressed using the statement
	`#undef PI`

#pragma	This is a special kind of directive that can be used to turn on or off certain features. The working of pragmas varies from compiler to compiler. In Turbo C compiler, a pragma can be used to write assembly language code within a C program. The detailed study of the pragmas is beyond the scope of the study.