Tokens in C

Character Sets

- C character sets consists of upper and lowercase alphabets, digits, special character as and white spaces
- The alphabets and digits altogether are called as the alphanumeric character

1. Alphabets : a-z, A-Z
2. Digits: 0-9
3. Special Characters: period, comma, semicolon, colon, number sign(#), apostrophe, double quotes, exclamation, tilde(~), underscore, dollar, asterisk, ampersand, caret, slash(/), backslash(\) and many more
4. White space characters: Space, newline \n, horizontal tab \t, carriage return \r and form feed \f

Keywords

- C keeps a small set of words(32 words) for its own use, these words are keywords
- predefined, reserved words used in programming that have special meanings to the compiler
- keywords are a part of the syntax

Keyword rules

a. keywords are always in lowercase letters
b. keywords cannot be changed by a programmer
c. keywords cannot be used as variables, function name, array name by a programmer

Keywords in uppercase letters can be used as identifiers but is considered as poor programming practice

example. int, float, short, long, double, struct, unsigned, sizeof, while, for, default, if, else, return, continue, break, void, goto, etc.

Identifiers

- names given to entities such as variables, functions, structures, etc.
- identifiers must be unique
- created to identify entities during execution of the program

Rules for naming identifiers

a. identifiers can have alphabets, digits and underscore, but $,#,... etc cannot be used
b. first letter must be an alphabet or underscore
c. you cannot use keywords as identifiers
d. there is no rule on how long an identifier can be but some compilers may run into problems with identifiers longer than 31 characters
e. blank space should not be used in an identifier, eg. 'student name' is not valid, 'student_name' should be used

C Data types

- used to define the kind of data being stored or manipulated
- specifies the values that a variables or constants can hold and how the information is stored in the memory
- C supports three classes of data types:
  a. Primary/Fundamental data types
  b. Derived data types
  c. User defined data types

Qualifiers

- modifies the behavior of the variable type to which they are applied
- there are four types of qualifiers:
  a. size qualifiers:
     - alters the size of basic data types
     - the keywords **long** and **short** are two size qualifiers

  b. sign qualifiers:
     - specify whether a variable can hold both positive and negative numbers or only positive numbers
     - the keywords **signed** and **unsigned** are two sign qualifiers

  c. const:
     - an object declared to be const can't be modified(assigned to, incremented or decremented) by a program
     - const variable can be declared as *const int a = 100;* *or* *int const a = 100;*

  d. volatile
     - a variable should be declared volatile whenever its value can be changed by some external sources from outside the program
     - volatile variable can be declared as *volatile int a = 100;* or *int volatile a = 100;*

a. Primary/Fundamental data types

- the data types which are predefined in the language and can be directly used to declare variable in C
- primary data type can be categorized as follows:

1. Integer type
    a. signed integer type
        i. int
        ii. short int
        iii. long int
    b. unsigned integer type
        i. unsigned int
        ii. unsigned short int
        iii. unsigned long int
2. Character type
    a. signed char
    b. unsigned char
3. Floating point type
    a. float
    b. double
    c. long double

- **int - integer:** A real number without a fraction. Ex 1,2,3 but not 1.5, 2.4, 3.14
    - to declare an int, you can use instruction:

      int variable _name;

      int a; declares an int(integer) variable called a
- **float - floating point:** A number with a fractional part. Ex 1.0, 1.5, 3.14, etc
    - to declare a float, you use the instruction:

      float variable_name;

      float a; declares a float variable called a
- **double:** A double-precision floating point value
    - to declare an double, you can use instruction:

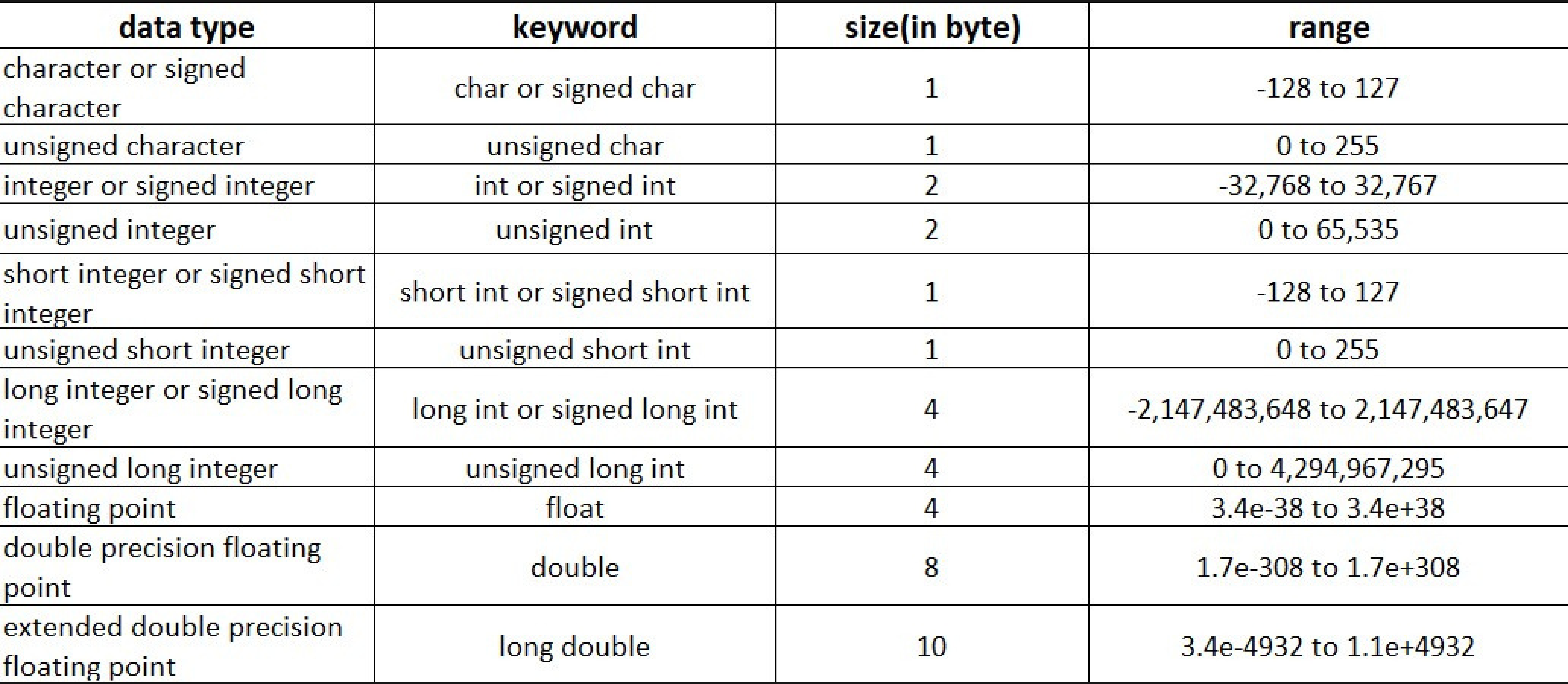
      double variable_name;

      double a; declares a double float variable called a
- **char:** A single character. Ex. all characters defined in character sets
    - to declare a character, you can use instruction:

      char variable_name;

      char a; declares a character variable called a
- **void:** Along with all the above types, there is another type. void type means a variable has no value. This is usually used in functions which return nothing.

Size and range of data types



data type	keyword	size(in byte)	range
character or signed character	char or signed char	1	-128 to 127
unsigned character	unsigned char	1	0 to 255
integer or signed integer	int or signed int	2	-32,768 to 32,767
unsigned integer	unsigned int	2	0 to 65,535
short integer or signed short integer	short int or signed short int	1	-128 to 127
unsigned short integer	unsigned short int	1	0 to 255
long integer or signed long integer	long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long integer	unsigned long int	4	0 to 4,294,967,295
floating point	float	4	3.4e-38 to 3.4e+38
double precision floating point	double	8	1.7e-308 to 1.7e+308
extended double precision floating point	long double	10	3.4e-4932 to 1.1e+4932

b. Derived data types

i) Functions
ii) Arrays
iii) Pointers

c. User defined data types

- C provides a feature to define new data types according to user's requirement
- The ways of defining user defined data types are:
  a. Structure
  b. Union
  c. Enumeration

Preprocessor directives

- Preprocessor directives are always started with Hash symbol(#)
- These preprocessors directives are used to add Header files to the program
- The main thing is preprocessor directives are used to include files to our programs
- Commonly used Preprocessor Directives are #include and #define
- #include is used for including files may be header files or normal files
- #define is used to define symbolic constants and macros
- All types of preprocessor directives are as follows:

1. #define
2. #include
3. #ifdef
4. #under
5. #ifunder
6. #if
7. #else
8. #elif
9. #endif
10. #error
11. #pragma

1. #define (Macros)

- A macro is a fragment of code which has been given a name.
- Whenever the name is used, it is replaced by the contents of the macro

  Syntax
  #define token value

- There are two types of macros:
  - Function - like macros
    - Example. #define   MAX(a,b)   ((a) > (b) ? (a) : (b))
  - Object - like macros
    - Example. #define PI 3.1415
  - #define vs typedef

    Example.   #define MY_TYPE int
               typedef int My_Type

2. #include

- It has two variants which replace the code with the current source files code
- two variants are as follows:

  i) #include<file> : searches for a file in the defined list of the system or directories as specified

  ii) #include"file" : used for your own customized header files of the program

Constants

- constant is an identifier that is always associated with the same data value
- when you declare a constant, the value cannot be changed
- constants are also called literals
- constants can be any of the data types
- it is considered best practice to define constants using only UPPERCASE names

  Syntax: *const type constant_name*

  Eg. const int SIDE = 4;

- Constants are categorized into two basic types, and each of these types has its subcategories. They are:
  a. Numeric Constants
     - Integer Constants
     - Real Constants

  b. Character Constants
     - Single Character Constants
     - String Constants

Integer Constants

- It's refering to a sequence of digits. Integers are of three types:
    a. Decimal Integer
    b. Octal Integer
    c. Hexadecimal Integer

  Examples:15,-265,0,99818,0X6

Real Constants

- The numbers containing fractional parts like 99.25 are called real or floating points constant.

Single Character Constant

- It simply contains a single character enclosed within a pair of single quote
- It is to be noted that the character '8' is not the same as 8
- Character constants have a specific set of integer values known as ASCII values(American Standard Code for Information Interchange)
  Examples:'X','5',';'

String Constants

- These are a sequence of characters enclosed in double quotes and they may include letters, digits, special characters, and blank spaces.
- It is again to be noted that "G" and 'G' are different because "G" represents a string as it enclosed within a pair of double quotes whereas 'G' represents a single character.
  Examples:"Hello","2015","2+1"

Variables

- symbolic names which is used to store different types of data in the computer's memory
- when the user gives the value to the variable, that value is stored at the same location occupied by the variable
- variables may be of integer, character, string or floating type

- Before you can use a variable you have to declare it. To declare a variable, user should state its type and give its name
- To declare a variable in C, do

```
            <data_type> <list_variables>;
    Examples
        int a;
        float a,b,c;
```

Operators

- An operator is defined as a symbol that specifies the mathematical, logical or relational operation to be performed.
- The data items that operators act upon are called operands.

  expression $\longrightarrow$ x = a + b / c - 2

- An expression is a combination of variables, constants and operators written according to the syntax of C language.
- In C, every expression evaluate to a value i.e. every expression result in some value of a certain type that can be assigned to a variable.

  Algebraic expression: 3x^2 + 2x +1

  C expression: 3*x*x + 2*x +1

Types of operators

A. Based on number of operands they act upon, C operators are classified as follows:

1) Unary Operators
2) Binary Operators
3) Ternary Operators

1) Unary Operators:

- Unary operators are the operators in C which are used to perform certain operation on single operand.
- They are named unary because they work on only one operand.
- Some of the unary operators are ++, --, -, !, etc.

2) Binary operators:

- Binary operators are those operators that operate on two operands.
- Examples: +, -, *, /, <, >, etc.

3) Ternary Operators:

- Ternary Operator is the operator which acts upon three operands.
- Conditional operator(?:) is the only example of ternary operator.
- The conditional operators is similar to if-else statement but it takes less space and helps to write if-else statements in the shortest way possible.
- The conditional operator is of the form:

```
        variable = expression1 ? expression2 : expression3;
```

- It can be visualized into if-else statement as:

```
if(expression1)
 {
       variable = expression2;
 }
 else
  {
       variable = expression3;
   }
```

B. Based on utility and action, C operators are classified as follows:

1) Arithmetic Operators (+, -, *, /, %)
2) Relational Operators (>, <, <=, >=, ==, !=)
3) Logical Operators (&&, ||, !)
4) Assignment Operators (=)
5) Conditional Operators (?:)
6) Increment/ Decrement operators (++/--)

1) Arithmetic Operators

- Arithmetic Operators are those operators that perform different arithmetic operations such as addition, subtraction, multiplication, division etc
- Examples: +, - , *, /, %
- Modulo Division operation is performed by % operator (also known as remainder operator).

2) Relational Operators:

- Relational Operators are used to compare two similar operands, and depending on their relation, take some actions.
- They compare their left hand side operand with their right hand side operand for different relations.
- The value of relational expression is either 1(if the condition is true) or 0(if the condition is false).
- Different relational operators in C are >, <, <=, >=, == (read equals to) and != (read not equals to)

  For example: if a = 5 and b = 6, then the expression (a>b) results 0(false).

3) Logical Operators:

- Logical operators are used to compare or evaluate logical and relational expressions.
- The final result produced by this operator is also either 1(True) or 0(False).
- The three logical operators in C are:

  a) &&   logical AND

b) ||	logical OR
c) !	logical NOT

For example: If a=5, b=6, c=3, then the expression (a<b)&&(a>c) results 1(true).

4) Assignment Operator (=) :

- It is a binary Operator that assigns the result of an expression in right hand side to a variable in left hand side.
- syntax: variable = expression;
- If a=5, b=6 the c = a+b means the value of a+b (i.e. 11) is assogned to c. Also 5 is assigned to a and 6 is assigned to b.

5) Conditional Operator (?:) :

- It is a ternary operator thet is used to write conditional structures in a single expression.

6) Increment and Decrement Operator:

- Increment operators are used to increase the value of the variable by 1 and and decrement variable are used to decrease the value of the variable by 1

- Both increment and decremenrt operator are used on a single operand or variable.
- There are two types of increment operators:
  a) pre-increment
  b) post-increment

a) Pre-increment:

- In pre-increment, first increment the value of variable and then used inside the expression.
- syntax: ++variable;
- Example:

```
#include<stdio.h>
#incluse<conio.h>
void main()
{
    int x, i;
    i = 10;
    x = ++i;
    printf("x: %d", x);
    printf("i: %d", i);
    getch();
}
```

output:

```
x: 11
i: 11
```

In above program first increase the value of i and then assign value of i into expression.

b) Post-increment:

- In post-increment, first the value of variable is used in the expression and then increment the value of the variable.
- syntax: variable++;
- Example:

```
#include<stdio.h>
#incluse<conio.h>
 void main()
 {
       int x, i;
       i = 10;
       x = i++;
       printf("x: %d", x);
       printf("i: %d", i);
       getch();
      }
```

output:

```
x: 10
i: 11
```

In above program first assign the value of i into expression then increase value of i by 1. Similarly it can be done same for the decrement operator.

Precedence and Associativity of operators

Precedence of operators:

- If more than one operators are involved in an expression then, C language has predefined rule of priority of operators. This rule of priority of operators is called operator precedence.
- In C, precedence of arithmetic operators (*, %, /, +, -) is higher than relational operators (==, !=, >, <, <=, >=) and precedence of relational operators is higher than logical operators (&&, || and !).

Associativity of Operators:

- Associativity indicates in which order two operators of same priority (precedence) executes.
- let's consider an expression,

  a == b != c

- Here, operators == and != have same precedence.
- The associativity of both == and != is left to right, i.e. the expression in left is executed first and execution take place towards right.

Table 20.24 Summary of C operators with their precedence and associatvity

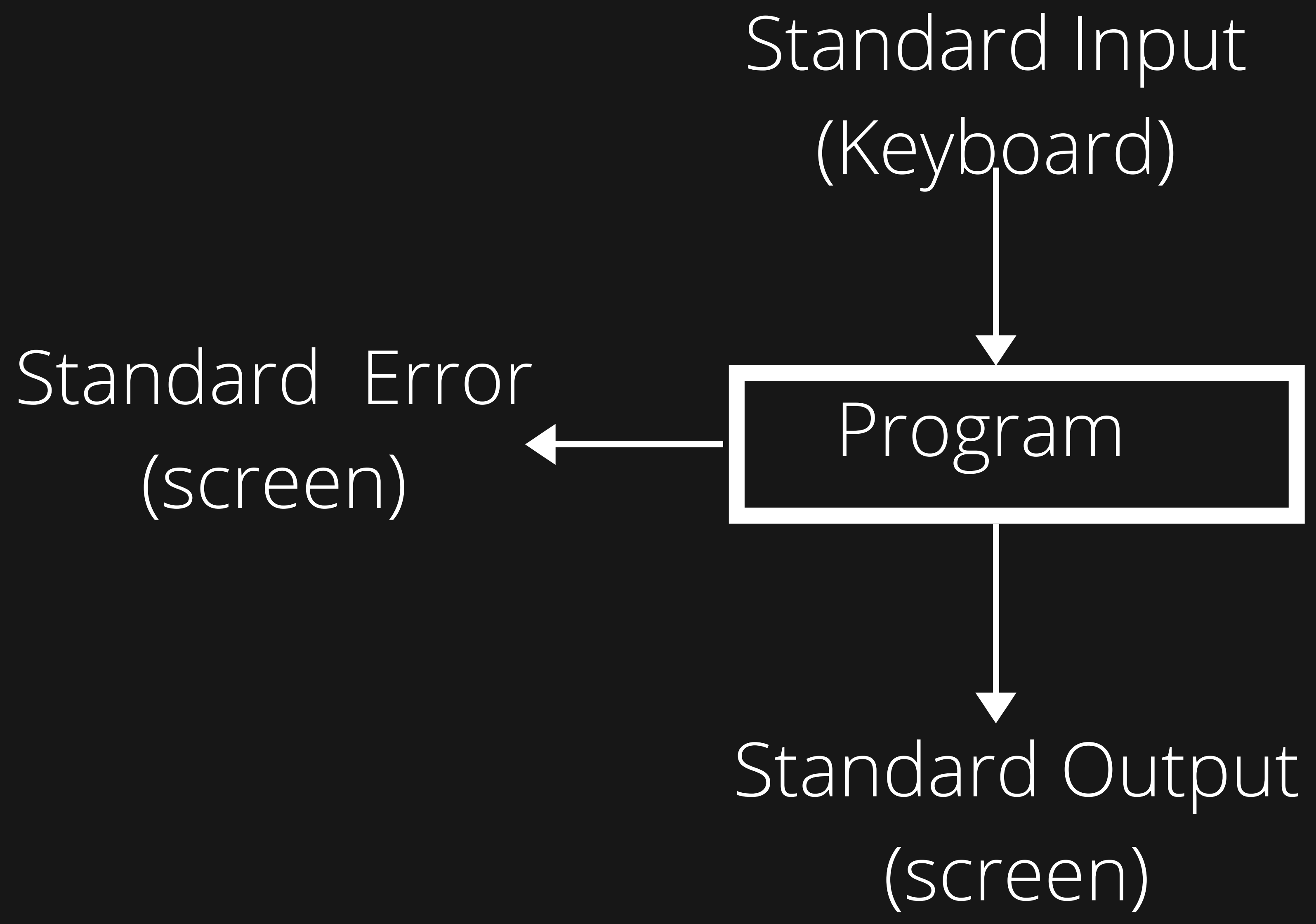
Operator	Operation	Precedence	Associativity
()	Functional call	1	Left to right
[]	Array element reference		
->	Indirect member selection		
.	Direct member selection		
!	Logical negation	2	Right to left
~	Bitwise(1's) complement		
+	Unary plus		
-	Unary minus		
++	Pre increment or post increment		
--	Pre decrement or post decrement		
&	Address		
*	Pointer reference(indirection)		
sizeof	Returns the size of an object in bytes		
(type)	Type cast (conversion)		
*	Multiply	3	Left to right
/	Divide		
%	Remainder(modulus)		
+	Binary plus(addition)	4	Left to right
-	Binary minus(subtraction)		
<<	Left shift	5	Lcft to right
>>	Right shift		
<	Less than	6	Left to right
<=	Less than or equal		
>	Greater than		
>=	Greater than or equal		

	Greater than or equal		
==	Equal to	7	Left to right
!=	Not equal to		
&	Bitwise AND	8	Left to right
^	Bitwise exclusive XOR	9	Left to right
|	Bitwise OR	10	Left to right
&&	Logical AND	11	Left to right
||	Logical OR	12	Left to right
?:	a ? x : y means "if a then x, else y".	13	Left to right
=	Simple assignment	14	Right to left
*=	Assign product		
/=	Assign quotient		
%=	Assign remainder (modulus)		
+=	Assign sum		
-=	Assign difference		
&=	Assign bitwise AND		
^=	Assign bitwise XOR		
|=	Assign bitwise OR		
<<=	Assign left shift		
>>=	Assign right shift		
Comma	separator as in int a, b, c ;	15	Left to right

Input and Ouput

INTRODUCTION

- Input means to provide the program with some data to be used in the program and output means to display data on screen or write the data to a printer or a file.
- C Programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.
- All these built-in functions are present in header files.



Types of Input/output

- There are two types of input/output statements:
  a. Formatted input/output.
  b. Unformatted input/output.

a. Formatted input/output

- **printf() -** formatted output function used to display messages and values on a screen.
  - **Syntax -** printf(" control string", arg1,arg2,.......);
    where, control string may contain only messages to display or with format specifier of individual output data items arg1, arg2,..... to be displayed.

Example: void main()

```
{
    int a=5;
    printf("\n Value of a = %d", a);
}
```

- **scanf() -** formatted input function used to read values from keyboard i.e. to give input.
  - **syntax:** scanf("control string", &arg1, &arg2,.......);

    where, control string specifies the type and format of the data that has to be obtained from keyboard and stored in the memory locations &arg1, &arg2,......

    Example:

    ```
    void main()
    {
        int a;
        scanf("%d", &a);
        printf("Entered value is %d",a);
    }
    ```

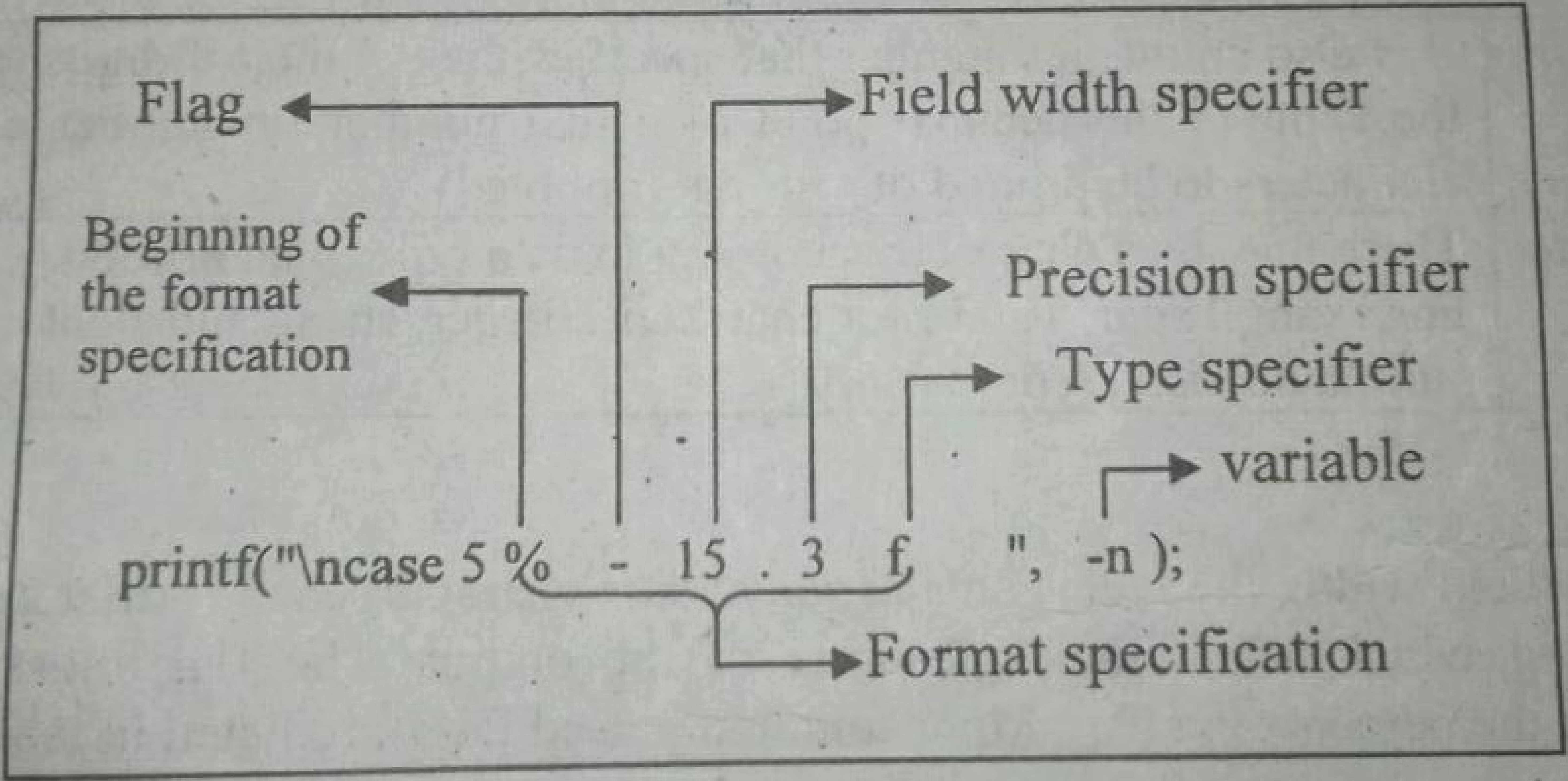


Fig. 4.2 Illustration of parts of format specification

for data output are listed in table 4.3

Table 4.3 Format specifications for data output (for printf function)

format specifier	Type of argument	Output
%c	character	single character
%s	string	prints characters until a null character is encountered
%d, %i	integer	signed decimal integer
%u	unsigned integer	unsigned decimal integer
%f	floating point	floating point value without an exponent
%e	floating point	floating point value with an exponent
%g	floating point	floating-point value either in e or f forms based on given value and precision.
%E	floating point	same as e; with e for exponent
%G	floating point	same as g; with e for exponent if e format is used.
%ld, %li	long integer	decimal signed long integer
%lu	unsigned long integer	decimal unsigned long integer
%hd, %hi	short integer	decimal signed short integer
%hu	unsigned short integer	decimal unsigned short integer
%le,%lf, %lg	double	double precision floating point
%Le,%Lf, %Lg	long double	floating point with precision more than in double.
%o	integer	octal integer.
%x	integer	hexadecimal integer (with a,b,c,d,e,f)

Table 4.4 Format specifications for data input (for scanf())

Basic format specification	Data type	Type of argument
%c	character	Address of a character variable
%s	string	Name of the string variable (why ? see chapter 7 string)
%d,	integer	Address of an integer variable
%i	decimal, octal or hexadecimal integer	Address of an integer variable
%u	unsigned integer	Address of an unsigned integer variable
%f	floating point	Address of a floating point variable
%e	floating point	Address of a floating point variable
%g	floating point	Address of a floating point variable
%E	floating point	Address of a floating point variable
%G	floating point	Address of a floating point variable
%ld, %li	long integer	Address of a signed long integer variable
%lu	unsigned long integer	Address of a unsigned long integer variable
%hd, %hi	short integer	Address of a short integer variable
%hu	unsigned short	Address of a unsigned short integer variable

	integer	
%le, %lf, %lg	double	Address of a double precision floating point variable
%Le, %Lf, %Lg	long double	Address of a long double variable.
%o	octal integer	Address of an integer
%x	hexadecimal integer	Address of an integer
[...]	string	Name of string. [It is used to read string including white spaces.]

ILLUSTRATIONS:

1. **Formatted I/O for numeric integer data:**
   a. Formatted output

   General form: %<flags> wd

   where, w is the field width and flags represent some characters(-, 0, +-, etc.) that are used to change the appearance of output.

   Example:

```
void main()
{
    int a = 12345;
    printf("\n Case 1: %d", a);       // w not specified
    printf("\n Case 2: %15d", a);
    printf("\n Case 3: %-15d", a);
    printf("\n Case 4: %015d", a);
    printf("\n Case 5: %3d", a);      // w less than width of data to print
}
```

```
OUTPUT:
case 1 : 12345
case 2 : ..........12345
case 3 : 12345..........
case 4 : 000000000012345
case 5 : 12345
```

b. Formatted input

general form: %wd

where w is field width.

example:

```
void main()
{
    int a, b;
    scanf("%3d %4d'", &a, &b);
    printf("\n a = %d \n b = %d", a, b);
}
```

```
OUTPUT 1:
12  1234
a = 12
b = 1234

OUTPUT 2:
12345  6789
a = 123
b = 45
```

2. Formatted Real Numbers:

For printing real numbers we use printf function and general form format specifier is :

    %<flags>w.pf

where, w is the field width and P specifies the number of digits to the right of the decimal point of a real number to be printed.

EXAMPLE:

```
void main()
{
    float a = 76.972563076;
    printf("\n case 1: %f", a);
    printf("\n case 2: %10f", a);
    printf("\n case 3: % 10.2f", a);
    printf("\n case 4: % -10.2f", a);
    printf("\n case 5: % e", a);
}
```

Output:

```
case 1: 76.972563
case 2: .76.972563
case 3: .....76.97
case 4: 76.97.....
case 5: 7.697256e+01
```

For inputting real numbers we use scanf function as,

```
scanf("%f", &float_var);
```

e.g. `scanf("%f", &a);`

3. Formatted Character I/O:

For printing character we use printf function and general form format specifier is :

        %<flags>wc

where, w is the field width of w column with right justification. e.g:

```
void main()
{
char ch = 'C';
printf("\n case 1: %c", ch);
printf('\n case 2: %5c", ch);
printf("\n case3: % -5c", ch);
}
```

OUTPUT:

```
case 1: C
case 2: ....C
case 3: C....
```

For inputting character we use scanf function as,

scanf("%c", &char_var);

e.g. scanf("%c", &ch);

4. Formatted I/O string data:

The formatted input string data has the specifications as below:

%ws

The formatted output string data has the specification as below

%w.ps

where w specifies the field for display and p instructs that only first p characters of the string are to be displayed. The display is right-justified.

Example:

```
void main()
{
    char a[10] = "Kantipur";
    printf("\n case 1: %s", a);
    printf("\n case 2: %15.4s", a);
    printf("\n case 3: %-15.6s", a);
}
OUTPUT:
 case 1: Kantipur
case 2: ...........Kant
case 3: Kantip.........
```

b. Unformatted input/output

1. Character I/O:
    - Input Functions:

      getchar() - reads a single character from user

      syntax: character_variable = getchar();

      e.g. void main()

      ```
      {
          char ch;
      ```

```
    printf("\n Enter a character");
    ch = getchar();
    printf("\n Entered character is : %c", ch);
}
```

For reading characters we can also use **getch()** and **getche()** functions.

syntax: character_variable = getche();
character_variable = getch();

e.g. ch = getche();
ch = getch();

getche() - reads a character from console and echoes to the screen. It allows a character to be entered without having to press the 'Enter' key afterwards.

Syntax - ch = getche();

where, ch is any character type variable.

```
e.g. void main()
{
        char ch;
        printf("enter any character: ");
        ch = getche();
        printf("entered character is %c", ch);
}
```

```
output: enter any character: B
                entered character is B
```

getch() - reads a character from console but doesn't echo to the screen. It allows a character to be entered without having to press the Enterkey afterwards.

syntax - ch = getch();

where, ch is any character type variable.

```
e.g. void main()
{
        char ch;
        printf("enter any character: ");
        ch = getch();
        printf("entered character is %c", ch);
        }

output: enter any character: B
            entered character is B
           // Here, the entered character B is not echoed to the screen.
```

Difference between getchar(), getch() and getche():

getchar() - wait for a character followed by the Enter key.

getch() - waits for a character and doesn't echo to the screen.

getche() - waits for a character and echoes to screen.

- Output Functions:

  putchar()

  It is an unformatted output statement for printing a character.

  syntax - putchar(ch);

  where, ch is any character variable.

  For printing a character we can also use putch();

  syntax - putchar(character_variable)

  e.g.
```
void main()
{
    char ch;
    printf("\n Enter a character");
    ch = getchar();
    printf("\n Entered character is :");
    putchar(ch);
}
```

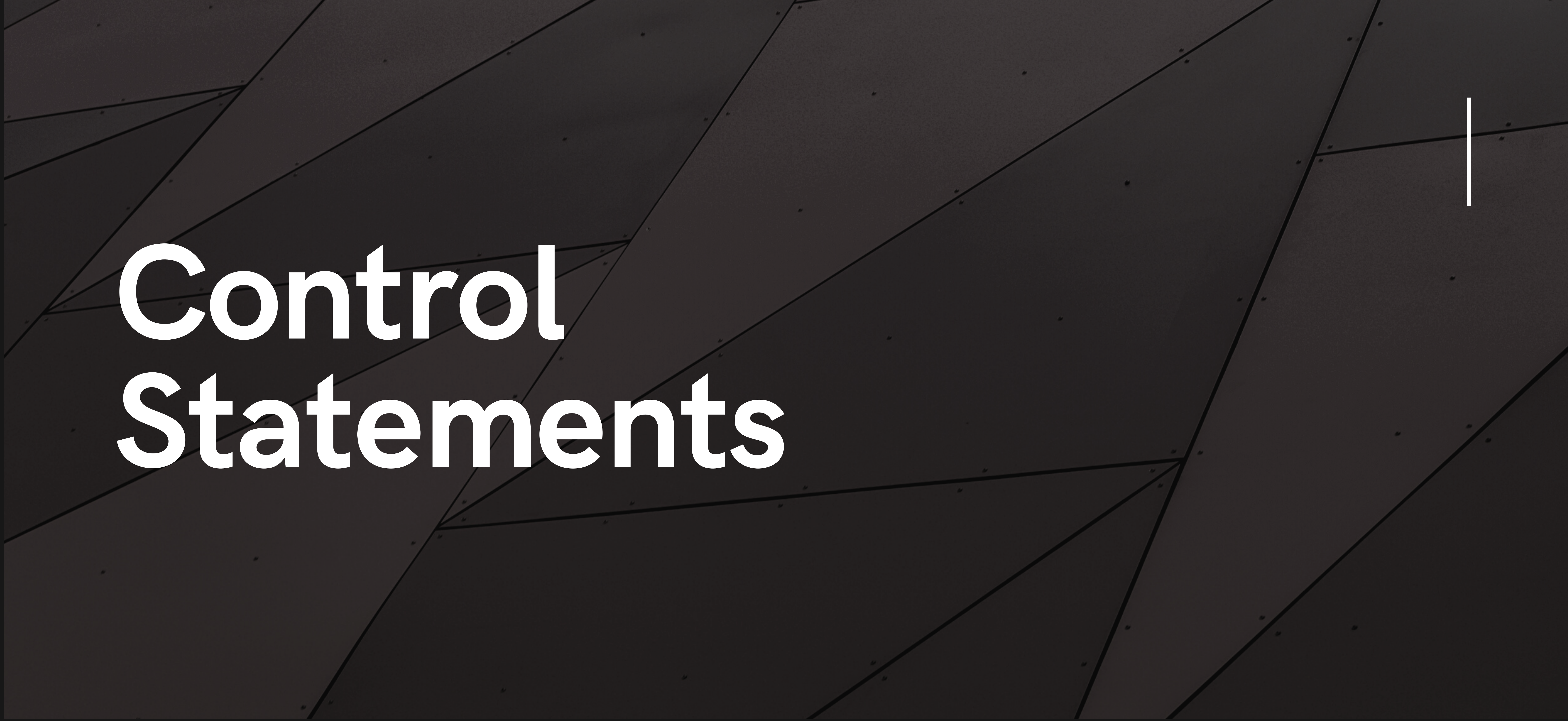
2. String I/O

i. input function

gets() - It is an unformatted input function for reading strings.**syntax() -** gets(str);
where 'str' is any string variable.

ii. Output function

puts() - It displays the string stored in string variable.
syntax - puts(string_variable)
e.g.
```
void main()
{
    char str[20];
    printf("\n Enter a string");
    gets(str);
    printf("\n Enteres string is: ");
    puts(str);
}
```



Control Statements

Prepared by

[redacted]in Devkota Udita Bista

Introduction

- The basic components of high level programming language such as C are its control structures.
- The structures which regulates the order in which program statements are executed are called control structures.
- The format of programming should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later.
- One method of achieving the objective of an accurate error resistant and maintainable code is to use one or any combination of the following three control structure available in programming:

    1. Sequential Structure(Straight-line flow)
    2. Selective Structure(Branching)
    3. Repetitive structure(Loop or iteration)

Sequential Structures

- Sequence simply means executing one instruction after another, in the order in which they occur in the source file.

Selective/Conditional Structures

- Selection means executing different sections of code de[pending on a condition or the value of a variable, this is what allows a program to take different courses of action depending on different circumstances

Repetitive Structures (Loops)

- Repetition means executing the same section of code more than once.
- A section of code may either be executed a fixed number of times, or while some condition is true

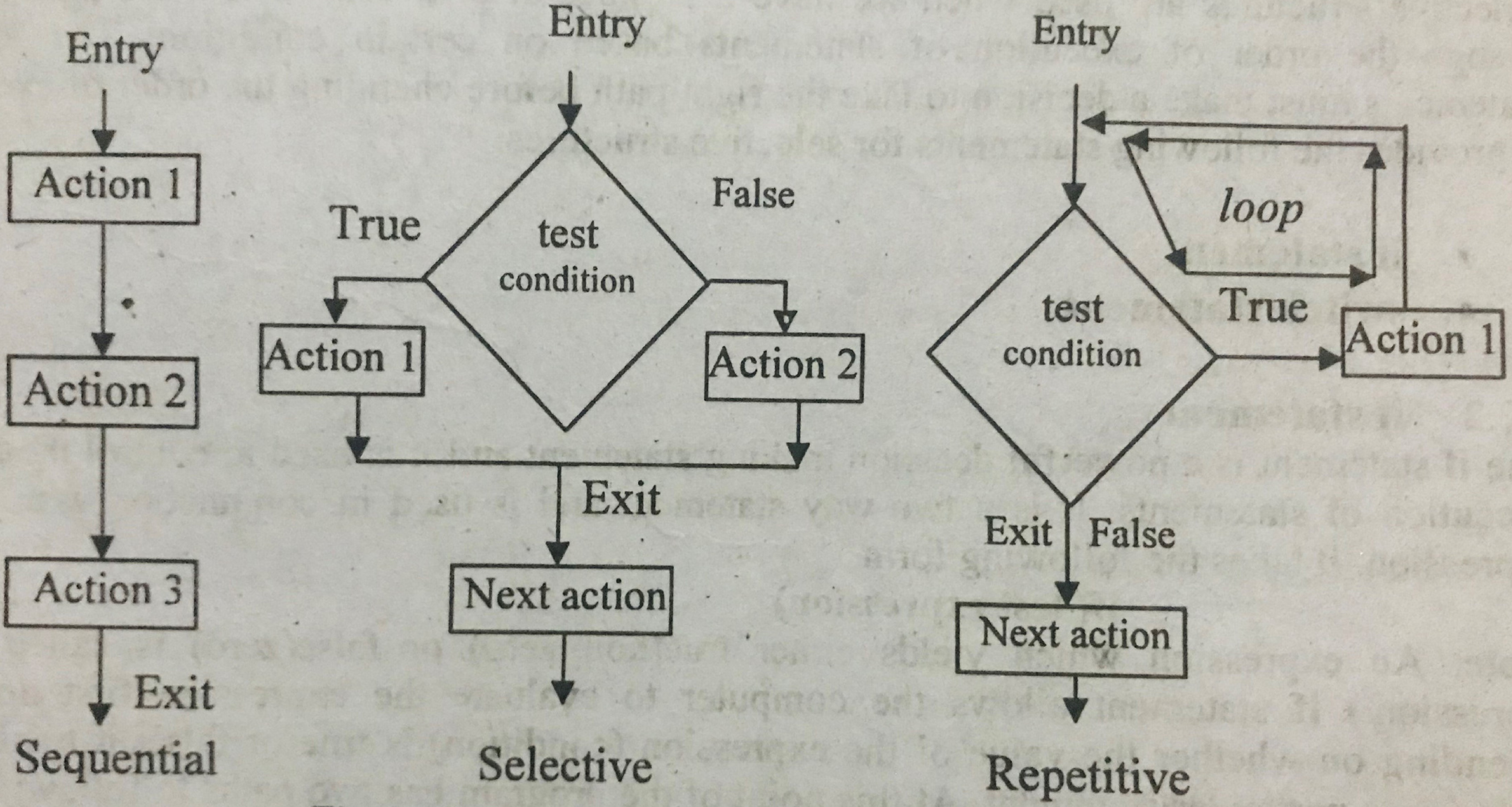


Fig. 5.1 Basic control structures

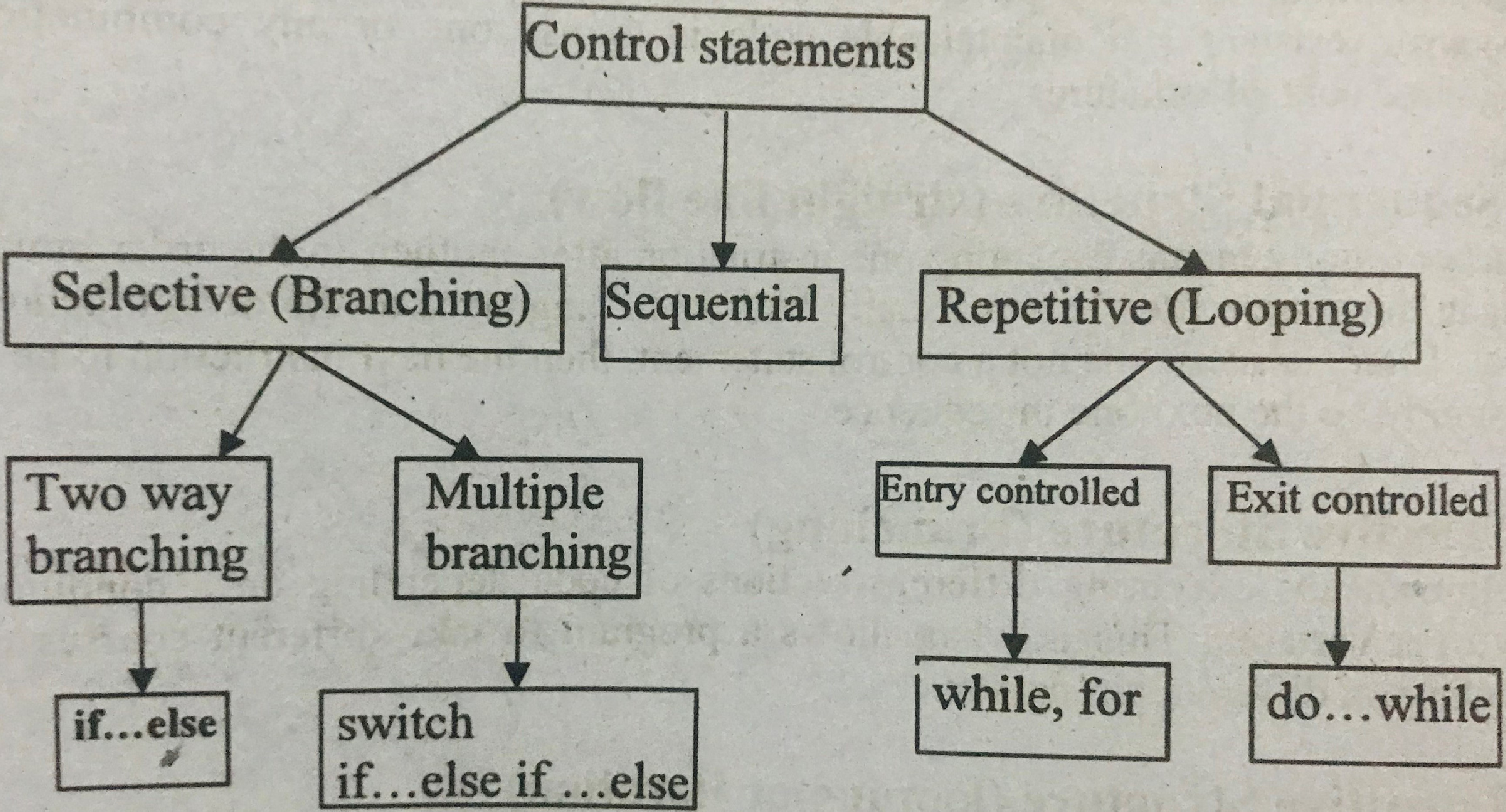


Fig. 5.2 Classification of C control statements

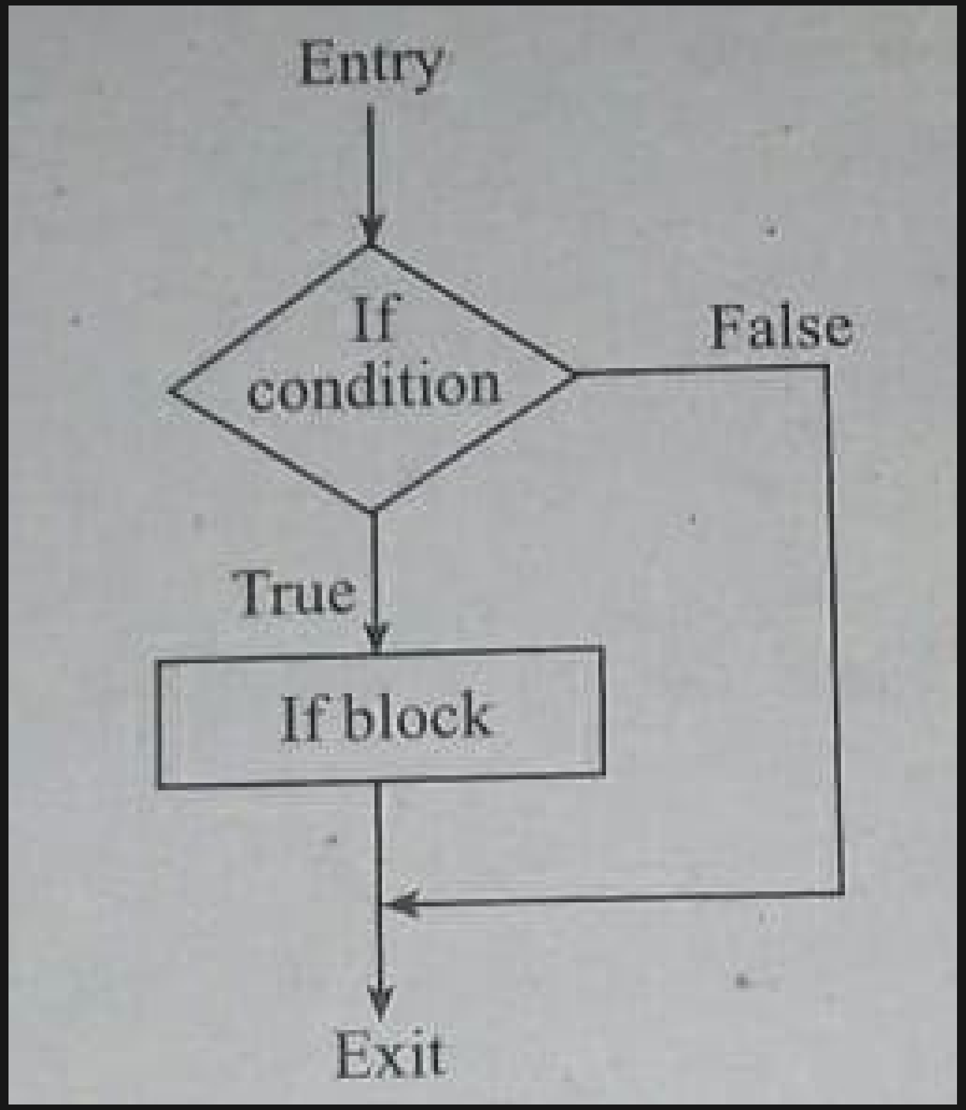
Conditional Structure

- Conditional statements help you to make a decision based on certain conditions.
- These conditions are specified be a set of conditional statements having boolean expressions which are evaluated to a boolean value true or false.
- There are following types of conditional statements in C
    a. If Statement
    b. If-else Statement
    c. Nested if-else Statement
    d. If-else-if ladder
    e. Switch Statement

a. If Statement

- The single if statement in C language is used to execute the code if a condition is true.
- It is also called one way selection statement.

Flowchart:



Syntax:

```
if(condition)
{
      statement(s);
}
```

How "if" statement works?

- If the condition is evaluated to non-zero(true) then if-block statements are executed
- If the condition is evaluated to zero(false) then control passes to the next statement following it

EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int n;
	printf("enter a number");
	scanf("%d",&n);
	if(n % 2 == 0)
	{
		printf("%d is even",n);
	}
	getch();
}
```

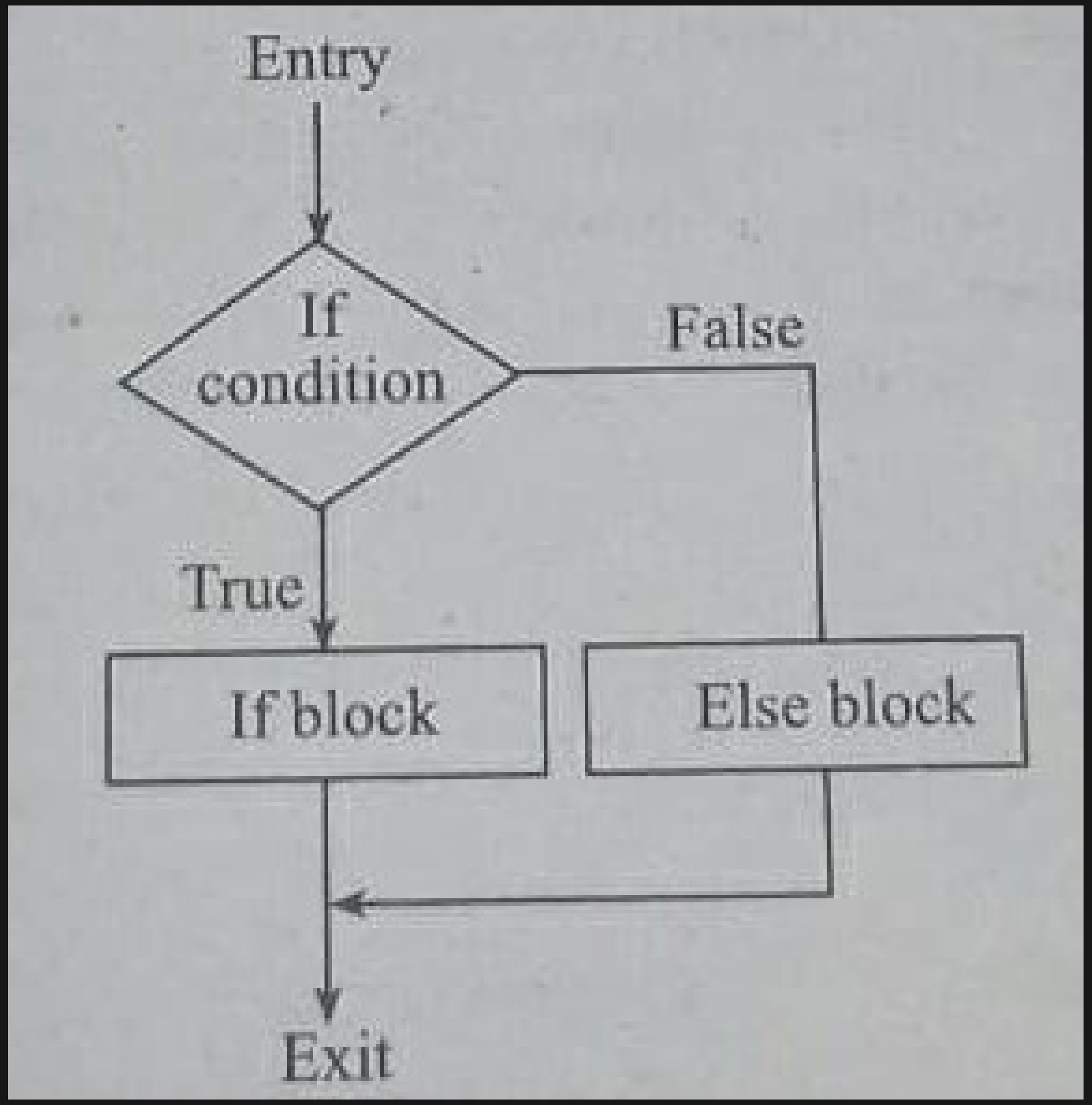
b. If...else Statement

- The if...else statement in C language is used to execute the code if a condition is true or false.
- It is also called two way selection statement.

Syntax:

```
if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

Flowchart:



How "if...else" statement works?

- If the condition is evaluated to non-zero(true) then if-block statements are executed
- If the condition is evaluated to zero(false) then else block statements are executed

EXAMPLE:

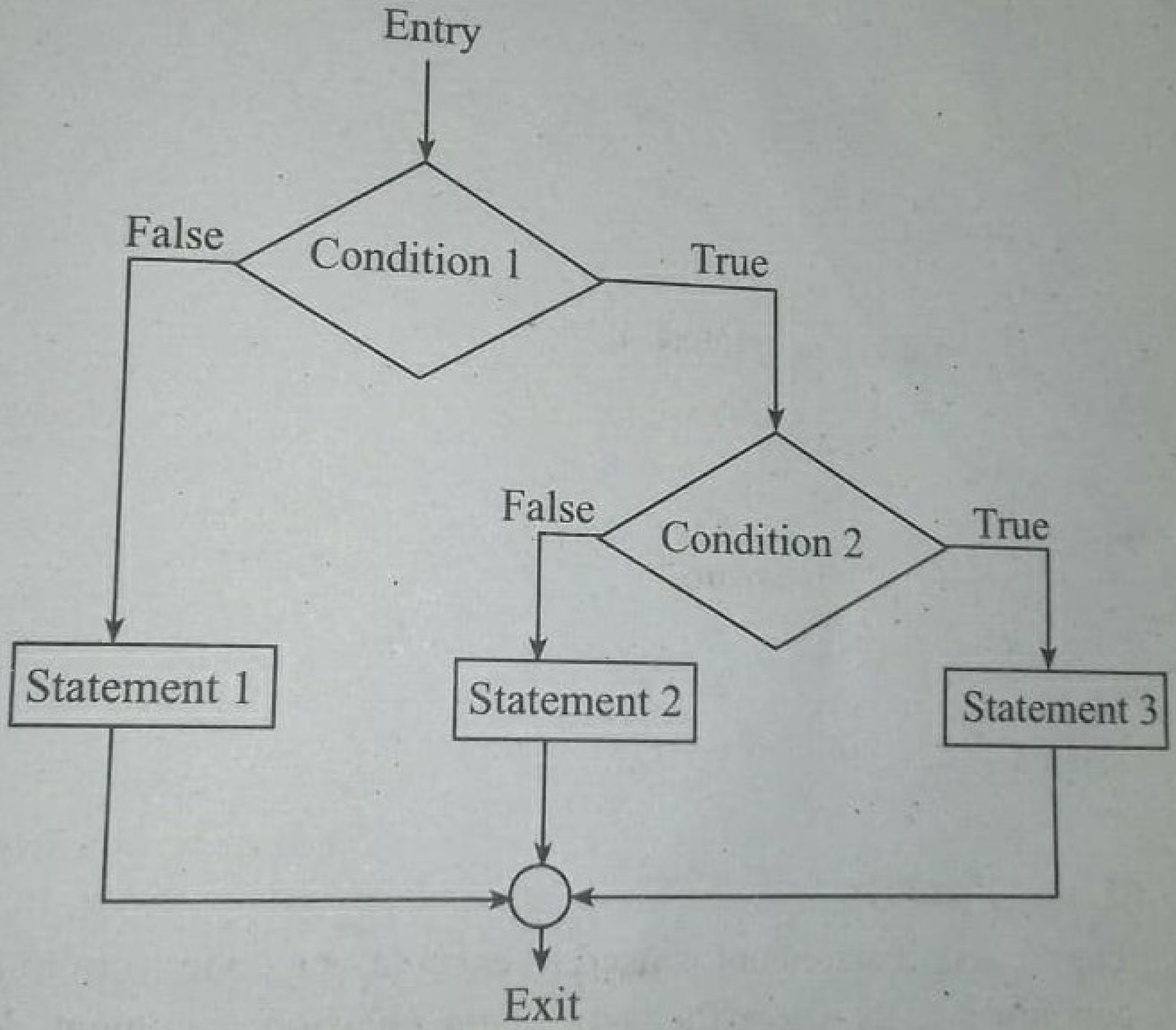
```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n;
    printf("enter a number");
    scanf("%d", &n);
    if(n % 2 == 0)
    {
        printf("%d is even", n);
    }
    else
    {
        printf("%d is odd", n);
    }
    getch();
}
```

3. Nested if...else statement

- The nested if...else statement is used when a program requires more than one test expressions. It is also called a multi-way selection statement.
- When a series of the decision are involved in a statement, we use if else statement in nested form.

Syntax:

```
if(condition1)
{
    if (condition2)
    {
        statement1;
    }
    else
    {
        statement2;
    }
}
else
{
    statement3;
}
```


Entry
Condition 1
False
True
Condition 2
False
True
Statement 1
Statement 2
Statement 3
Exit

EXAMPLE:

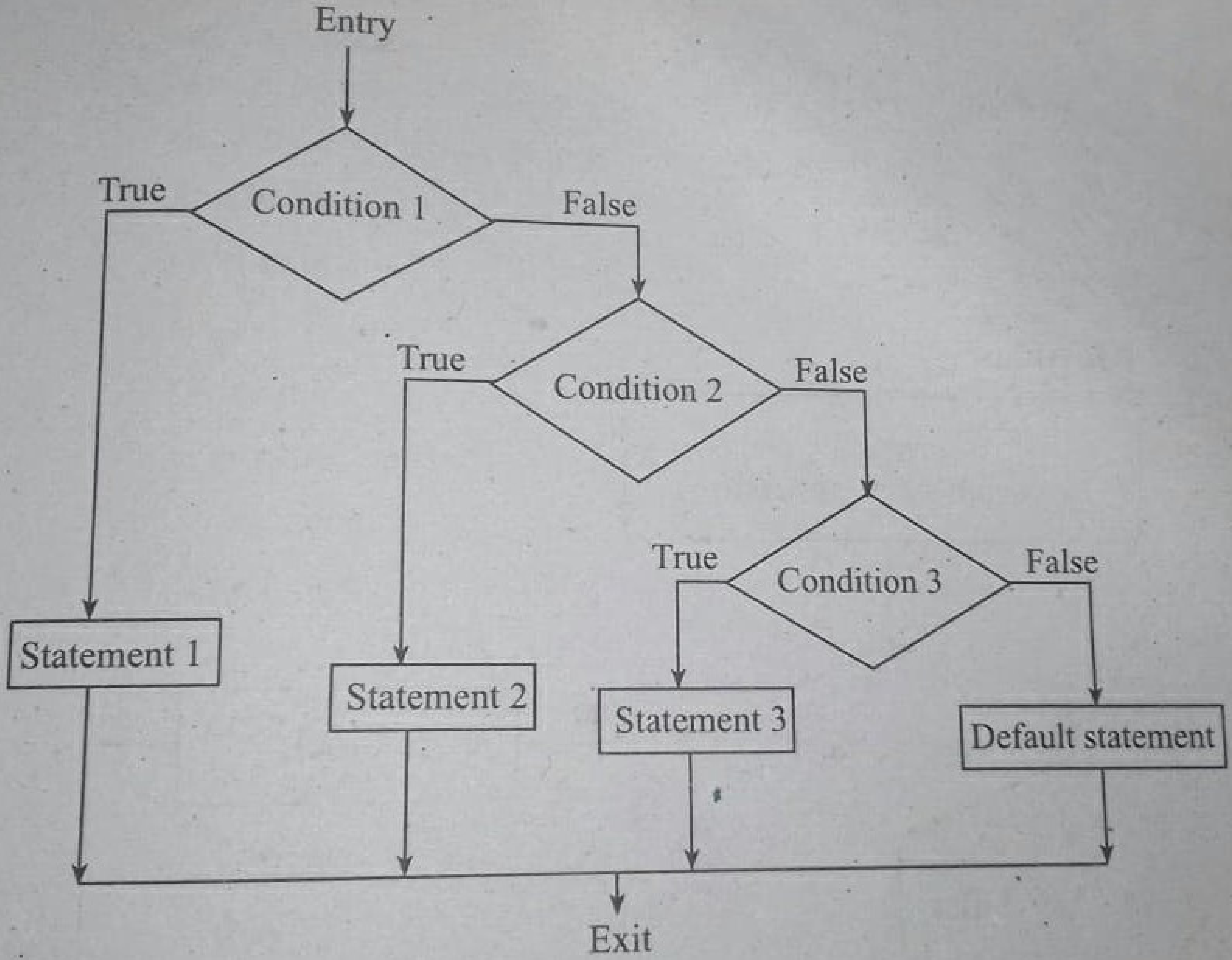
```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a, b, c;
    printf("Please Enter 3 numbers");
    scanf("%d %d %d",&a, &b, &c);
    if(a>b)
    {
        if(a>c)
        {
            printf("a is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
    else
    {
        if(b>c)
        {
            printf(" b is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
    getch();
}
```

4. if...else if ladder

- The if...else if statement is used to execute one code from multiple conditions. It is also called multipath decision statement.
- It is a chain of if...else statements in which each if statemnt is associated with else if statement and last would be an else statement.

Syntax:

```
if(condition1)
      statement1;
else if(condition2);
            statement2;
      else if(condition3)
                  statement;
                              .
                              .
            else
                  default_statement;
```

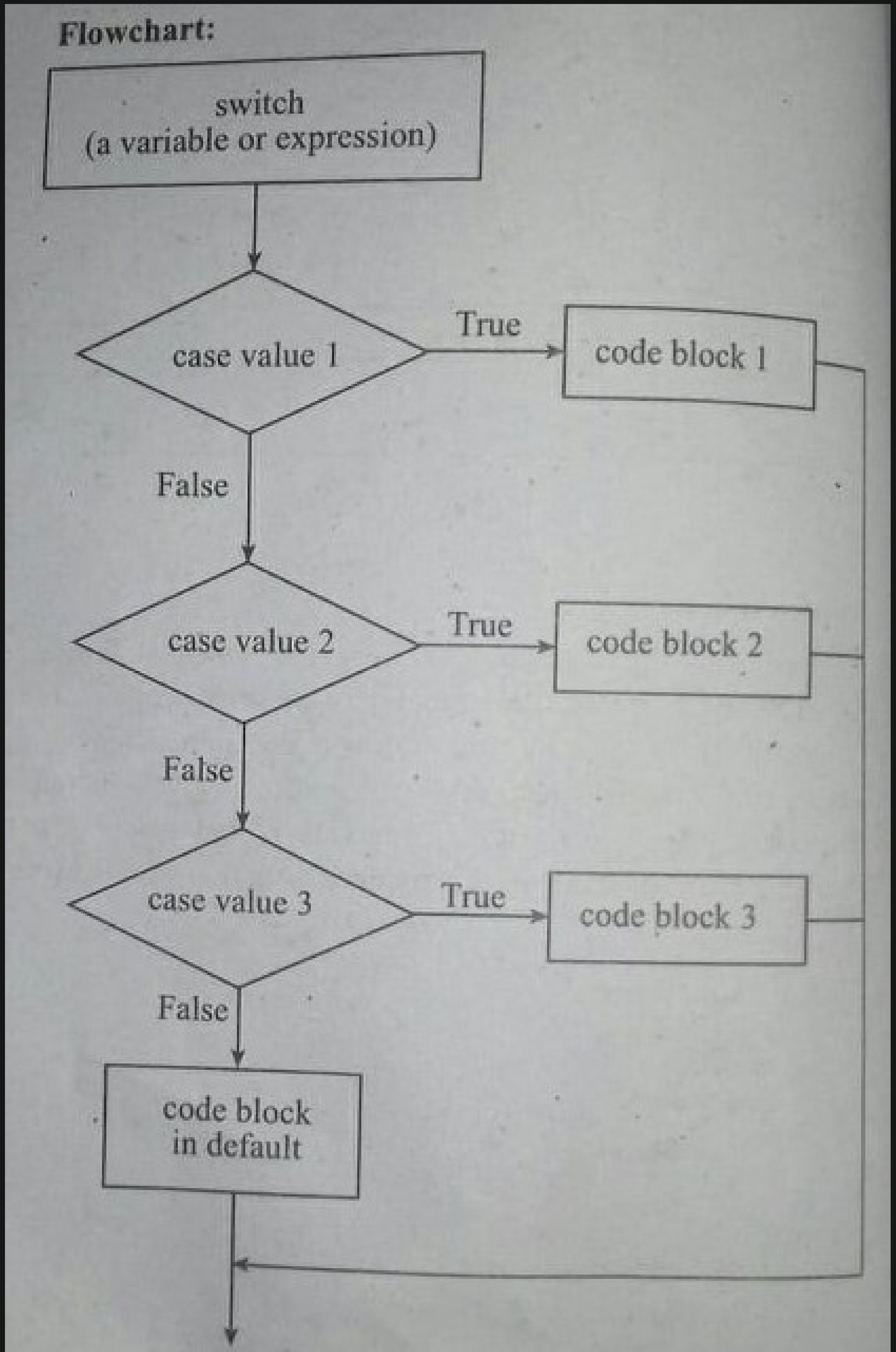

Entry
Condition 1
True
False
Condition 2
True
False
Condition 3
True
False
Statement 1
Statement 2
Statement 3
Default statement
Exit

5. Switch statement

- In multiway decision making program, when the number of alternatives increases, the complexity of such program increases dramatically.
- C has a built-in multiway decision statement known as "switch" which tests the value of a given variable (or expression) against a list of case values, and when a match is found, a block of statements associated with that case is executed.

Syntax:

```
switch (expression)
{
    case value-1:
        block-1;
        break;
    case value-2:
        block-2;
        break;
    .
    .
    .
    default:
        default-block;
        break;
}
```



Program to display the corresponding days of a week according to the numbers entered from 1 to 7

```
#include<stdio.h>
void main()
{
    int n;
    printf("Enter any no. from 1 to 7");
    scanf("%d", &n);
    switch (n)
    {
        case 1:
            printf("Sunday");
            break;
        case 2:
            printf("Monday");
            break;
        case 3:
            printf("Tuesday");
            break;
        case 4:
            printf("Wednesday")
            break;
        case 5:
            printf("Thursday");
            break;
        case 6:
            printf("Friday");
            break;
        case 7:
            printf("Saturday")
            break;
        default:
            printf("Enter valid number(1 to 7)");
            break;
    }
    getch();
}
```

- When the switch is executed, the value of the expression is successfully compared against the values: value1, value2,.....
- If a case is found whose value matches with the value of the expression, then only the block of statements that follows the case are executed.
- For example, consider the program to display the corresponding days of a week according to the numbers entered from 1 to 7.
- The break statements must be used at the end of each case block if it were not used, then all the cases from the one met will be executed. For example, if the value switch expression matches with that of case constant2, then all the statements in case statement2 as well as rest of the cases including default will be executed.
- The break statement tells the compiler to jump out of the switch case statement and execute the statement following the switch case construct.
- Default is also a case that is executed when the value of the switch expression does not match with any of the values of the case statements.
- Although the default case is optional, it is always recommended to include it as it handles any unexpected cases.

Repetitive Structure

- Repetitive structures, or loops, are used when a program needs to repeatedly process one or more instructions until some condition is met, at which time the loop ends.
- The process of performing the same task over and over again is called iteration, and C provides built-in iteration functionality.
- A loop executes the same section of program code over and over again, as long as a loop condition of some sort is met with each iteration.
- This section of code can be a single statement or a block of statements(a compound statement).
- There are three looping structures in C:
    a. for loop
    b. do......while loop
    c. while loop

a. for loop

- It is a count controlled loop which is used when some statement is to be repeated certain number of times.

Syntax:

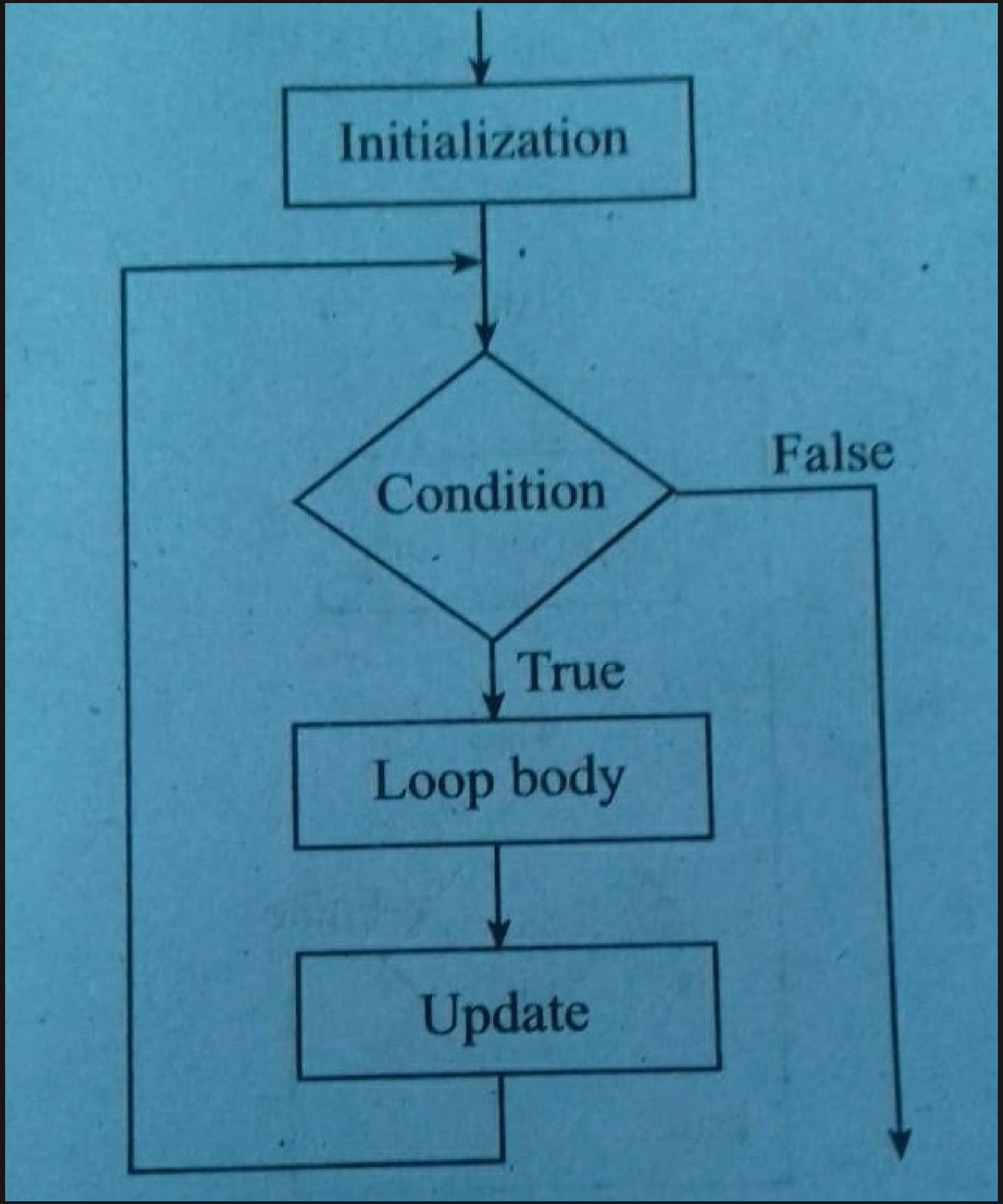
```
for(initialization; condition; update)
{
        body of loop; //statement(s) to be executed repeatedly
}
```

How "for" statement works?

- The initialization statement is executed only once
- Then, the test expression is evaluated. If the test expresion is evaluated to false, the for loop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated.
- Again the test expression is evaluated

This process goes on until the test expression is false. When the test expression is false, the loop terminates

Flowchart:



EXAMPLE:

```
#include<stdio.h>
void main()
{
        int i;
        for(i = 1; i<=10; i++)
        {
                printf("%d",i);
        }
}
```

Output:

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

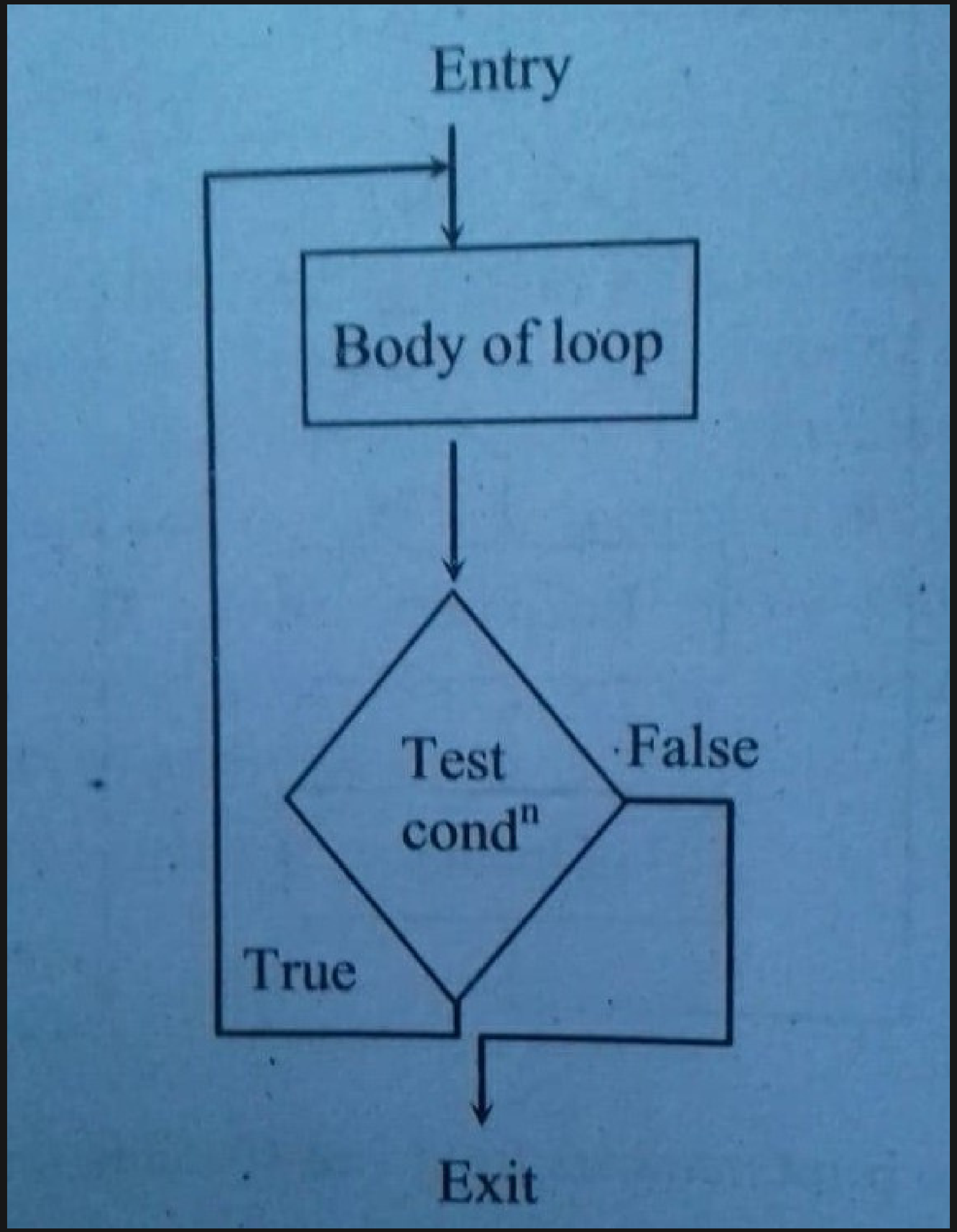
b. do.....while loop

- "do-while loop" is exit-controlled loop(condition controlled) in which test condition is evaluated at the end of loop.
- The body of the loop executes at least once without depending on the test condition and continues until the condition is true.

Syntax:

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

Flowchart:



EXAMPLE:

```
//Program to print numbers from 1 to 10 using do...while loop
#include<stdio.h>
void main()
{
    int i;
    i = 1;
    do
    {
        printf("%d",i);
        i++;
    }while(i <= 10);
}
```

Output:

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

Here, loop is repeated till value of i is less than ot equal to 10.

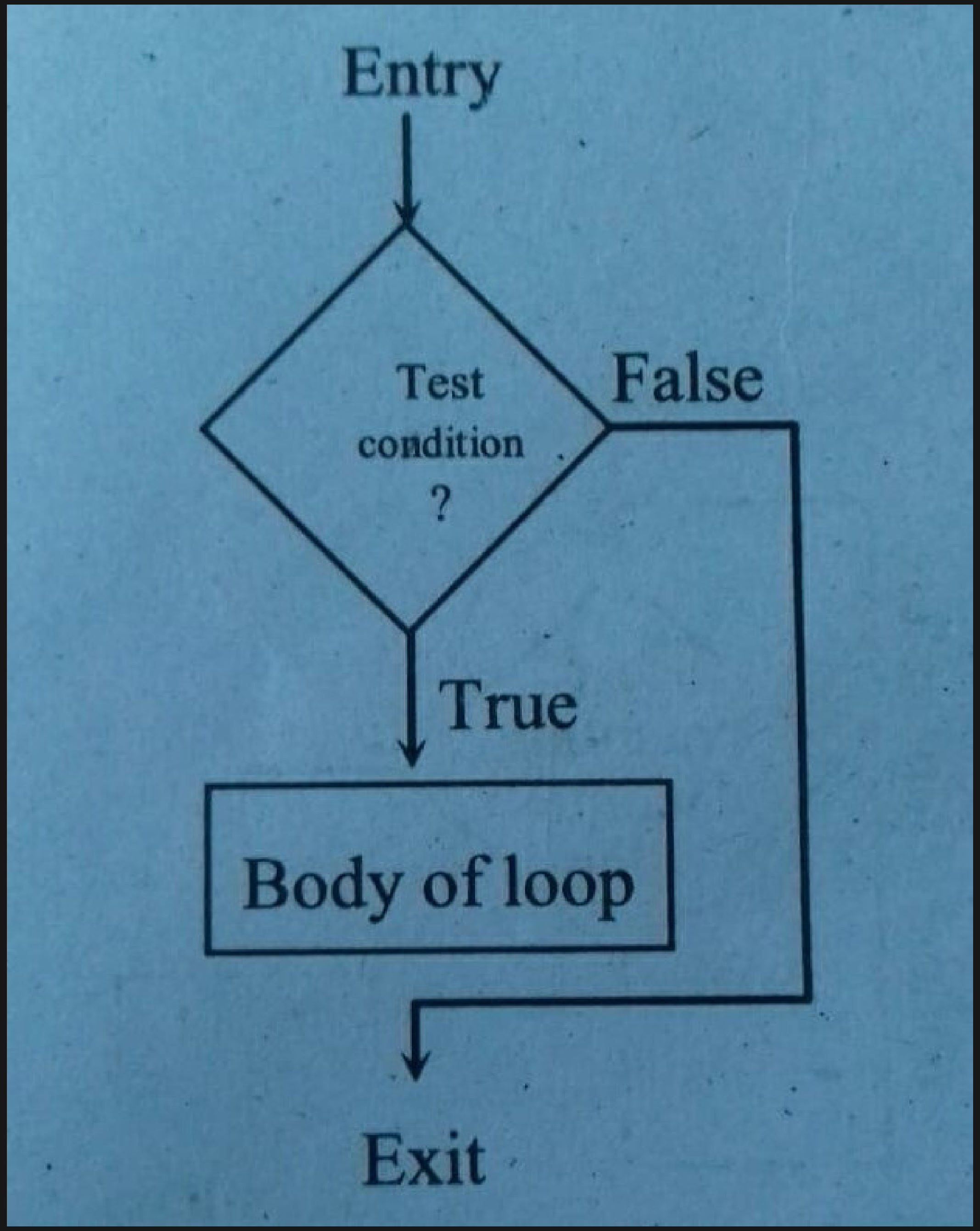
c. while loop

- "While loop" is entry-controlled loop (condition controlled) in which test condition is evaluated at the beginning of the loop execution.
- If the condition is true, only then the body of the loop executes, or else it does not.

Syntax:

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

Flowchart:



Example:

//program to print numbers from 1 to 10 using while loop.

```
#include<stdio.h>
void main()
{
    int i;
    i = 1;
    while(i<=10)
    {
        printf("%d ", i);
        i++;
    }
}
```

Output:

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

Here, loop is repeated only if the value of i is less than or equal to 10.

Jumps in Loops

- There are two statements available that are used to jump out from the loop after some condition is satisfied.
- They are useful in solving various problems related to repetitive structure.

1. Break statement
2. Continue statement

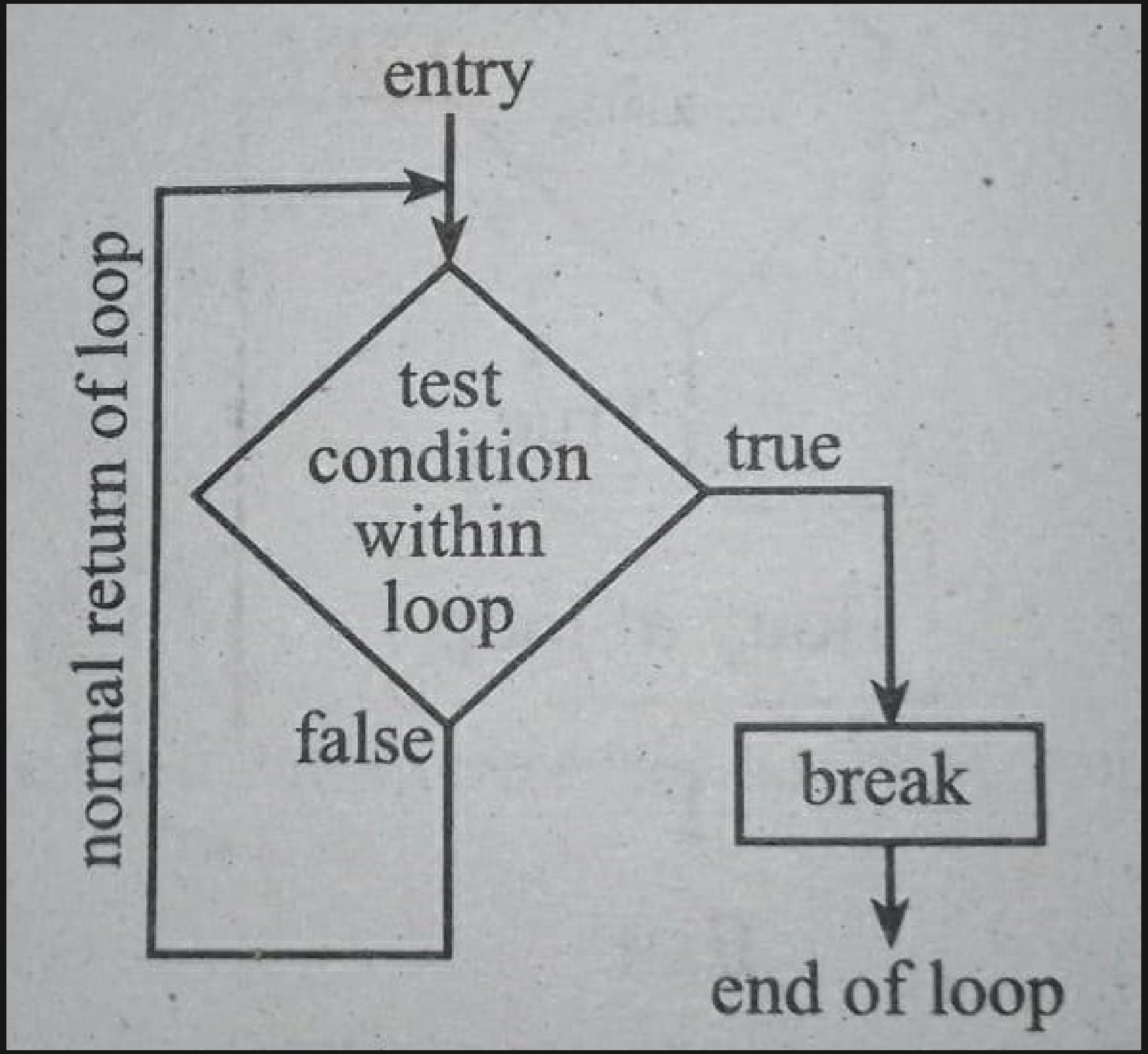
1. Break statement

- It is used to jump out of a loop. It terminates the execution of the nearest enclosing loop.

Syntax:

```
break;
```

Flowchart:



Example:

```
do
{
        scanf("%d",&n);
        if(n == 0)
                break;
        printf("%d", n);
}while(1);
```

Here the numbers are read until user enters 0(zero). All the numbers entered except 0 are displayed on the screen.

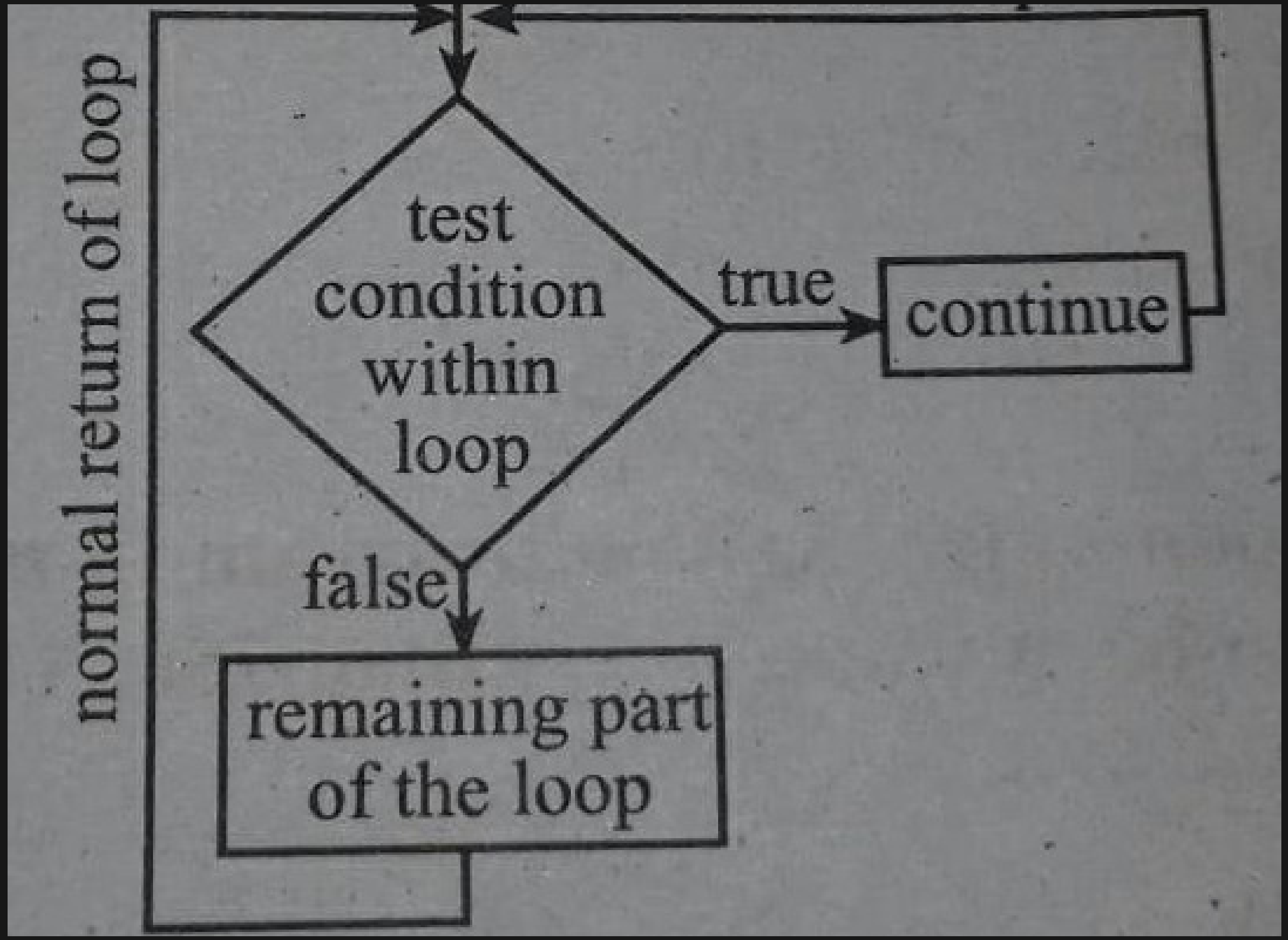
2. Continue statement

- It is used to bypass the remaining part of the current pass of a loop.
- The loop will not be terminated when a continue statement is encountered.

Syntax:

```
continue;
```

Flowchart:



Example:

```
for(i=1; i<=10; i++)
{
        if(i == 5 || i == 6)
                continue;
        printf("%d",i);
}
```

Here the numbers from 1 to 10 except 5 and 6 are printed on the screen.

Function

Introduction

- A Function is a self-contained program segment that carries out some specific, well-defined task.
- Functions are used to encapsulate a set of operations and return information to the main program or calling routine.
- A C program is usually made up of many small functions, each performing a particular task, rather than one single large main() function.

Function is necessary in programming because the use of function provides several benefits:

1. It makes programs significantly easier to understand and maintain.
2. Well written functions may be reused in multiple programs.
3. Different programmers working on one large project can divide the workload by writing different functions.
4. Programs that use functions are easier to design, program, debug and maintain

Types of functions:

There are two types of functions. They are:

- Library Functions
- User-defined Functions

- **Library Functions:**
  - Library functions are predefined functions/built in functions in C library.
  - They are part of header files (such as stdio.h, math.h) which is called at runtime.
  - The function name, its return type, their argument number and types have been already defined and can't be changed.

    Example: printf(), scanf(), sqrt(), pow() etc.

- **User-defined Function:**
  - The functions which are developed by user at the time of writing a program are called user defined functions.
  - These functions can be defined and used by the programmers in C programs according to their requirements.

Example: Here **addNumbers()** is an user defined function.

```
//declaring an user defined function which has been defined later.
float addNumbers(float, float);                    //function declaration(function prototype)
int main()
{
        float result;
        /* Calling user-defined function from the main function */
        result = addNumbers(0.5,0.8);
        printf(" Sum = %f ",result);
        return 0;
}
```

```
//defining an user defined function which has been declared earlier.
float addNumbers(float a, float b)
{
    float s;
    s = a+b;
    return s;       //return statement with return expression
}
```

Output: Sum = 1.300000

Concept associated with functions:

- Function declaration or prototype
- Function definition (Function declarator or function body
- Passing arguments
- Return statement
- Function call
- Combination of function declaration and function definition

1) Function declaration or prototype:

A function declaration provides the following information to the compiler:

- The name of the function.
- The type of the value to be returned (optional, default return type is integer).
- The number of arguments that must be supplied in a call to the function.
- The type of arguments that must be supplied in a call to the function.

When a function call is encountered, the compiler checks the function call with its declaration so that the correct argument types are used.

syntax: ***return_type function_name(type1 , type2 , type3 , ... , typeN);***

2) Function definition(Function declarator and function body)

A function definition has two principal components:

a) Function declarator(The first line of the function definition): In function declarator and the declaration, we must use the same function name, number of arguments, argument types and the return type same as in the function declaration but does not have semicolon at the end.

syntax: ***return_type function_name(type1 argument1 , type2 argument2 , type3 argument3, ... , typeN argumentN)***

Note: where argument1, argument2, argument3...argumentN are called formal arguments or formal parameters which are local to the function.

b) Body of the function: Function declarator is followed by the function body. It is composed of statements that make up the function, delimited by braces which define the actions to be taken by the function.

3) Passing arguments

Providing values to the formal arguments of called function through the actual arguments of the calling function is called passing arguments.

Note: The function which calls other function is called the calling function and the function which is called by other function is called the called function. In the above example, ***main*** is a calling function because it calls the add function and ***add*** function is a called function because it is called by the main function. Here add function can call other functions also. At that time, the add function becomes a calling function

4) Return Statement

A function may or may not send back any value to the calling function. If it does, it is through the return statement. While it is possible to pass to the called function any number of arguments but the called function can only return one value per call, at the most. The return statement can take one of the following forms.

Synatx: ***return or return (expression);***

The plain return does not return any value. It acts as the closing brace of the function. When a return is encountered,the control is immediately passed back to the calling function.

```
Example:   if(error)
            return;
```

The second form of return expression returns the value of the expression.

It is possible for a function to have multiple return statements as in the following segment:

```
int calculate(char ch)
{
      switch(ch)
```

```
    {
          case '+':
                      return 1;
          case '-':
                      return 2;
          case '*':
                      return 3;
          case '/':
                      return 4;
          default:
                      return 0;
    }
}
```

If no value is to be returned, return statement need not be present

5) Function Call

A function is a inactive part of the program which comes into life when a call is made to the function. A function call is specified by the function name followed by the values of parameters enclosed within parentheses, terminated by a semicolon. If you need to store the returned data in a variable, then assign this call to a variable.

Syntax: ***return_type_variable = function_name (arg1, arg2,.....);***

where arg1, arg2,.... are the actual arguments that are passed to the function. The values of actual arguments are copied to the formal arguments respectively.

Note: *The number, type and order of arguments in the function declaration, function call and function declarator must be the same.*

6) Elimination of function declaration

If the functions are defined before they are called, then the declaration is unnecessary. In the example below, the function add is defined before main i.e.,it is defined before calling it.

```
#include<stdio.h>
float add( int p, int q)    /*defining function above the main function to eliminate function
declaration part*/
{
        float s;
        s = p+q;
        return s;
}
void main()
{
        int a;
        float b, sum;
        printf(" Enter the values to be added:\n");
        scanf("%d%f", &a, &b);
        sum = add( a , b);
        printf("The sum is %d", sum);
}
```

Types of User Defined Functions

Depending upon the presence of arguments and the return type, user defined functions can be classified into four categories:

1. Function with no arguments and no return type
2. Function with no arguments but having return type
3. Function with arguments and no return type
4. Function with both arguments and return type

1. Function with no arguments and no return type:

Function with no argument means the called function does not receive any data from calling function and Function with no return value means calling function does not receive any data from the called function. So there is no data transfer between calling and called function.

C program to calculate the area of square using the function with no arguments and no return values

```
#include<stdio.h>
void area();
int main()
{
    area();
    return 0;
}
void area()
{
    int square_area, side;
    printf(" Enter the side of the square: ");
    scanf("%d", side);
    square_area =  side * side;
    printf("Area of aquare = %d", square_area);
}
```

Explanation:

In the program aside, area(); function calculates area and no arguments are passed to this function. The return type of this function is void and hence return nothing.

2. Function with no arguments but having return type:

As said earlier function with no arguments means called function does not receive any data from calling function and function with one return value means one result will be sent back to the caller from the function.

C program to calculate the area of square using the function with no arguments and one return values

```
#include<stdio.h>
int area();
int main()
{
    int square_area;
    square_area = area();
    printf("Area of aquare = %d", square_area);
    return 0;
}
```

```
int area()
{
    int square_area, side;
    printf(" Enter the side of the square: ");
    scanf("%d", side);
    square_area = side * side;
    return square_area;
}
```

Explanation: In this function int area(); no arguments are passed but it returns an integer value square_area.

3. Function with arguments and no return type:

Here function will accept data from the calling function as there are arguments, however, since there is no return type nothing will be returned to the calling program. So it's a one-way type communication.

C program to calculate the area of square using the function with arguments and no return values

```
#include<stdio.h>
void area(int);    //function declaration
int main()
{
      int square_side;
      printf("Enter the side of the square");
      scanf("%d", &square_side);
      area(square_side);    //function call
      return 0;
}
```

```
void area(int side)
{
      int square_area,;
      square_area = side * side;
      printf("Area = %d", square_area);
}
```

Explanation: In this function, the integer value entered by the user in square_side variable is passed to area();. The called function has void as a return type as a result, it doesn't return value.

4. Function with both arguments and return type:

Function with arguments and one return value means both the calling function and called function will receive data from each other. It's like a dual communication.

C program to calculate the area of square using the function with arguments and return values

```
#include<stdio.h>
int area(int);
int main()
{
    int square_side, square_area;
    printf("Enter the side of the square");
    scanf("%d", &square_side);
    square_area = area(square_side); //function call
    printf("Area = %d", square_area);
    return 0;
}
```

```
int area(int side)
{
    int square_area,;
    square_area = side * side;
    return square_area;
}
```

Ways of passing arguments to a Function

We can pass arguments to a function in two ways:

i) pass by value: Pass by value means to call the function by passing the value as argument to the function. In this method, changes made to the formal argument in the called function has no effect on the values of actual argument in the calling function.

Example:

```
#include<stdio.h>
#include<conio.h>
void swap(int, int);
void main()
{
    int  x=2, y=3;
    printf("The values before swap are:");
    printf("x = %d and y = %d",x,y);
    swap( x, y);
    printf("\n The values after swap are:");
    printf("x = %d and y = %d", x, y);
    getch();
}
void swap( int p, int q)
{
        int temp;
        temp = p;
        p = q;
        q = temp;
}
```

Output:

```
The values before swap are: x = 2 and y = 3
The values after swap are: x = 2 and y = 3
```

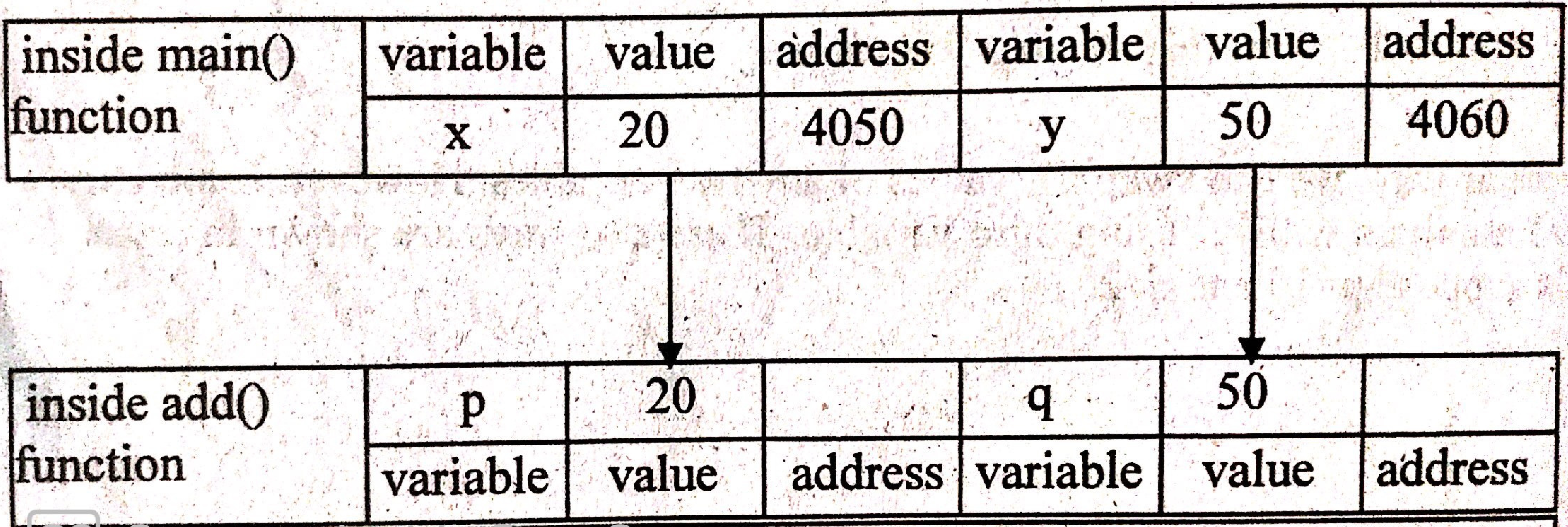


Fig:6.2 Illustration of passing by value.

ii) Pass by reference: Pass by reference means to call the function by passing address(reference) as argument to the function. In this method, we can change the value of actual argument from the called function.

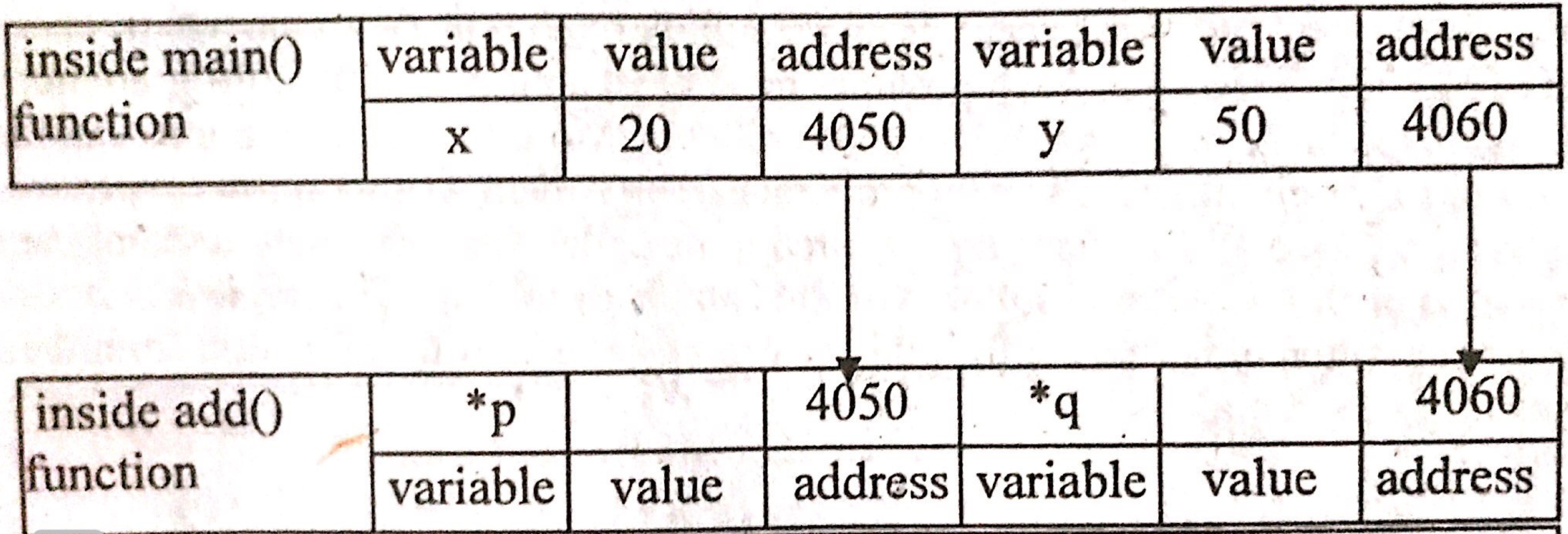
Example:

```
#include<stdio.h>
#include<conio.h>
void swap(int *, int *);
void main()
{
    int  x=2, y=3;
    printf("The values before swap are:");
    printf("x = %d and y = %d",x,y);
    swap( &x, &y); //pass by reference
    printf("\n The values after swap are:");
    printf("x = %d and y = %d", x, y);
    getch();
    }
```

```
void swap( int *p, int *q)
{
        int temp;
        temp = *p;
        *p = *q;
        *q = temp;
}
```

Output:

```
The values before swap are: x = 2 and y = 3
The values after swap are: x = 3 and y = 2
```



inside main() function	variable	value	address	variable	value	address
	x	20	4050	y	50	4060

inside add() function	*p		4050	*q		4060
	variable	value	address	variable	value	address

Fig:6.3 Illustration of passing by reference.

Write a program to add two numbers using function. Make the function return the value to the main function.

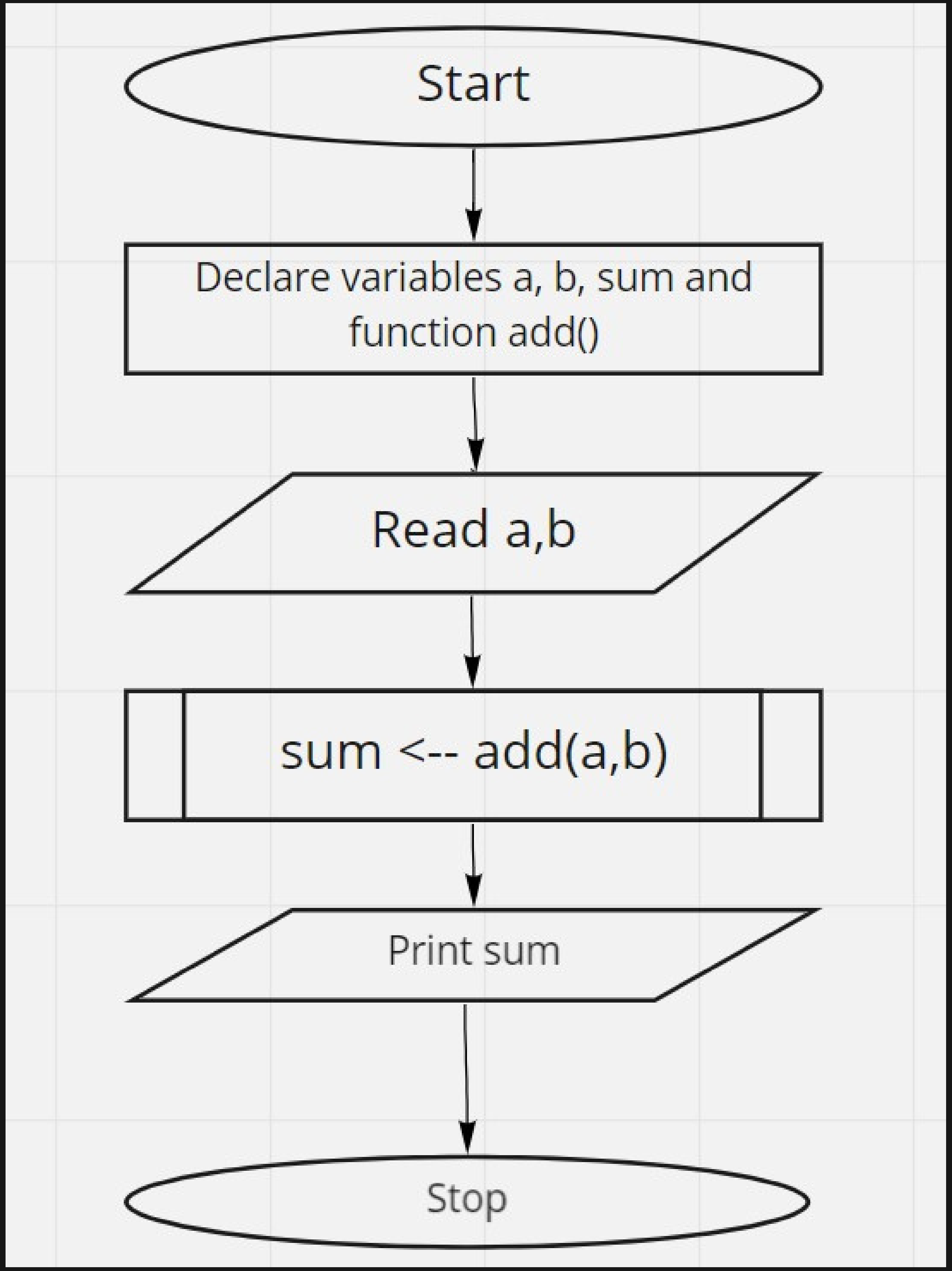
Algorithm of calling function:

```
Step 1: Start
Step 2: Declare local variables a,b, sum and function add()
Step 3: Read values of a and b
Step 4: Call function add() with argument a and b
        sum <-- add(a,b)
Step 5: Print sum
Step 6: Stop
```

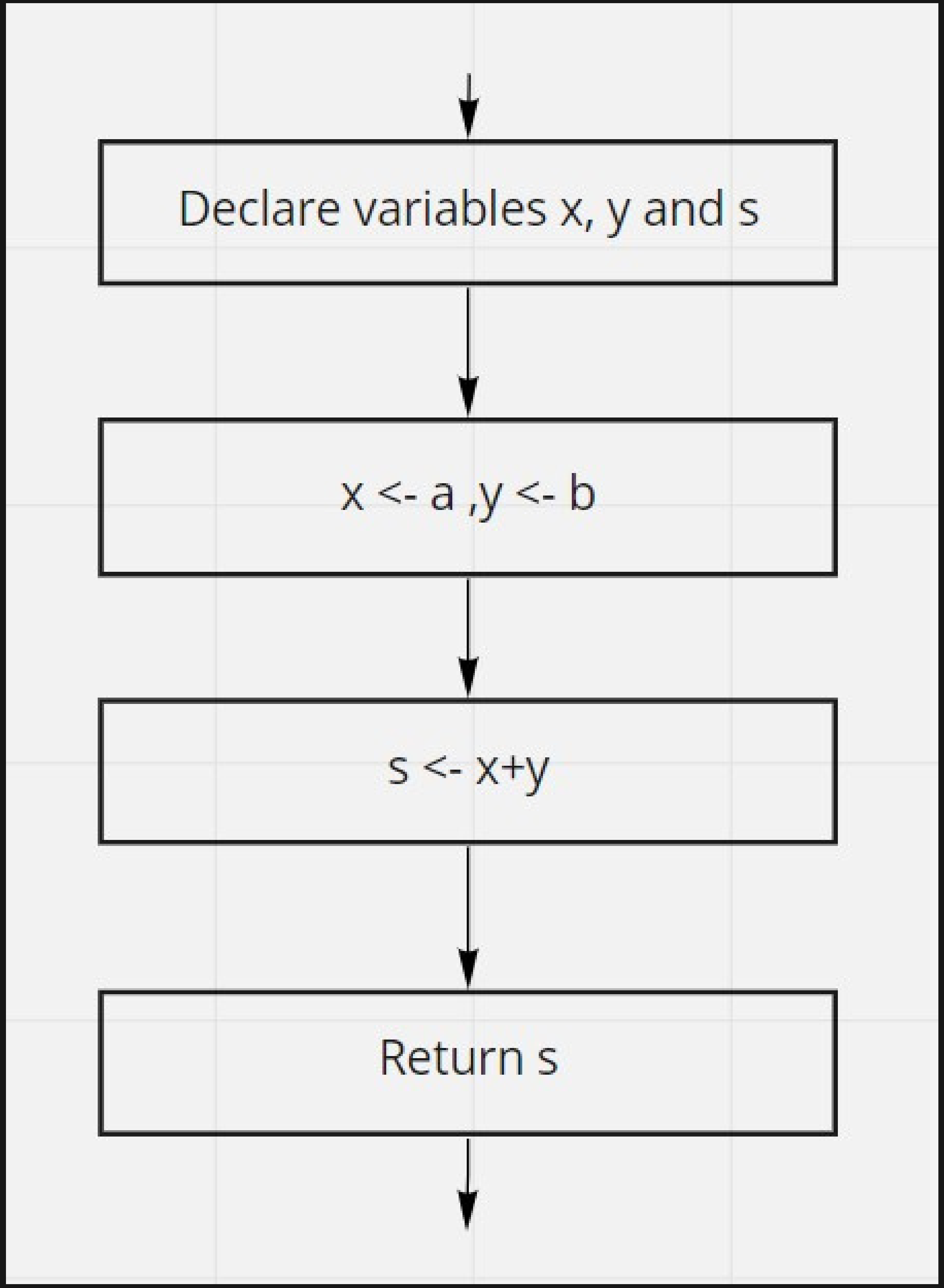
Algorithm of called function:

```
Step 1: Declare local variables x,y,s
Step 2: Assign a to x and b to y
Step 3: s <-- x + y
Step 4: Return s
```

Flowchart of calling function:



Flowchart of called function:



Source code:

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

int add(int,int);

void main()
{
	int a,b,sum;
	printf("Enter two numbers: ");
	scanf("%d %d",&a,&b);
	sum = add(a,b);
	printf("The sum of the given numbers is %d",sum);
	getch();
}
```

```
int add(int x, int y)
{
	int s;
	s = x  + y;
	return s;
}
```

Recursive Function

- Recursion is a programming method in which a function calls itself.
- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Two important conditions that must be satisfied by any recursive function are:
    - Each time a function calls itself, it must be closer to a solution.
    - There must be a decision criteria for stopping the process.
- Therefore, recursion is defining large and complex problems in terms of a smaller and more easily solvable problem.
- A function may direcctly or indirectly call itself in the course of its execution. If the function call is within its own body then the recursion is direct. If the function calls another function which in turn calls itself, then such recursion is indirect.

A recursive function has the following general form:

```
return_type  function_name( pass appropriate arguments)
{
     if it is a simple case
```

```
            return the simple value     //base case or stopping condition
        else
            call function with simpler version of problem
 }
```

```
/*an example to calculate the factorial of a given number using recursion */
#include<stdio.h>
int factorial(int);
int main()
{
      int n, fact;
     printf("Enter a humber");
     scanf("%d", n);
     fact = factorial(n);
     printf("The factorial of %d is %d", n, fact);
     return 0;
}
```

```
int factorial( int n);
{
     if(n<=1)
          return 1;
     else
          return( n * factorial(n-1);
}
```

In the above example, To calculate n!, we have multiplied the number with factorial of the number that is 1 less than that number. In other words, n! = n * (n-1)!

Let us say we need to find the value of 5!

5! = 5 * 4 * 3 * 2 * 1 = 120

This can be written as

5! = 5 * 4!

= 5 * 4 * 3!

= 5 * 4 * 3 * 2!

= 5 * 4 * 3 * 2 * 1! $\longleftarrow$ terminating condition

= 5 * 4 * 3 * 2 * 1

= 120

Nested Function

- C language also allows nesting of functions i.e. to use/call one function inside another function's body.
- It must be carefully used as it may lead to infinite nesting.

```
function1()
{
    //function1 body here
    function2();
    //function1 body here
}
```

- If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. That means, they will keep calling each other and the program will never terminate.

How is nested function different from recursion?

- Recursion is a special way of nesting functions, where a function calls itself inside it.
- We must have a certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
        //function1 body here
        function1();
        //function1 body here
}
```

Scope of variables

Scope of a variable determines over what part(s) of a program a variable is actually available for use.

1.Local

- These variables only exist inside the specific function that creates them.
- They are unknown to other functions.
- Local variables cease to exist once the function that created them is completed.
- They are recreated each time a function is executed or called.

2.Global

- These variables can be accessed by any function compromising the program.
- They do not get recreated if the function is recalled.
- To declare a global variable, declare it outside of all the functions.
- If a variable of the same name is declared both within a function and outside of it, the function will use the variable that was declared within it and ignore the global one.
- It is recommended to use as few global variables as possible.

Extent of variables

The time period during which memory is associated with a variable is called extent of variable

1.auto variables

- All variables declared within functions are auto by default.
- Variables declared auto can only be accessed within the function or nested block within which they are declared
- They are created when the control is entered into the functions and destroyed when the control is exited from the function.

auto int a;

2.register variables

- C allows the use of the prefix **register** in primitive variable declarations. Such variables are register variables.
- They are stored in the registers of the microprocessor
- The number of variables, which can be declared register, are limited

register int i;

3.static variables

- The value of the static variables persists until the end of the program.
- A variable can be declared static using the keyword **static** like **static** int p;
- Static variable may be internal or external type, depending on the place of declaration i.e. they may be internal static variable or external static variable.
- They are similar to auto variables except that they remain in existence(alive) throughout the remainder of the program.
- The static variables can be initialized only once.

```
#include<stdio.h>
void staticdemo();
void main()
{
    staticdemo();   /*loop can be used to call staticdemo() four times */
    staticdemo();
    staticdemo();
    staticdemo();
}
```

```
void staticdemo()
{
    static int p=150;
    p=p+10;
    printf("p=%d\n",p);
}
```

in the given example, p is static variable. It retains its previous value so that the old value is incremented by 1 in each function call. That is shown in the output.

Output:

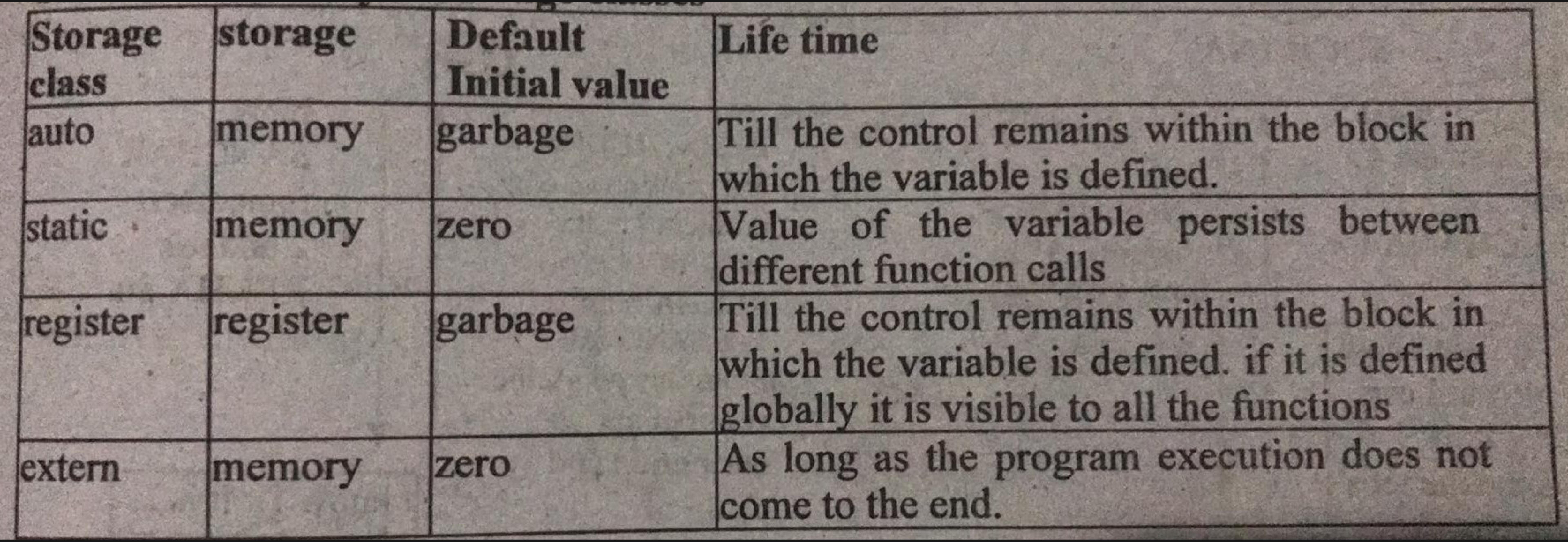
```
p=160
p=170
p=180
p=190
```

4.extern variables

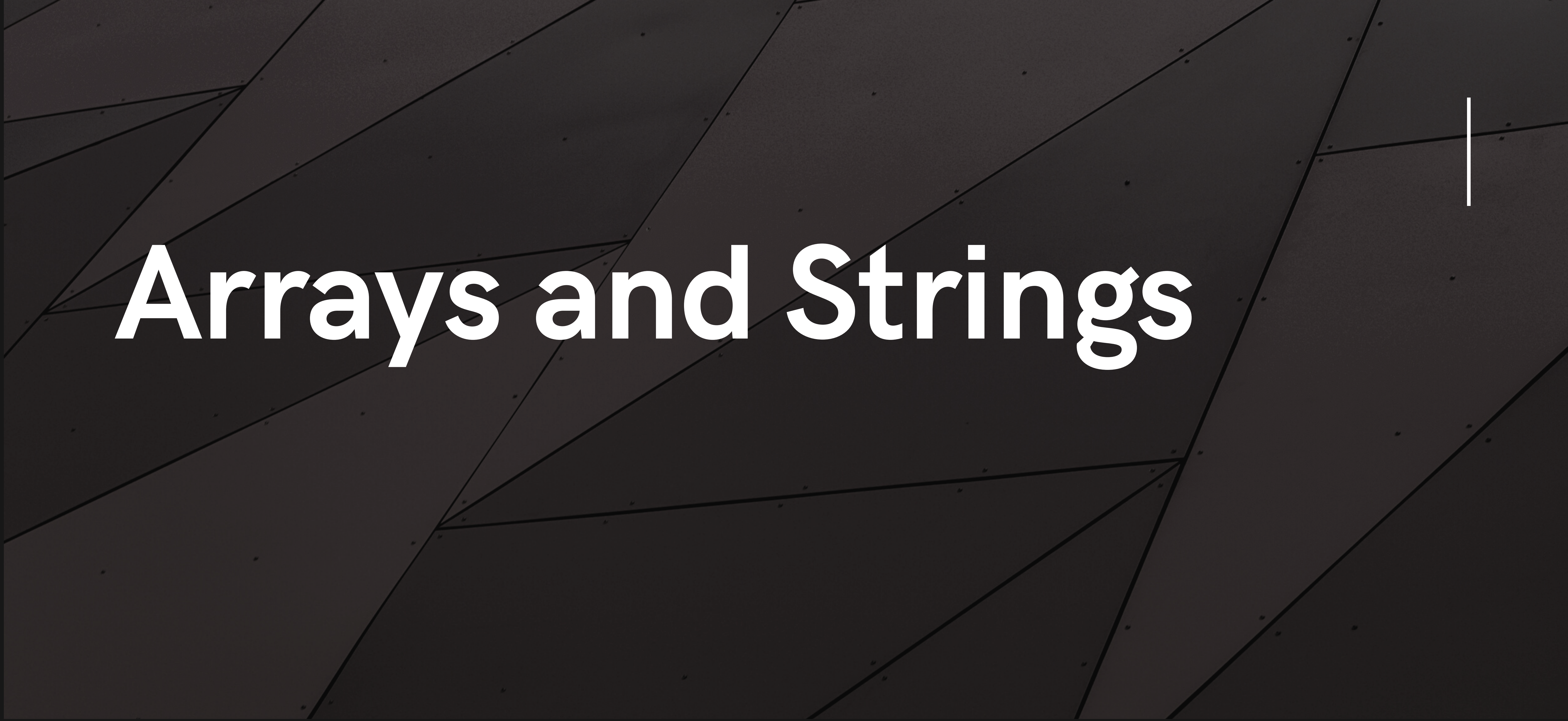
- This class is used to transmit information across the blocks and functions and even across files. Such variable is considered 'global' variables.
- If the variable is declared outside function it is extern by default.

extern int a;

Summary of storage classes



Storage class	storage	Default Initial value	Life time
auto	memory	garbage	Till the control remains within the block in which the variable is defined.
static	memory	zero	Value of the variable persists between different function calls
register	register	garbage	Till the control remains within the block in which the variable is defined. if it is defined globally it is visible to all the functions
extern	memory	zero	As long as the program execution does not come to the end.



Arrays and Strings

Array

- An array is a collection of data items of similar data type in contiguous memory location.
- Arrays are used when we require processing of a number of data items of same type i.e., having the common characteristics. For example, marks of students, salary of employees of a company.
- Therefore, arrays can be defined as a group of related data items that share a common name.
- If we want to use many elements of similar type then it is not feasible to declare all variables and also manipulate these elements. So, in this case we use array.
- Arrays are also called derived data type because they are derived from fundamental data types.
- The use of arrays allows for the development of smaller and more readable programs.

Types of Array

1) One-dimensional arrays

- Elements of the array can be represented either as a single column or as a single row

Declaration *Syntax*: data_type array_name[size];
e.g., int marks[5]; *//creates an array marks to store 5 elements of integer type*

To access individual element of an array, a subscript or index must be used as shown in the figure.

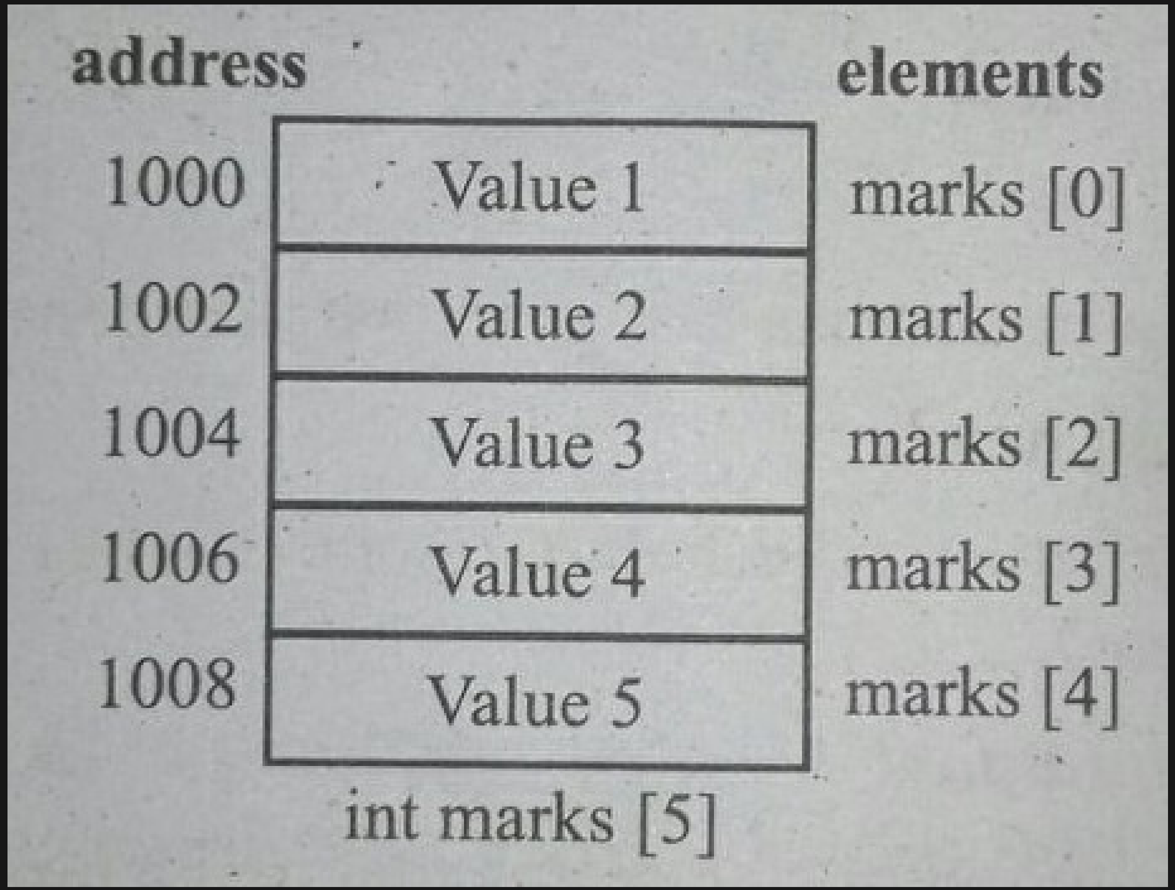


Fig. *Memory allocation for one dimensional array of type int*

Input/Output in 1D array

```
for(i = 0;i<5;i++)
{
        scanf("%d",&marks[i]); //input
}

for(i=0;i<5;i++)
{
        printf("%d",marks[i]); //output
}
```

2) Two- dimensional arrays

- As we know, in one-dimensional arrays, data items are arranged in one direction.
- Whereas data items in two-dimensional arrays are arranged in two directions: horizontal and vertical.
- The data items arranged in horizontal direction are referred as rows and the data items arranged in vertical direction are referred as columns.

Declaration syntax: *data_type array_name[row_size][column_size];*

e.g. int marks[3][3]; *// creates a two dimensional array to store 9 elements of integer type. There are 3 rows and 3 columns in the array marks.*

To access individual elements of two dimensional array, a pair of subscript is used to indicate row and column position as shown in the figure.

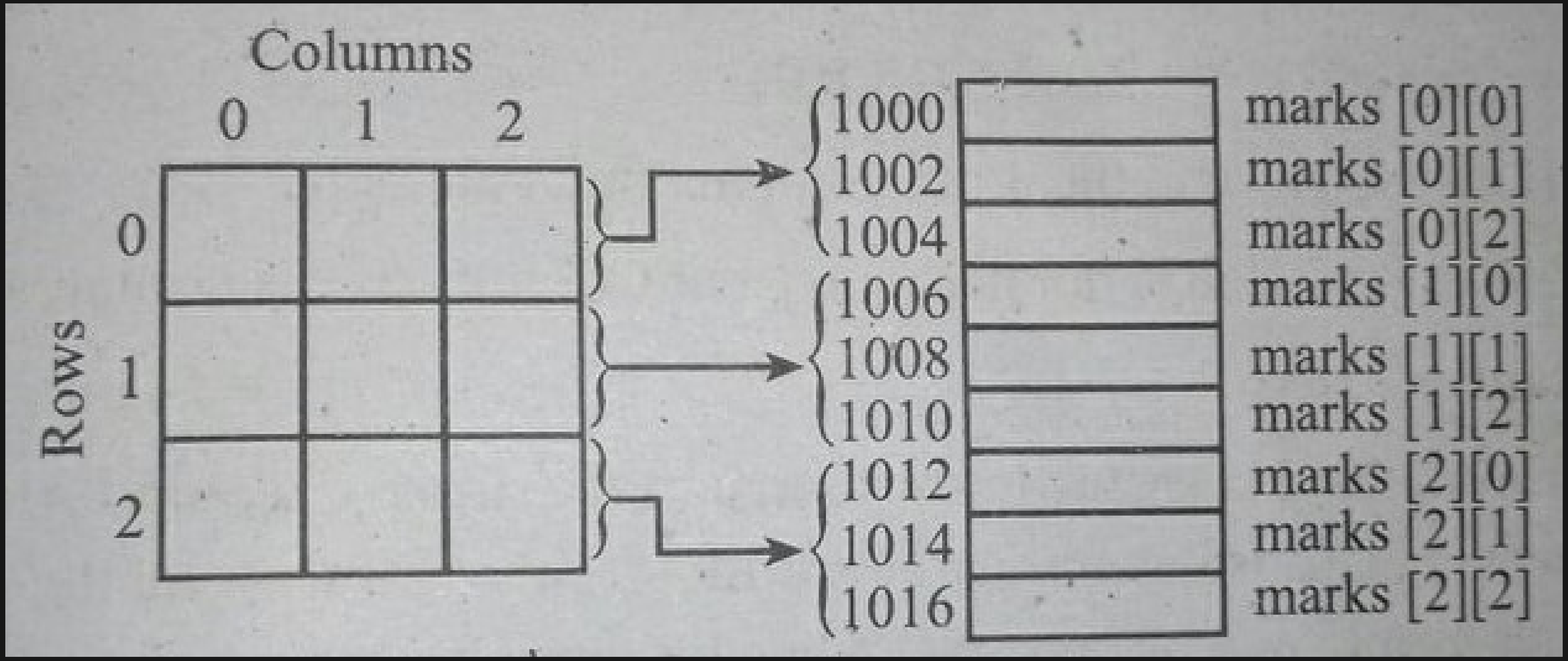


Fig. *Memory allocation for each element of two-dimensional array*

Input/Output in 2D array

```
for( i=0 ; i<3 ; i++)
{
        for( j=0 ; j<3 ; j++)
        {
                scanf("%d", &marks[i][j]);      //input
        }
}
for( i=0 ; i<3 ; i++)
{
        for( j=0 ; j<3 : j++)
        {
                printf("%d", marks[i][j]);    //output
        }
}
```

```
// A program that reads two dimensional arrays, adds the corresponding elements and
displays the result on the screen.
#include<stdio.h>
void main()
{
    int   i, j, arr1[5][5], arr2[5][5], arr3[5][5], r1, r2, c1, c2;
    printf( "Enter the maximum size of row and column of array 1:");
    scanf( "%d %d", &r1, &c1);
    printf( "Enter the maximum size of rows and column of array 2:");
    scanf( "%d %d", &r2, &c2);
    if( r1 == r2 && c1 == c2 )
    {
        for( i = 0 ; i < r1; i++)
        {
            for( j = 0; j < c1; j++)
            {
```

```
            printf( " arr1[%d][%d]: " ,i ,j );
            scanf( " %d " , &arr1[i][j]);
        }
    }
    for( i = 0 ; i < r2 ; i++)
    {
        for( j = 0 ; j < c2 ; j++)
        {
            printf( "  arr2[%d][%d]: ", i , j);
            scanf( " %d " , &arr2[i][j] );
        }
    }
    for( i = 0 ; i < r1 ; i++)
    {
        for( j = 0 ; j < c1 ; j++)
        {
                arr3[i] [j] = arr1[i] [j] + arr2[i] [j];
                printf( "  arr3[%d] [%d] = %d ", i, j, arr3[i][j] );
            }
        }
        else
            printf( " Array size mismatch.");
}
```

```
OUTPUT:
Enter the maximum size of row and column of array 1: 2  3
Enter the maximum size of row and column of array 2: 2  3
arr1[0][0] : 1                 arr2[0][0]: 7                 arr3[0][0] : 8
arr1[0][1] : 2                 arr2[0][1]: 8                 arr3[0][1] : 10
arr1[0][2] : 3                 arr2[0][2]: 9                 arr3[0][2] : 12
arr1[1][0] : 4                 arr2[1][0]: 10                arr3[1][0] : 14
arr1[1][1] : 5                 arr2[1][1]: 11                arr3[1][1] : 16
arr1[1][2]: 6                  arr2[1][2]: 12                arr3[1][2] : 18
```

Initialization in array:

- **One dimensional array:**
  We can initialize the elements of array in the same way as the ordinary variables when they are declared. The general form if initialization is:
  Syntax: *data_type array_name[ ] = { List of initializers };*
  e.g. int a[ ] = {1,2,3,4,5};
  Note: Size is not necessary while initializing.

- **Two dimensional array:**
  Syntax: *data_type array_name[ ] [column_size] = { List of initializers };*
  e.g: int a[ ] [3] = { {1,2,3}, {4,5,6}, {6,7,8} };
  int a[ ] [3] = { 1,2,3,4,5,6,7,8 };
  Here each row in the array contains three elements.
  Note: Row size is not necessary but column size is compulsory.

Important:

Errors that occur due to the mismatch of the array size and the number of the elements can cause overflow and underflow errors. Actually there is no bounds checking in array in C, hence it doesn't generate any real error but it may cause undesirable effects.

When input data is longer than will fit in the reserved space, if you don't truncate it, the data will overwrite other data in the memory. When this happens it is called an overflow. If the memory overwritten contained data essential to the operation of the program, this overflow causes a bug that, being intermittent, might be very hard to find.

e.g: int a[5] = { 1, 2, 3, 4, 5, 6 }

Similarly, when the input data is or appears to be shorter than the reserved space (due to errorneous assumptions, incorrect length values, or copying raw data as a C string) this is called an underflow error.

e.g: int a[5] = { 1, 2, 3 }

Passing Arrays to User Defined Functions

3 rules govern the passing of arrays to user-defined functions:

- The function must be called by passing only the name of the array and its size
- In the function definition, the formal parameters must be an array type; the size of the array does not need to be specified
- The function prototype must show that the argument is an array

1.Passing 1D Array to Function

- We can pass the whole array element from a function by passing the name of the array.
- The array name refers to the first byte of the array in memory and the address of rest of the elements in the array can be calculated using the array name and the index value of the element. Therefore, we simply pass the name of the array when an entire array needs to be passed as shown below:

```
void func(int [ ], int);

void main()
{
      int arr[5] = {1,2,3,4,5};
      func(arr,5);
}
```

```
void func(int arr[ ], int n)
{
      int i;
      for(i=0;i<n;i++)
            printf("%d",arr[i]);
}
```

2.Passing 2D Array to Function

- To pass a two-dimensional array to a function we have to pass the array name and size(row and column) as the actual argument.
- The size of dimension except the first must be included in the function prototype(declaration) and in function definition.

```
void display(int [ ][10], int, int); //function declaration
void main()
{
    int arr[10][10] , i , j;
    for(i = 0; i < 10; i++)
    {
        for(j = 0; j < 10; j++)
        {  scanf( " %d " , &arr[i][i]) ;  }
    }
    display(arr,10,10); //function call
}
```

```
void display(int b[ ][10], int m, int n)
{
    int i , j ;
    printf("the entered array is: ");
    for(i=0;i<n;i++)
    {
        for(j = 0; j < n; j++)
        { printf( " %d\t " , b[i][j]); }
        printf("\n");
    }
}
```

String

- String is an array of characters terminated by a null character('\0').
- It is a sequence of characters that is treated as a single data item.
- After the last character, a null character is stored to signify the end of the character array.

For e.g., if we write

```
char str[] = "HELLO";    //string initialization
```

- We are declaring a character array that has five usable characters namely, H,E,L,L and O.
- Apart from these characters, a null character('\0') is stored at the end of the string.
- So, the internal representation of the string becomes HELLO'\0', to store a string of length 5, we need 5+1 locations(1 extra for the null character)

String handling functions

There are various string handling functions defined under the header file <string.h>.
Some of them are as follows:

- **strlen():** it gives the number of characters excluding the null character in any string, which is the length of the string.

  Syntax: l = strlen(str); where 'l' is any integer variable and 'str' is any string variable
- **strcpy():** It copies the content of any string to another string similar to assigning any value to another variable .

  Syntax: strcpy(str1,str2); here the content of string 'str2' is copied to the string 'str1'.
- **strcat():** It concatenates any two strings i.e. joins any string to another string

  Syntax: strcat(str1,str2); here the content of string 'str2' is attached to the last of the string 'str1'.
- **strcmp():** It gives the difference between the ASCII values of first mismatching characters between any two strings.

  Syntax: d = strcmp(str1,str2); where 'd' is any integer variable. If d = 0, it means that both strings str1 and str2 are same or equal. If d>0, it means that str2 comes before str1 in alphabetical order and if d<0, it means that str1 comes before str2 in alphabetical order.

```
//  A program to calculate length of string without using strlen function
#include<stdio.h>
#include<conio.h>
void main()
{
        int  i, len = 0;
        char  str[20];
        printf( " Enter any string: " );
        gets( str );
        for( i = 0 ; str[i] != '\0' ; i++)
        {
                len++;
        }
        printf( "  The length of the string is %d. " , len );
}
```

OUTPUT:

```
Enter any string: Nepal
The length of the string is 5
```