UNIT 2

DECISION MAKING AND BRANCHING

Decision Making & Branching

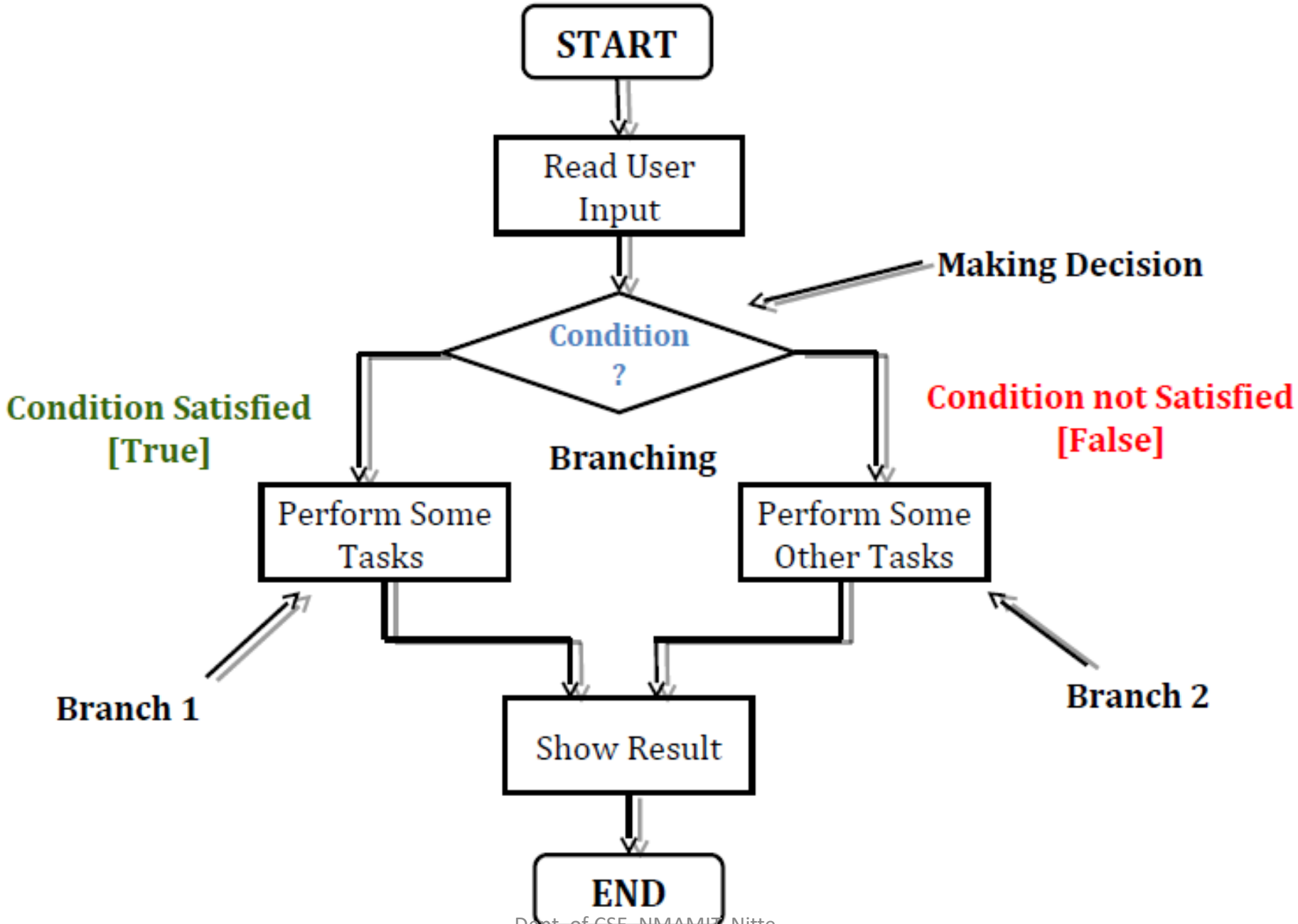
Why decision making and branching?

- Programs should be able to make logical (true/false) decisions based on the condition provided.

- Every program has one or few problems to solve. In order to solve those particular problems important decisions have to be made depending on the nature of the problems.

- Generally C program execute it's statements sequentially. But in order to solve problems we may have some situations where we have to change the order of executing the statements based on whether some conditions have met or not.

- So **controlling the execution of statements based on certain condition or decision is called decision making and branching.**


START
Read User Input
Making Decision
Condition ?
Condition Satisfied [True]
Condition not Satisfied [False]
Branching
Perform Some Tasks
Perform Some Other Tasks
Branch 1
Branch 2
Show Result
END

C provides following Control Statements / decision making statements:

(1) Conditional Branching Control Statements

(a) **if** statement

(b) **if-else** statement

(c) **nested if-else** statement

(d) **switch** statement

(2) Unconditional Branching Control Statement

(a) **goto** statement

if statement

- It is a **one way branching statement.**

Syntax:

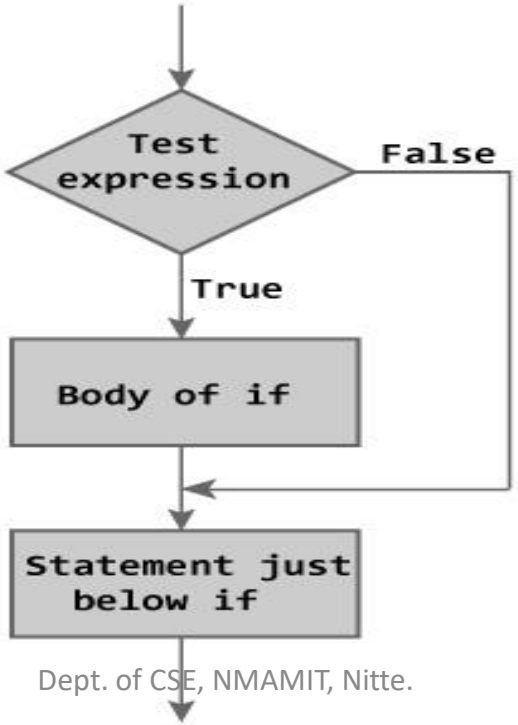
Single statement

```
if (testExpression)
{
     statement;  // body of if
}
statement –x;
```

Compound Statement

```
  if (testExpression)
{
  statement-1; // body of if
  statement-2;
  ……
  statement –n;
 }
  statement –x;
```

- The if statement evaluates the test expression inside the parenthesis.
- If the test expression is evaluated to true (nonzero), statements inside the body of if is executed.
- If the test expression is evaluated to false (0), statements inside the body of if is skipped from execution.



Example - if statement

Single statement usage:

```
if (marks>=90)
    marks=marks+ bonus_marks;
printf(“The mark achieved:marks” , %d);
```

Compound statement usage with braces:

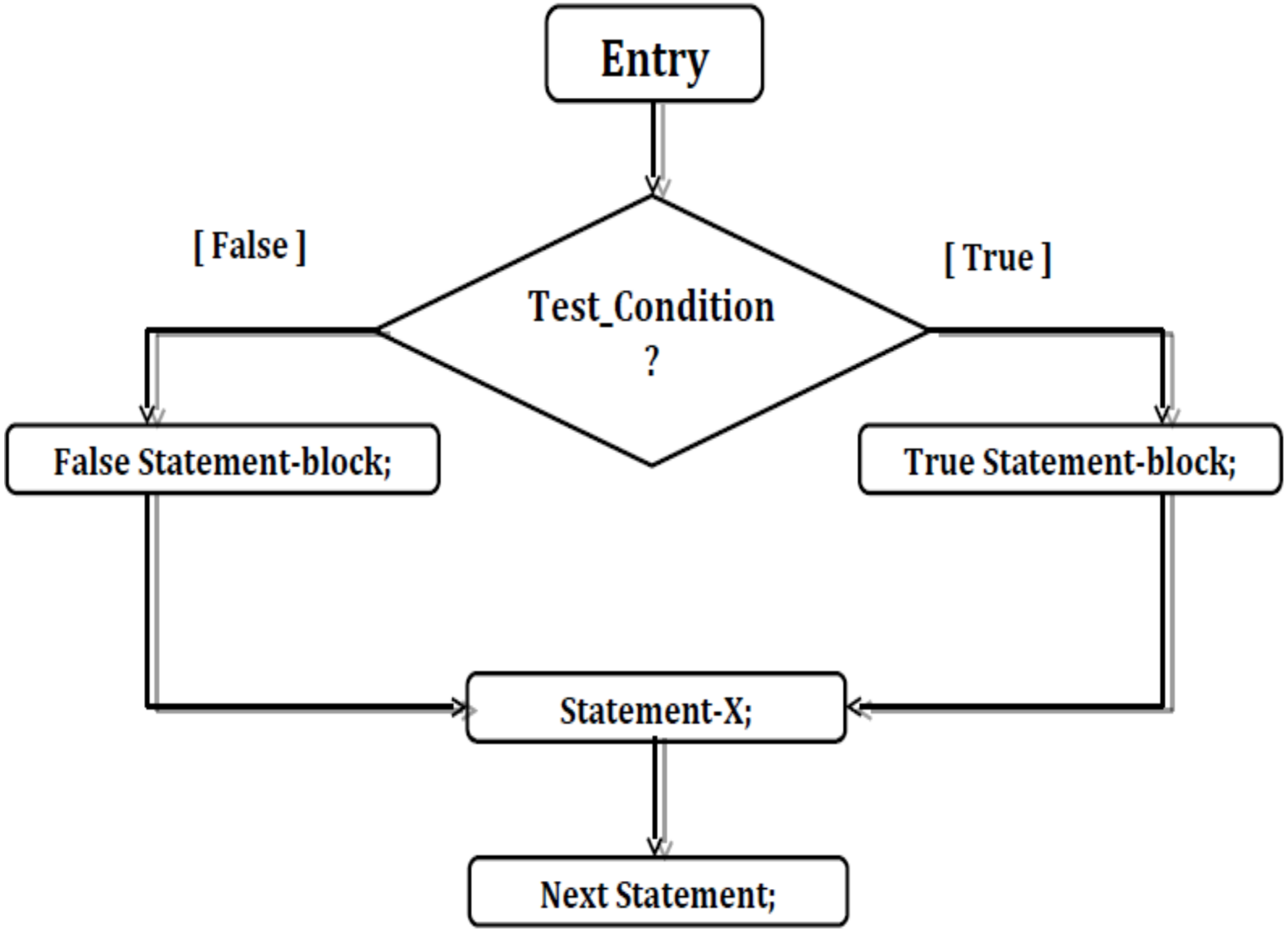
```
if (marks>=90)
{
  marks=marks+ bonus_marks;
  grade=“A+”;
}
printf(“The mark achieved:marks” , %d);
```

if-else statement

- if-else statement is an extension of the simple if statement.

Syntax:

```
if (test_condition)
{
    True block statements;
}
else
{
    False block statements;
}
statement-x;
```


Entry
[False]
Test_Condition
?
[True]
False Statement-block;
True Statement-block;
Statement-X;
Next Statement;

- If the test condition is true then the true block statements, immediately following the if statements are executed;
- Otherwise the false block statements are executed.

- i.e., either true-block or false-block of statements will be executed, not both.

- But in both cases the control is transferred subsequently to the statement-x as it is an independent (not controlled by the if-else statement) statement.

- **It is also called two way conditional branching**

Example - if-else

```
if (marks>=40)
{
   marks=marks+ bonus_marks;
   grade=“passed”;
}
else
{
    marks=marks;
    grade=“failed”;
}
printf(“The mark achieved:marks” , %d);
```

Nested if statements

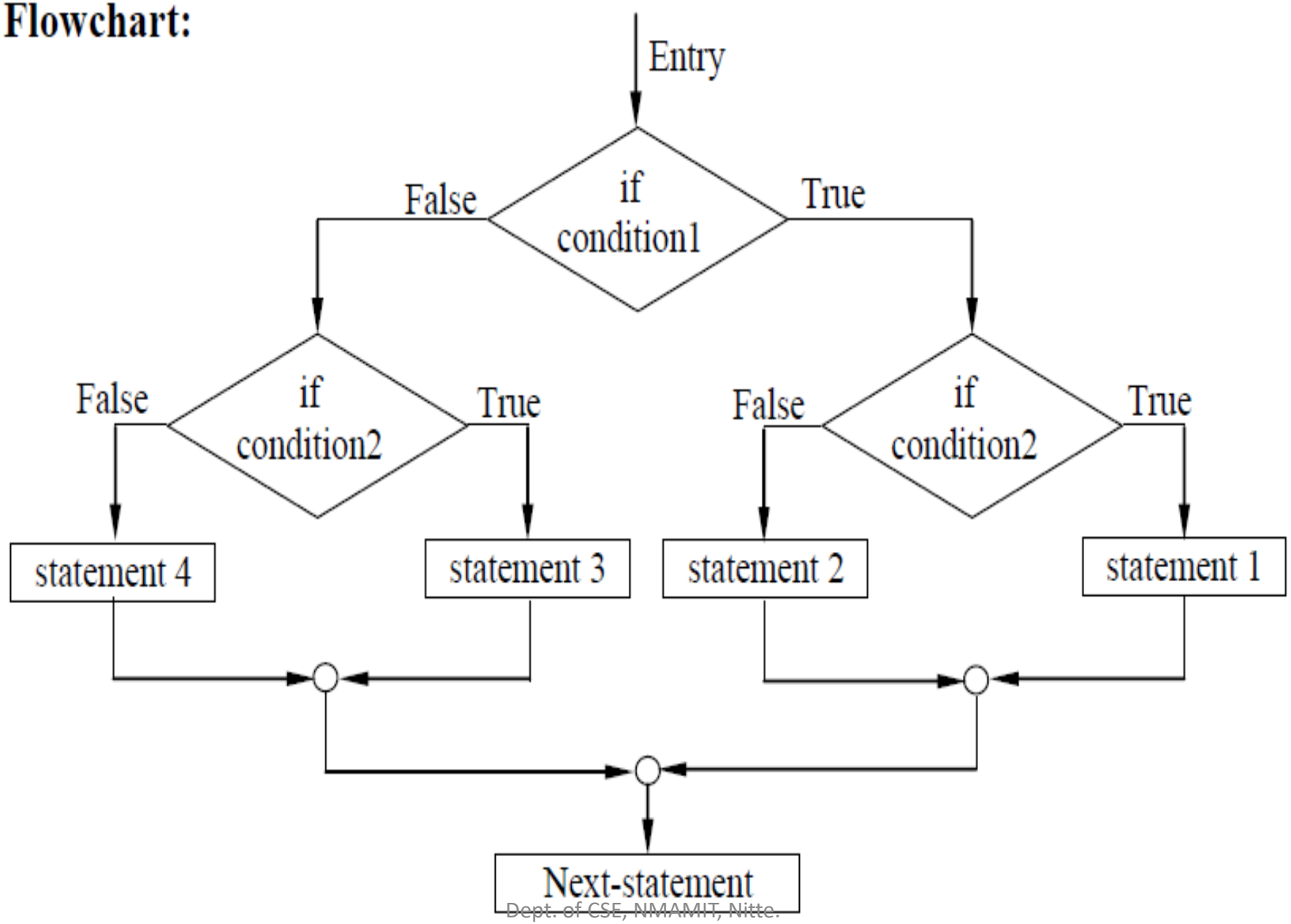
- Using “if…else statement” within another “if…else statement” is called ‘nested if statement’.

- “Nested if statements” is mainly used to test multiple conditions or used when a series of decisions are involved.

Syntax:

```
if (test_condition)
{
    if (test_condition)
    {
        statement-block;
    }
    else
    {
        statement-block;
    }
}
else
{
    statement-block;
}
```

Flowchart:


Entry
if condition1
False
True
if condition2
False
True
if condition2
False
True
statement 4
statement 3
statement 2
statement 1
Next-statement

Example - nested-if

```
if((s1+s2)>s3&&(s2+s3)>s1&&(s3+s1)>s2)
{
  printf("Given 3 sides form a triangle...");
  if(s1==s2 && s2==s3)
     printf("It is an Equilateral triangle");
  else if(s1==s2||s2==s3||s3==s1)
    printf("It is an isosceles triangle");
  else
   printf("It is a scalene triangle");
}
else
printf("Sorry...entered 3 sides do not form atriangle..."); }
```

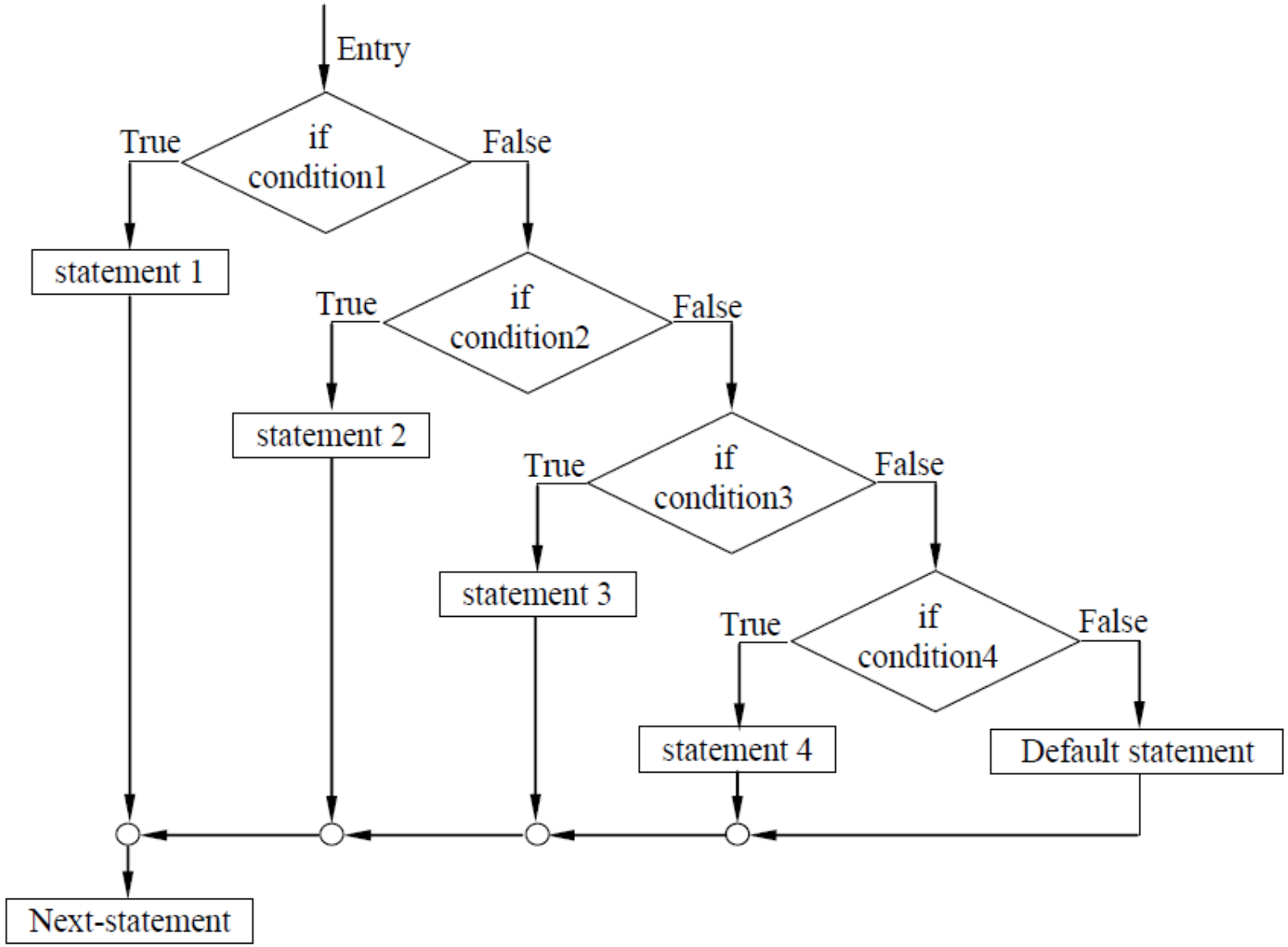
The else-if ladder

- A multiway decision is a chain of if conditions in which the statement associated with an else condition behaves like another if condition.

- Else if ladder is also called 3 way or multiway decision making statement.

Syntax:

```
if (test_condition 1)
    statement-1;
else if (test_condition 2)
     statement-2;
else if (test_condition 3)
     statement-3;
else if (test_condition 4)
      statement-4;
……..
else if (test_condition n)
       statement-n;
else
       default-statement
Next-statement;
```


Entry
if condition1
True
False
statement 1
if condition2
True
False
statement 2
if condition3
True
False
statement 3
if condition4
True
False
statement 4
Default statement
Next-statement

Example - else-if ladder

```
if(Mark>=50 && Mark<60)
{   printf("Your grade is D");
}
else if(Mark>=60 && Mark<70)
{   printf("Your grade is C n");
}
else if(Mark>=70 && Mark<80)
{    printf("Your grade is B n");
}
else if(Mark>=80 && Mark<90)
{     printf("Your grade is A n");
}
else
printf("you have failed");
```

Dangling Else Problem

It explains the association of single else statement in a nested if statement.

In nested if statements, when a single “else clause” occurs, the situation happens to be dangling else!

For example:

```
if (condition)
if (condition)
if (condition)
else
printf("dangling else!\n"); /* dangling else, as to which if statement, else clause associates */
```

1. In such situations, else clause belongs to the closest if statement which is incomplete that is the innermost if statement!
2. However, we can make else clause belong to desired if statement by enclosing all if statements in block outer to which if statement to associate the else clause.

For example:

```
if (condition)
{
if (condition)
if (condition)
}
else
printf("else associates with the outermost if statement!\n");
```

Switch Statement

- When one of the many statements is to be selected, then if conditional statement can be used to control the selection.

- However the complexity of such a program increases dramatically when the number of statements increases.

- For such scenarios, C has a built in multiway decision making statement known as switch.

- The switch statement tests the value of a given variable or expression against a list of case values and when a match is found only then a block of statements associated with that case is executed.

Syntax:

```
switch(expression/value)
{
   case value-1:
         statement-block-1;
         break;
   case value-2:
         statement-block-2;
         break;
……….
   case value-n:
         statement-block-n;
         break;
   default:
         default-statement-block;
          break;
}
statement-x;
```

Rules for Switch Statement

- The switch statement must be an integral type.
- Case labels must be constant or constant expression.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with colon.
- The break statement transfer the control out of the switch statement.
- The break statement is optional. So two or more case labels may belong to the same statements.
- The default label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one default label.
- The default may be placed any where but usually placed at the end.
- It is permitted to nest switch statements.

Example - switch statement

```
printf(“\n\nEnter i (1 to 3):”);
scanf(“%d”,&i);
switch(i)
{
 case 1: printf(“\n\nI am in case 1...”);
          break;
case 2: printf(“\n\nI am in case 2...”);
          break;
case 3: printf(“\n\nI am in case 3...”);
          break;
default: printf(“\n\nI am in default case...”);
}
```

goto statement

- The **goto** statement is an unconditional statement in C which allows the flow of execution of statements to be altered without the use of any conditional expression.

- The **goto** statement transfers the control from one point to another in a C program. This is also known as *Jumping of control* or *Unconditional Jumping*.

- The syntax of **goto** statement is as follows:

  goto label;

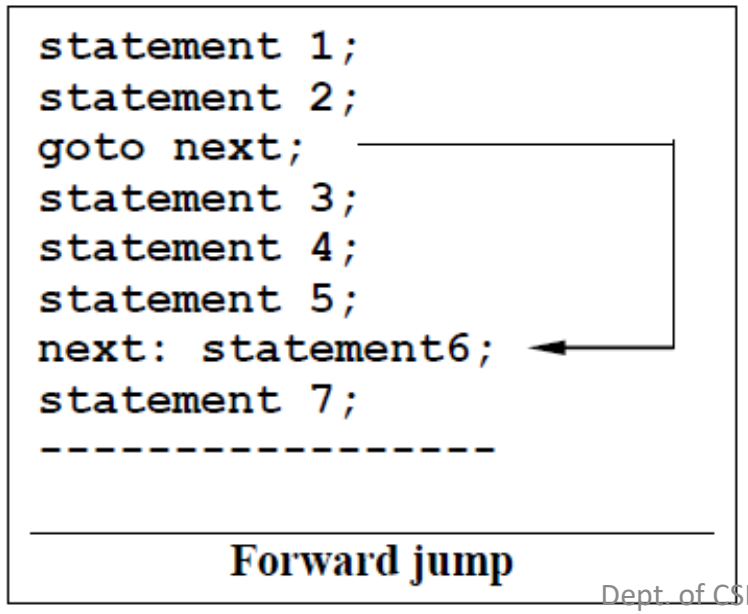
  where, **goto** is a keyword, and

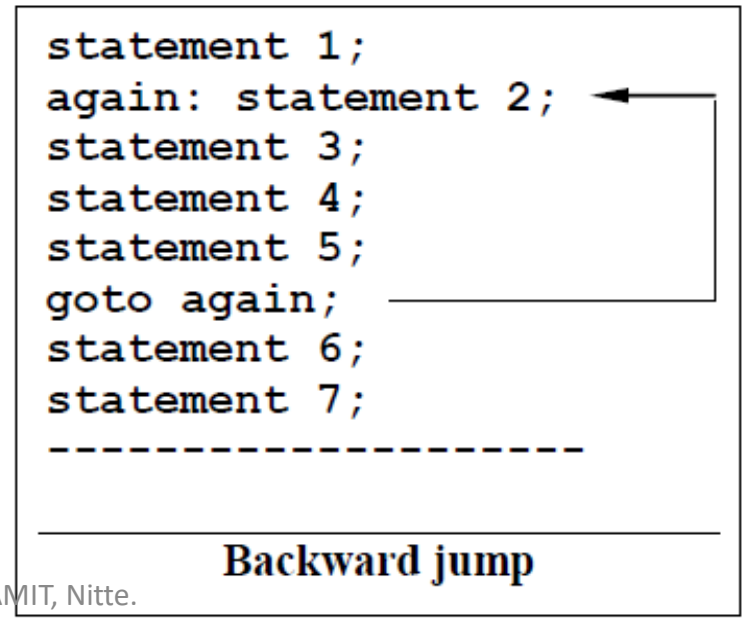
  label is a symbolic constant or an identifier which need not be pre-declared.

Two types of jumping (branching) is possible by using **goto** statement:
(1) Forward jump: Here the statements immediately following the **goto** statement will be skipped and control is transferred to the statement beginning with the symbolic constant **label**

.

(2) Backward jump: Here the symbolic constant **label** is placed before the **goto** statement and the statements between the **label** and **goto** statements are repeated resulting in looping.





Example – goto statement

```
printf(“\n\n Enter five numbers and find their sum:”);
read: scanf(“%f”,&num);
        sum += num;
        count ++;
        if (count < 5)
           goto read;
printf(“\n The sum of 5 numbers is %f”, sum);
}
```

Conditional Operator

The general structure of conditional operator:

Conditional expression? true-statement 1: false-statement;

Conditional operator:

flag = (x<0) ? 0 :1;

It is equivalent to:

```
if(x<0)
   flag=0;
else
   flag=1;
```

DECISION MAKING AND LOOPING

Looping

- Mechanism by means of which a set of statements are executed repeatedly for a fixed number of times or until a condition is fulfilled.

Two types: ***entry – controlled loop*** and ***exit – controlled loop***.

- In ***entry – controlled loop***, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed.

- In ***exit – controlled loop***, the test is performed at the end of the loop and therefore the body is executed unconditionally for the first time.

C language provides ***three*** types of loop structures.

- **while** statement.
- **do** statement.
- **for** statement.

The while statement

- It executes a set of statements repeatedly as long as the specified condition is true.
- Entry-controlled loop.

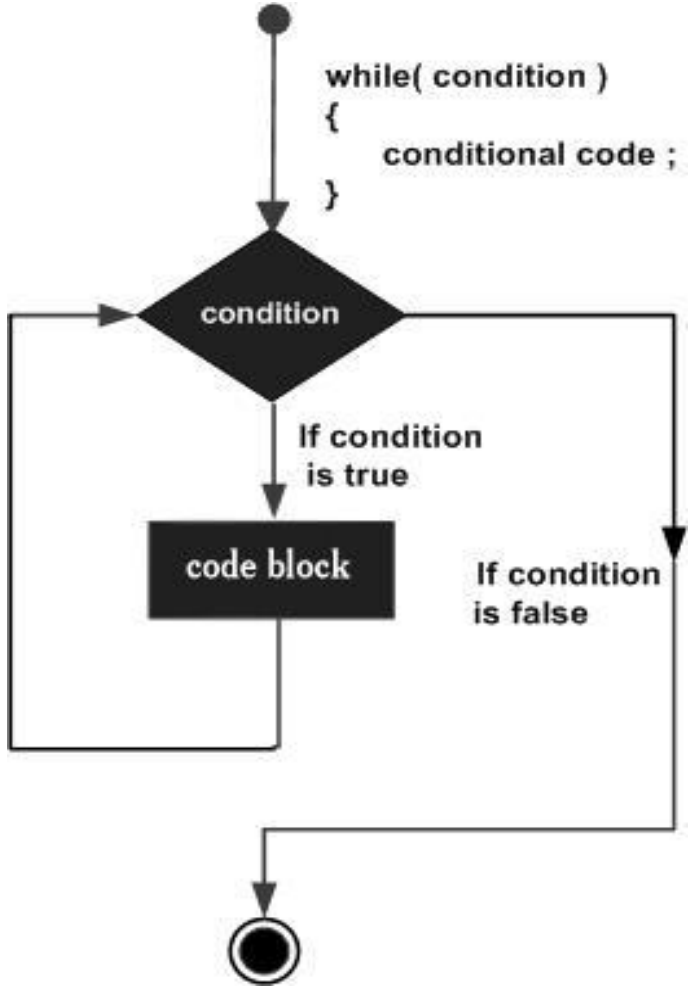
Syntax of while statement:

```
while (test_condition)
        statement;      // single statement
statement x;
statement y;
```

while is a keyword.

test_condition is a logical or relational expression that results in either TRUE or FALSE.

```
while (test_condition)
{
    statement1;          // compound statement
    statement2;
    statement3;
}
 Statement x;
 Statement y;
```



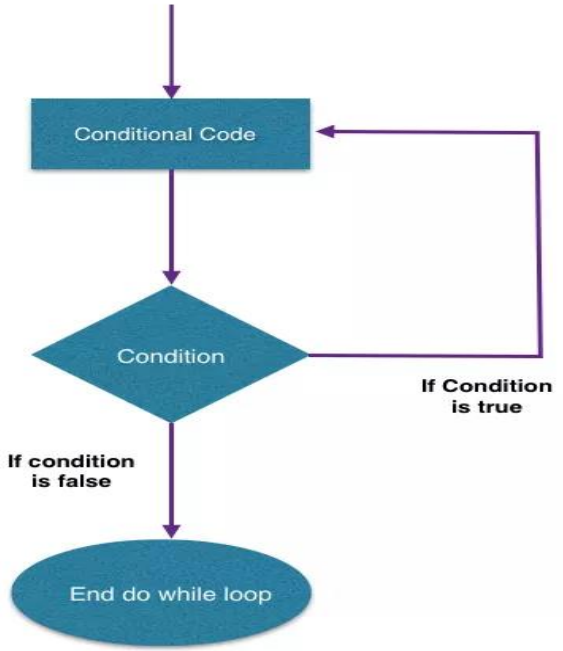
Example for while

```
int main ()
{
        int a = 10;
        while( a < 20 )
        {
                printf("value of a: %d\n", a);
                a++;
        }
 return 0;
}
```

do..while statement

Syntax of do..while statement:

```
do
{
        statement1;
         statement2;
} while (test_condition);
Statement x;
Statement y;
```



- **do** and **while** are keywords.
- **test_condition** is a logical or relational expression that results in either TRUE or FALSE.

- On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test_condition in the **while** statement is evaluated.

- If the condition is *true*, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true.

- When the condition becomes *false*, the *loop* will be terminated and the control goes to the statement that appears immediately after the while statement.

- Since the test_condition is evaluated at the bottom of the loop, the **do...while** construct provides an ***exit controlled*** loop and therefore the body of the loop is a*lways executed at least once*.

Example for do..while

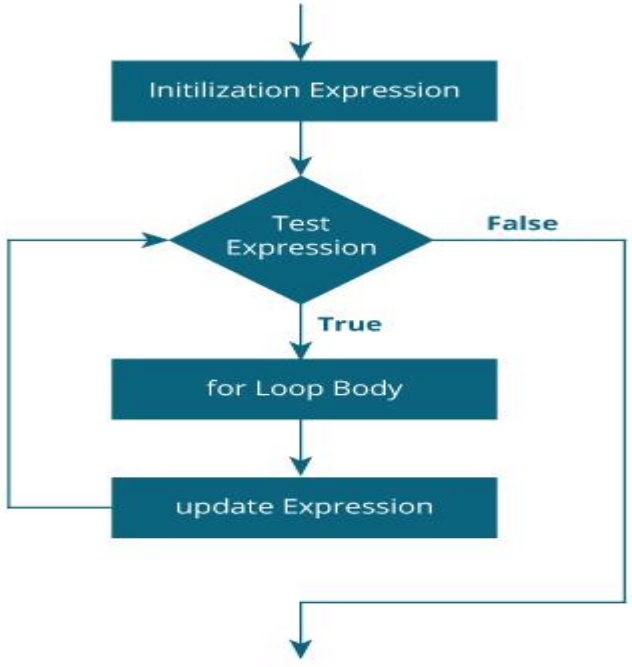
```
main()
{
int odnum, sum = 0;
odnum = 1;
do
{
        sum += odnum;
        odnum += 2;
} while(odnum <= 50);
printf("Sum =%d\n",sum);
}
```

for statement

- **for** is an *entry – controlled loop* statement.
- This statement is used when the programmer knows how many times a set of statements are executed.

Syntax of for statement:

```
for (initialization; test_condition; increment)
{
        statement1;
        statement2;
}
Statement x;
Statement y;
```



- **Initialisation** of the *control variable* is done first. Using assignment statements such as i = 1 and count = 0. The variables *i* and *count* are known as *loop – control variables*.

- The value of the *control variable* is tested using the **test_condition**. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement the immediately follows the loop.

- When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the *control variable* is **incremented** using an assignment statement such as i = i+1 and the new value of the *control variable* is again tested to see whether it satisfies the loop condition.
- If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test_condition.

for statement example

```
 int main()
{
      int num, count, sum = 0;
      printf("Enter a positive integer: ");
      scanf("%d", &num);
      // for loop terminates when n is less than count
      for(count = 1; count <= num; ++count)
      {
          sum += count;
       }
printf("Sum = %d", sum);
return 0;
 }
```

Jumps in Loops

- When executing the body of loop, sometimes it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.

- *Example, in the case of searching for a particular name in a list containing, say, 100 names, program loop must be terminated as soon as the desired name is found.*

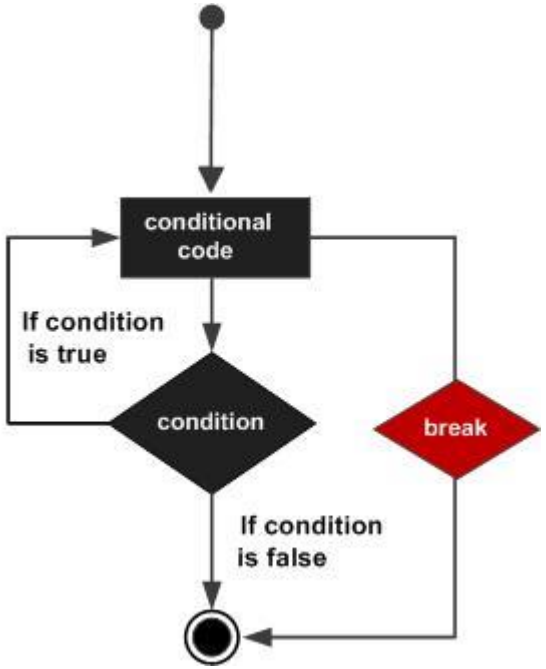
Jumping Out of a loop:

- Early exit from a loop can be accomplished by using the **break** statement or the **goto** statement.

- **break** statement or the **goto** statement can be used within if …else, switch, while, do …while, or for loops.

- When the **break** statement is encountered *inside the loop*, the loop is immediately exited and program continues with the statement immediately following the loop.

- When the loops are *nested*, the **break** would only exit from the loop containing it. That is **break** will *exit only a single loop*.



break statement example

```
int main()
{
    int num=0;
    while(num<=100)
    {
        printf("value of variable num is: %d\n", num);
        if (num==2)
        {
            break;
        }
        num++;
    }
    printf("Out of while-loop");
}
```

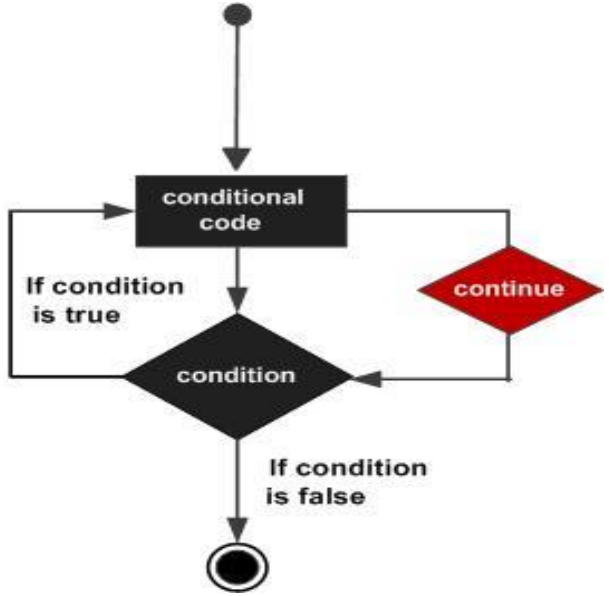
Skipping a part of a loop:

- To skip certain statements in the loop and to continue with the next iteration of the loop **continue** statement is used.

```
while (test_condition1)
{
    statement1;
    if (test_condition2)
        continue;   // back to while when test_condition 2 is true
    statement2;
    statement3;
}
```

continue statement example

```
int main()
 {
    for (int j=0; j<=8; j++)
    {
        if (j==4)
        {
            continue;
             /* The continue statement is encounterec
             value of j is equal to 4. */
            }
            /* This print statement would not execute for the loop
               iteration where j ==4 because in that case this
               statement would be skipped. */
            printf("%d ", j);
       }
}
```



• Arrays and String

Arrays

Arrays (1-D, 2-D) Initialization and Declaration.

Examples for one dimensional and two dimensional arrays.

Strings

Declaring, Initializing, Printing and reading strings,

string manipulation functions,

String input and output functions.

Learning Objectives

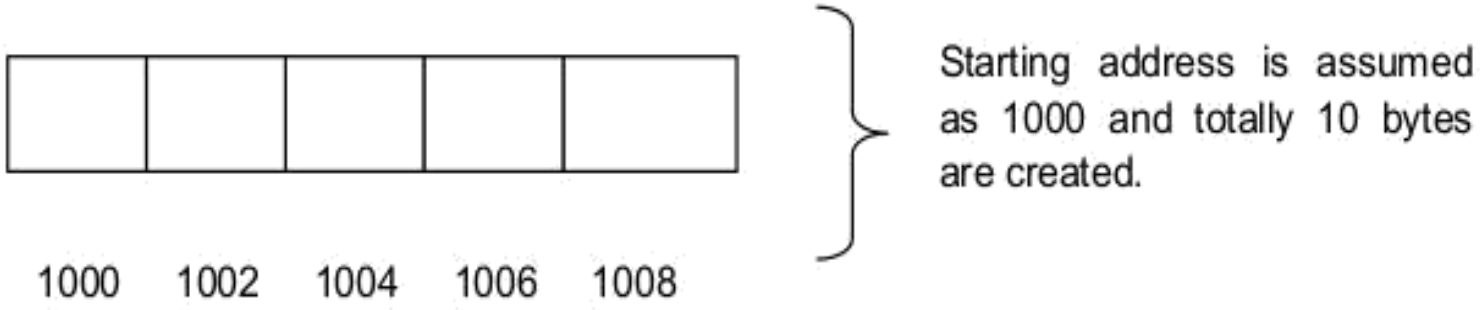
After completing this chapter, you will be able to:

- Understand the concept of array and its memory organization
- Know how arrays are declared and initialized in C
- Use multi-dimensional arrays
- Know about various character and string functions.

- What is the Need for an Array?
- Types of Arrays
  - One dimensional array
  - Multi dimensional arrays (2-D, 3-D, etc.)

Memory Organization of an Array

The elements in an array are always stored in consecutive memory locations. If an array of 5 integers elements is created, totally 10 contiguous bytes will be allocated in memory. Note: size of an integer is 2 bytes.



Declaration of Array

General Form:

datatype arrayname[size] ;

This is called a single-dimensional array.

For example, to declare a 10-element array called **balance** of type integer

int balance[10];

Here balance is a variable array which is sufficient to hold up to 10 integer numbers. Arrays are defined by appending an integer encapsulated in square brackets at the end of a variable name.

Some more examples

int x[5]; Defines an integer array x of 5 integers, starting at x[0], and ending at x[4].

char str[16]="qwerty"; Defines a character array, which represents a string of maximum of 16 characters.

float sales_amt[10];

Defines a floating point array sales_amt of 10 floating point numbers, starting at sales_amt[0] and ending at sales_amt[9].

int matrix[2][2]; Defines a 2*2 matrix (totally 4 elements) of integers.

Initializing Arrays

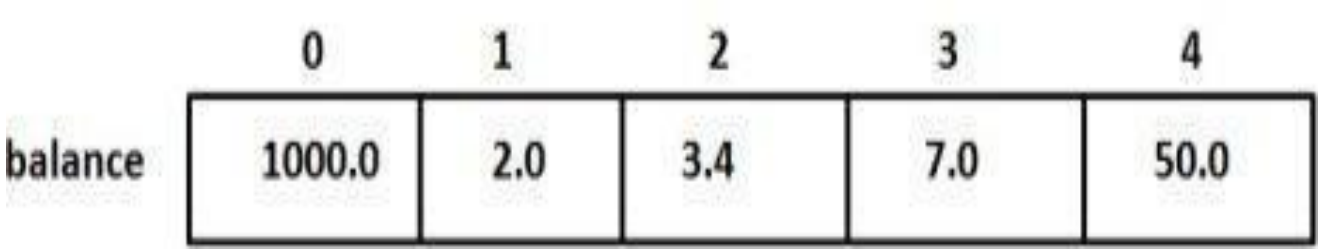
General Forms:

datatype arrayname[size] = {value(s)};

datatype arrayname[ ] = {value(s)};

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

or double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};



Some more examples

```
int a[5]={1,2,3,4,5};          /*a[0] = 1, a[1] = 2 , a[2] = 3 , a[3] = 4 and a[4] = 5*/
int a[5]={0};                  /*all the array elements are initialized to zero*/
int a[5]={1,2,3,4};            /*a[4] = 0*/
                               /*a[0]=1, a[1]=2, a[2]=3, a[3]=4 (if size not specified, size
int a[ ] = {1,2,3,4};          depends
                               upon the number of values initialized. ) */
float b[2]={10.2,45.34};         /* b[0] = 10.20 , b[1] = 45.34 */
```

Accessing Array Elements

The array elements are accessed by specifying the subscript or index.

General Form:

arrayname[index or subscript]

Example:

x[0] to access the 1st element in array

x[4] to access the 5th element in array

str[2] to access the 3rd character in the string (character array)

sales_amt [8] to access the 9th sales amount in the array

Basic Operations on Arrays

Basic operations allowed on arrays are storing, retrieving, and processing of array elements.

Reading the value for Arrays using scanf function:

Input statement is used to get the values for an array.

Example:

```
        int a[3];
(1)     scanf("%d", &a[0]);        /*gets value for 1st location*/
        scanf("%d", &a[1]);        /* gets value for 2nd location*/
        scanf("%d", &a[2]);        /* gets value for 3rd location*/
        scanf("%d%d%d", a, a+1,    /* gets value for first 3 locations (array
        a+2);                      name
                                       has the base address - pointer)*/

        for(i=0;i<3;i++)           /* loop statement is used to get the array
(2)       scanf("%d",&a[i]);           elements */
```

Printing out the array elements

Example:

```
                int a[3];

(1)             printf(“%d”, a[0]);                 /*prints value of 1st location*/

                printf(“%d”,a[1]);                  /*prints value of 2nd location*/

                printf(“%d”, a[2]);                 /*prints value of 3rd location*/

(2)             printf(“%d%d%d”,a[0],a[1],a[2]);    /* prints value of first 3 locations*/


                  for (i=0;i<3;i++)
(3)             printf(“%d”,a[i]);      /*loop statement is used to print the array elements */
```

Multi-dimensional Array

- The elements of an array can themselves be arrays.
- Two dimensional arrays can be viewed as group of one dimensional array (rows & columns) and 3 dimensional arrays can be viewed as set of two dimensional arrays and so on.
- Multidimensional arrays will also occupy the contiguous memory locations.

Two-dimensional array – Declaration

Two-dimensional arrays are defined in the same way as one dimensional array, except that a separate pair of square brackets are required for second dimension.

General Form:

datatype arrayname[row][column];

Example:

int a[2][2]; creates 8 bytes of contiguous memory locations. (2*2 = 4 elements).

Two-dimensional array Initialization

Two-dimensional arrays can also be initialized in the declaration statement. In partial initialization, the uninitialized array elements are initialized to Zero.

```
int   num[2][3] = {1,2,3,4,5,6};
int   num[2][3] = {1,2,3,4,5};              /*num[1][2] = 0*/
int   num[2][3] = {{1,2,3},{1,2,3}};        /*row elements are initialized separately*/
int   num[2][3] = {{1,2},{4}};              /*num[0][2] = 0 num[1][1]=num[1][2]=0*/
```

Example for 4-dimensional array:

sales [year] [month] [area] [salesperson]

Advantages of arrays:

- Simple and easy to use.
- Stored in Contiguous locations.
- Fast retrieval because of its indexed nature.
- No need to worry about the allocation and de-allocation of arrays.

Limitations of arrays:

- Conventional arrays are static in nature.
- Memory is allocated in the beginning of the execution.
- If m elements are needed, out of n locations defined, n-m locations are unnecessarily wasted.
- No automatic array bounds checking during compilation.

➢ Programs on Arrays:

- Write a C program to transpose a matrix of order M x N and find the trace of the resultant matrix.
- Write a C program to input N real numbers in 1-D array. Compute mean, variance and Standard Deviation. Mean= sum/N, Variance = Σ (Xi-mean) 2 /N, STD Deviation= √variance.
- Write a C program to perform a linear search for a given key integer in a single dimensional array of numbers and report success or failure in the form of a suitable message using functions.
- Write a C program to find smallest and largest among the elements of matrix of order M x N.
- Write a C program using functions readmat ( ), rowsum ( ), colsum ( ), totsum ( ) and printmat( ) to read the values into a two dimensional array A, find the sum of all the elements of a row, sum of all the elements of a column, find the total sum of all the elements of the two dimensional array A and print the results.

Strings

Strings are sequence of characters.

In C, there is no built-in data type for strings.

String can be represented as a **one-dimensional array of characters**.

String should always have a **NULL** character (‘\0’) at the end, to represent **the end of string.**

String constants can be assigned to character array variables.

String constants are always enclosed within double quotes and character constants are enclosed within single quotes.

Declaration and initialization of string variables:

The following declaration and initialization create a string consisting of the word "Hello".

```
char str[6] = {'H','e','l','l','o','\0'};
```

Other examples:

- char c[4]={‘s’,’u’,’m’,’\0’);
- char str[16]="qwerty"; Creates a string. The value at str[5] is the character ‘y’. The value at str[6] is the null character. The values from str[7] to str[15] are undefined.
- char name[5]; int main( )

```
 {
        name[0] = ‘G’;
        name[1] = ‘O’;
        name[2] = ‘O’;
        name[3] = ‘D’;
        name[4] = ‘\0’;
        return 0;
 }
```

- char name[5] = “INDIA”; - Strings are terminated by the null character, it is preferred to allocate one extra space to store null terminator.

Explain following Examples:

- char name[20]; int i=0;
- while((name[i] = getchar ()) != ‘\n’)
  i++;
- scanf(“%s“ , name);
- printf(“%s” , name);

String Manipulation Functions

C does not provide any operator, which manipulates the entire string at once.

Strings are manipulated either using pointers or special routines (built-in) available from the standard string library **string.h.**

The following is the list of string functions available in string.h:

strcpy(string1, string2)	Copy string2 into string1
strcat(string1, string2)	Concatenate string2 onto the end of string1
strcmp(string1,string2)	Lexically compares the two input strings (ASCII comparison) returns 0 if string1 is equal to string2 < 0 if string1 is less than string2 > 0 if string1 is greater than string2

- strlen (string) Gives the length of a string
- strrev (string) Reverse the string and result is stored in same string.
- strncat(string1, string2, n) Append n characters from string2 to string1
- strncmp(string1, string2, n) Compare first n characters of two strings.
- strncpy(string1,string2, n) Copy first n characters of string2 to string1
- strupr (string) Converts string to uppercase
- strlwr (string) Converts a string to lowercase
- atoi (string) Converts the string to integer number
- atof (string) Converts the string to floating point number
- atol (string) Converts the string to long integer number
- strchr (string, c) Find first occurrence of character c in string.
- strrchr (string, c) Find last occurrence of character c in string.

String input and output functions

String input/read functions: scanf(), getchar() and gets.

String output/write functions printf(),putchar() and puts.

Using scanf and printf function:

The scanf() function can be used to read a string like any other data types.

Printf function used to print or display a string on display screen.

Format:

scanf(<string format specifier>, string_variable/s);

printf(<string format specifier>, string_variable/s);

Here <string format specifier> is %s used within the "".

Example: char c[20];

Here c represents array of 20 characters and it is also called as string variable.

```
scanf("%s", c);
printf("%",c);
```

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);
    //Function to read string from user.
    printf("Name: ");
    puts(name);
    //Function to display string.
    return 0;
}
```

Output

```
    Enter name: Tom Hanks
    Name: Tom Hanks
```

Home work:

- Reverse a given string.
- Compare two given string.
- Write a C program to find if a given string is a palindrome or not.