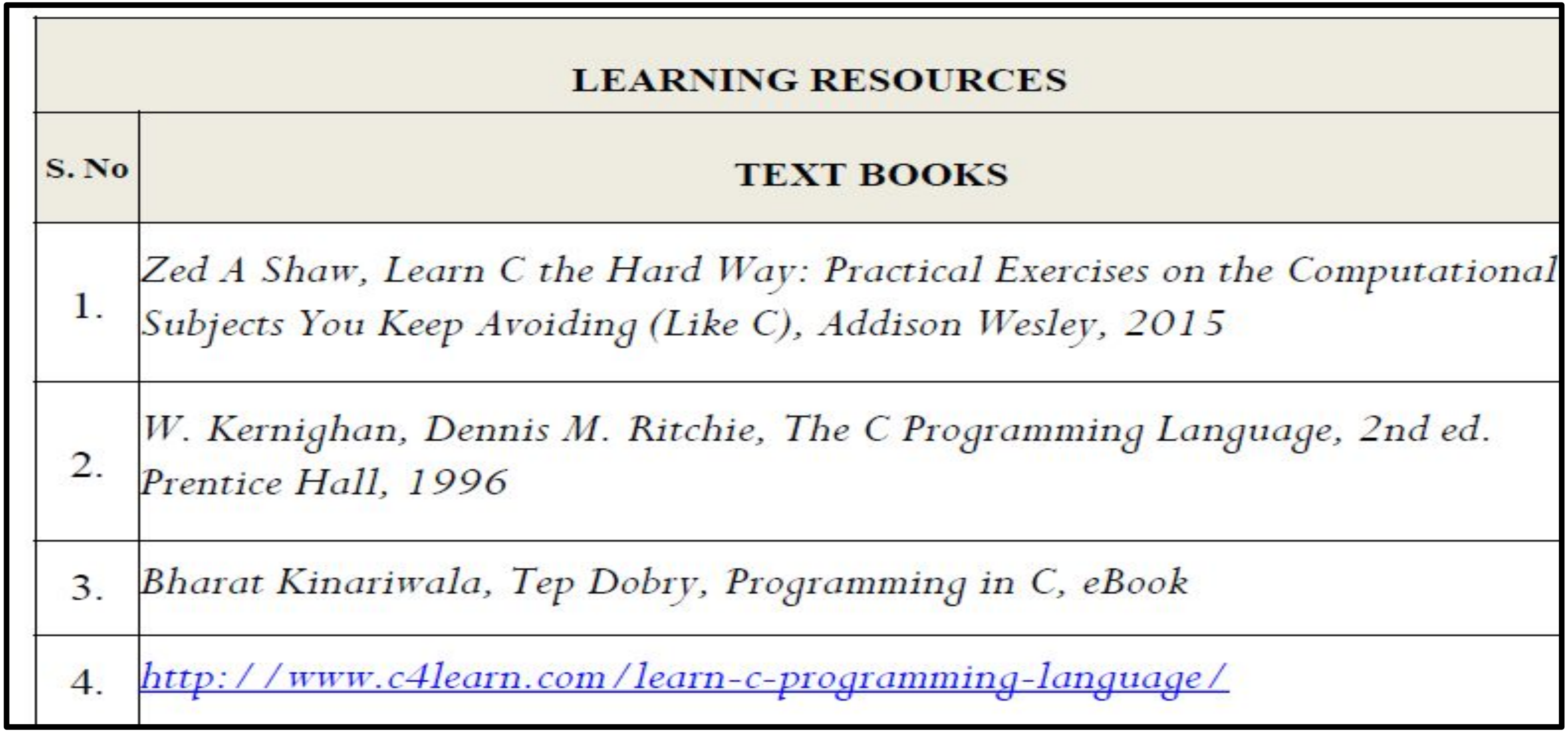
SRM Institute of Science and Technology, Chennai

21CSS101J –Programming for Problem Solving
Unit-II

SRM Institute of Science and Technology, Chennai

Prepared by:
Dr. P. Robert
Assistant Professor
Department of Computing Technologies
SRMIST-KTR

SRM Institute of Science and Technology, Chennai



LEARNING RESOURCES	
S. No	TEXT BOOKS
1.	*Zed A Shaw, Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C), Addison Wesley, 2015*
2.	*W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2nd ed. Prentice Hall, 1996*
3.	*Bharat Kinariwala, Tep Dobry, Programming in C, eBook*
4.	*http://www.c4learn.com/learn-c-programming-language/*

Unit-II

Conditional Control -Statements :Simple if, if...else - Conditional Statements : else if and nested if - Conditional Statements : Switch case - Un-conditional Control Statements : break, continue, goto - Looping Control Statements: for, while, do.while - Looping Control Statements: nested for, nested while - Introduction to Arrays -One Dimensional (1D) Array Declaration and initialization - Accessing, Indexing and operations with 1D Arrays - Array Programs – 1D - Initializing and Accessing 2D Array, Array Programs – 2D - Pointer and address-of operators -Pointer Declaration and dereferencing, Void Pointers, Null pointers ,Pointer based Array manipulation

Conditional Control Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (**if-then, if-then-else, switch**), the looping statements (**for, while, do-while**), and the branching statements (**break, continue, goto**) supported by the Java programming language.

Selection Statement	Iteration Statement	Jumping Statement
if	while	break
if-else	for	continue
switch	do-while	goto

Conditional Control Statements

Selection Statement

Selection statements are also referred to as decision making statements, branching statements, and conditional control statements. The selection statements are used to choose a portion of the program to run based on a condition.

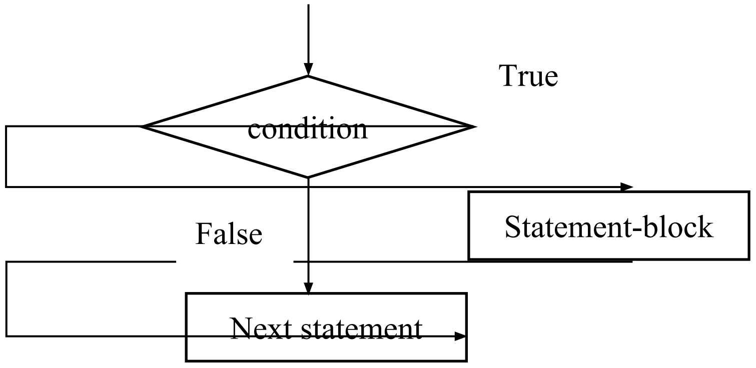
Simple if Statement

The if statement is used when you want a specific statement to be executed if a given condition is true. A condition is any expression that returns a boolean value, such as true or false (1 or 0)

Syntax

```
if` (condition)

{

Statement-block;

}

Next statement;
```

Flow of execution



Conditional Control Statements

Example:// simple if

```
#include<stdio.h>
void main()
{
int x=10;
if(x<20)
{
printf("Statements inside if executed\n");
}
printf("Statements outside if executed");
}
```

Output:

```
Statements inside if executed
Statements outside if executed
```

Conditional Control Statements

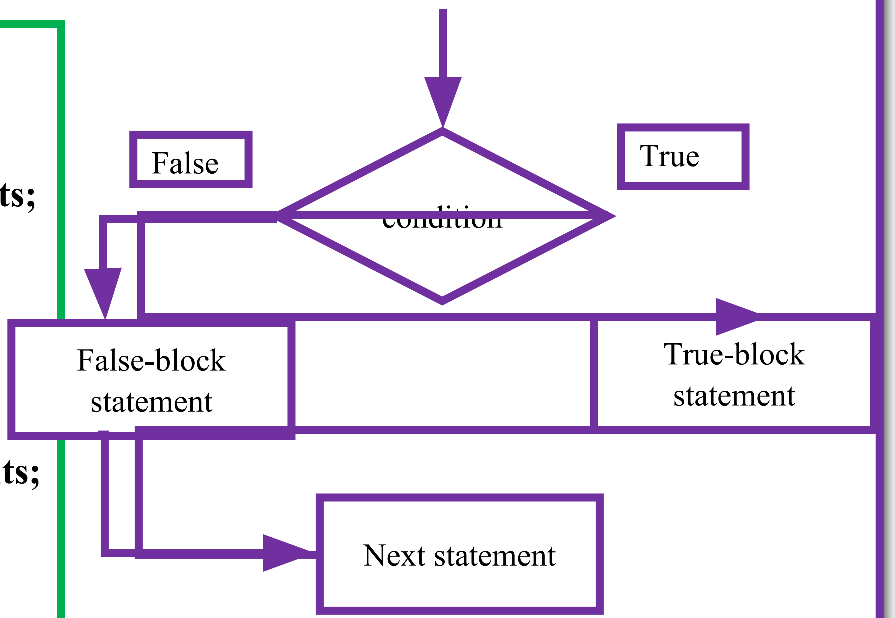
if..else Statement

If the condition is True, the true block of statements is executed; if the condition is False, the false block of statements is executed.

Syntax

```
if (condition)
{
True block statements;
}
else
{
False block statements;
}
```

Flow of execution



Conditional Control Statements

Example:// if..else

```
#include<stdio.h>
void main()
{
int age=21;
if(age>18)
{
printf("Eligible to Vote");
}
else
{
printf("Not eligible to Vote");
}
}
```

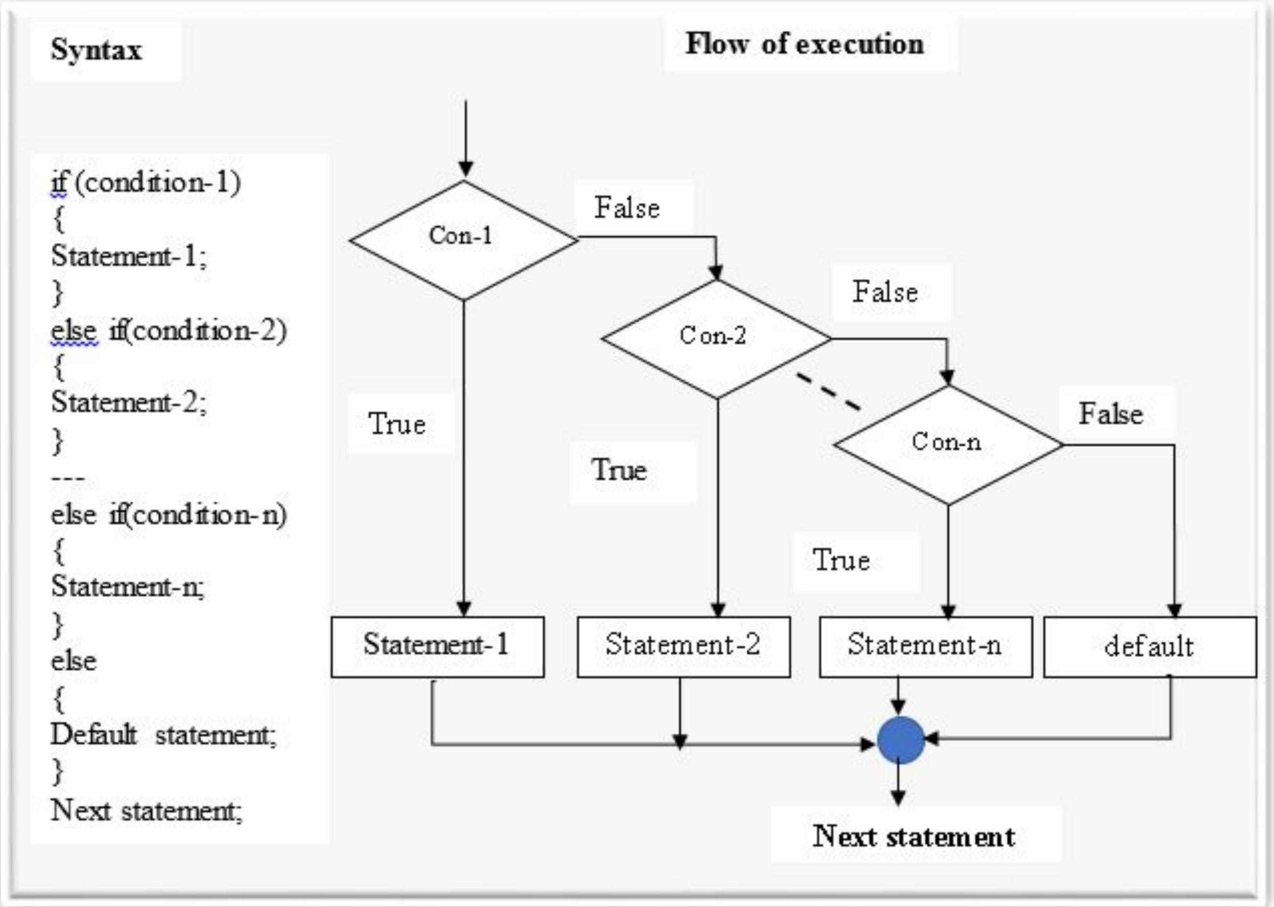
Output:

```
Eligible to Vote
```

Conditional Control Statements

else if

A cascading if-else statement is a collection of if-else statements in which the outer statement's false path is a nested if-else statement. The nesting process can go on for several levels.



Syntax

```
if (condition-1)
{
Statement-1;
}
else if(condition-2)
{
Statement-2;
}
---
else if(condition-n)
{
Statement-n;
}
else
{
Default  statement;
}
Next statement;
```

Flow of execution

Conditional Control Statements

Example:// simple if

```
#include<stdio.h>
void main()
{
int m;
printf("Input m value");//Month
scanf("%d",&m);
if(m==1 || m==2 || m==12)
{
printf("Winter");
}
else if(m==3 || m==4 || m==5)
{
printf("Spring");
}
else if(m==6 || m==7 || m==8)
{
printf("Summer");
}
```

Conditional Control Statements

```
else if(m==9 || m==10 || m==11)
{
printf("Autumn");
}
else
{
printf("Invalid Month");
}
}
```

Output:

```
Input m value5
Spring
```

Conditional Control Statements

Nested if

Nested If in C Programming is placing If Statement inside another IF Statement. Nested If in C is helpful if you want to check the condition inside a condition.

Syntax:

```
if( boolean_expression 1) {

   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2) {
      /* Executes when the boolean expression 2 is true */
   }
}
```

Conditional Control Statements

Example:

```
#include <stdio.h>
int main () {
  int a = 100;
  int b = 200;
  if( a == 100 ) {
      if( b == 200 ) {
      printf("Value of a is 100 and b is 200\n" );
    }
  }
    printf("Exact value of a is : %d\n", a );
  printf("Exact value of b is : %d\n", b );
  return 0;
}
```

Output:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Conditional Statements

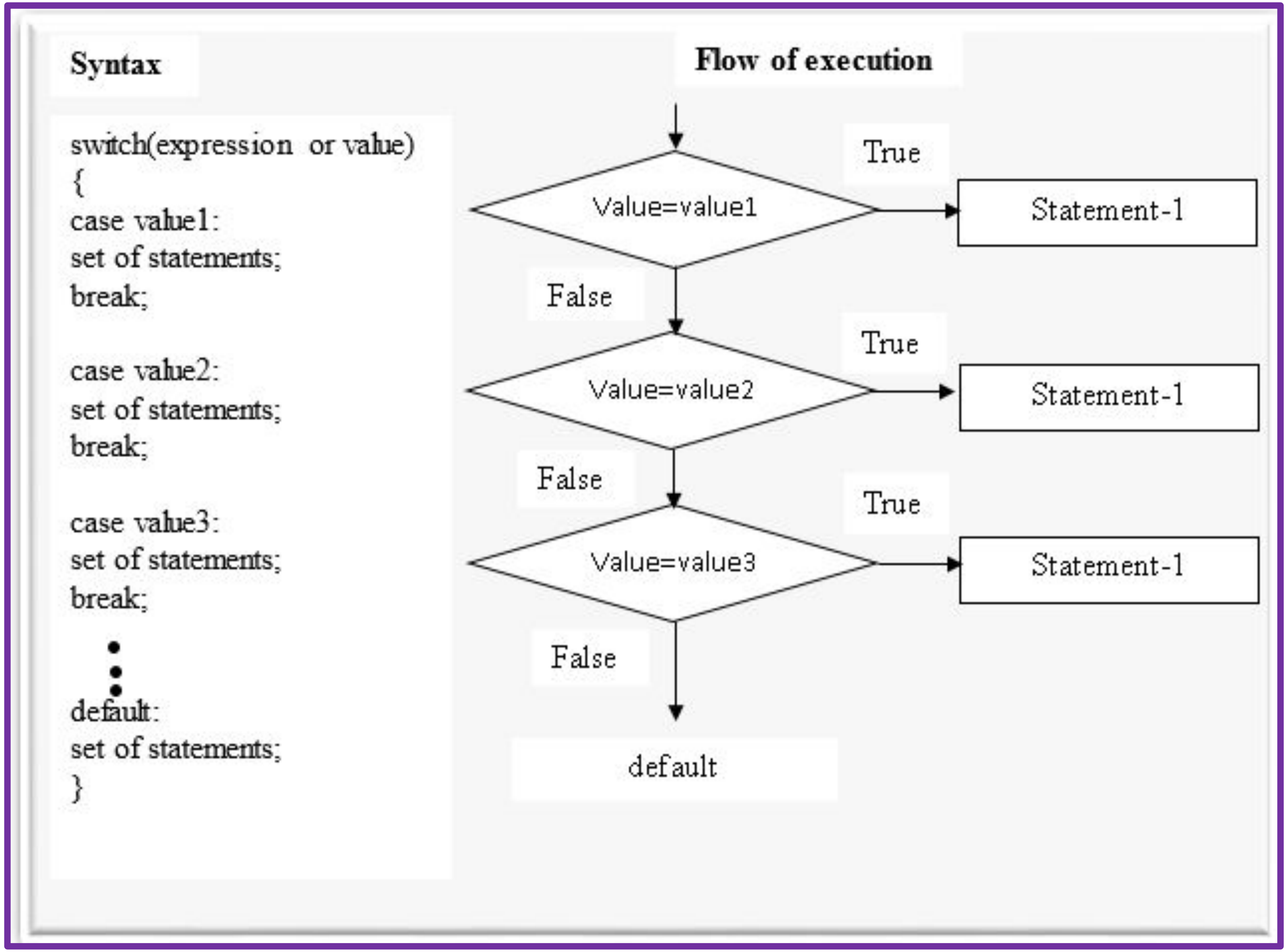
Switch case

The switch statement is an example of a multiway branch statement. Using the switch statement, it is very simple to select only one option from a large number of options. We provide a value to be compared with a value associated with each option in the switch statement. When the given value matches the value associated with an option, the option is selected.

Notes

- ✔ If we do not use the break statement, all statements after the matching label are also executed.
- ✔ The default clause inside the switch statement is optional.

Conditional Statements



Syntax

```
switch(expression  or value)
{
case value1:
set of statements;
break;

case value2:
set of statements;
break;

case value3:
set of statements;
break;
   •
   •
   •
default:
set of statements;
}
```

Flow of execution

Conditional Control Statements

Example: // Switch case

```
#include<stdio.h>
int main()
{
int day;
printf("Enter the day\n");
scanf("%d",&day);
switch(day)
{
case 1:
printf("Sunday");
break;
case 2:
printf("Monday");
break;
case 3:
printf("Tuesday");
break;
```

Conditional Control Statements

```
case 4:
printf("Wednesday");
break;
case 5:
printf("Thursday");
break;
case 6:
printf("Friday");
break;
case 7:
printf("Saturday");
break;
default:
printf("Invalid day");
}
}
```

Output:

```
Enter the day
1
Sunday
```

Un-Conditional Control Statements

break

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

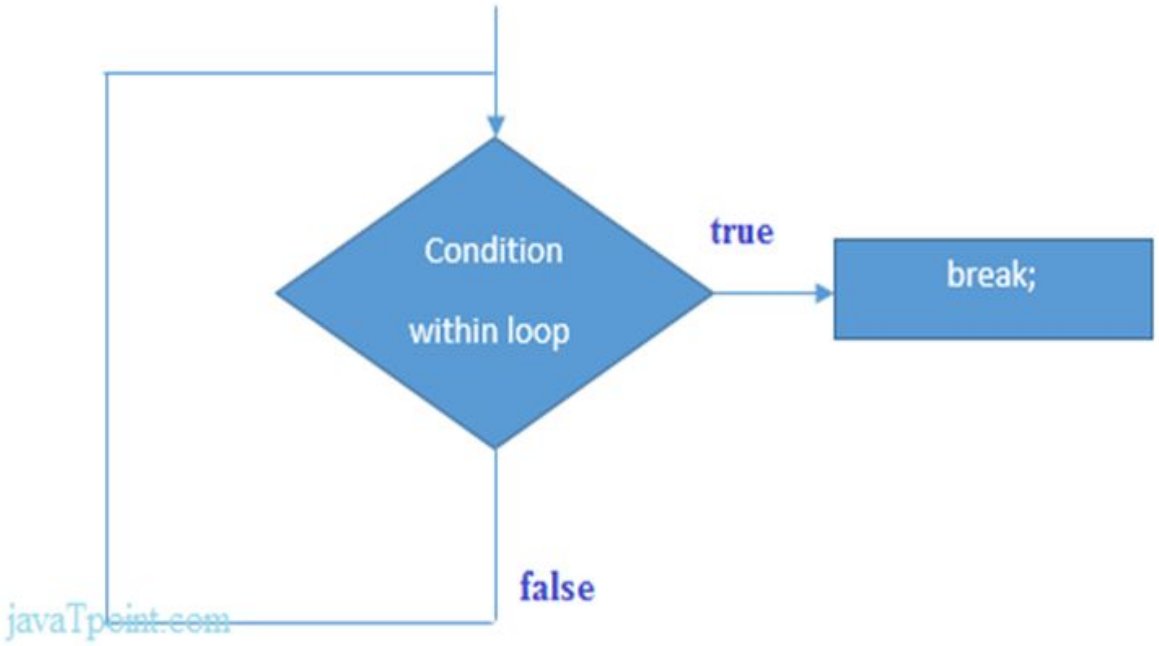


Figure: Flowchart of break statement

Conditional Control Statements

Example-1: // break

```
#include<stdio.h>
void main()
{
for(int i=1;i<=5;i++)
        {
            if(i==2)
            break;
            printf("%d\n",i);
        }
}
```

Output:

```
1
```

Conditional Control Statements

Example-2: // break

```
#include<stdio.h>
void main()
{
for(int i=1;i<=5;i++)
{
    if(i==3)
    {
        break;
    }
printf("%d\n",i);
}
}
```

Output:

```
1
2
```

Un-Conditional Control Statements

continue

The C *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.

Example:

```
#include<stdio.h>
void main()
{
for(int i=1;i<=5;i++)
{
   if(i==3)
   {
      continue;
   }
printf("%d\n",i);
}
}
```

Output:

```
1
2
4
5
```

Un-Conditional Control Statements

goto

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statment can be used to repeat some part of the code for a particular condition.

Syntax:

```
goto label;
... .. ...
... .. ...
label:
statement;
```

The label is an identifier. When the goto statement is encountered, the control of the program jumps to label: and starts executing the code.

Un-Conditional Control Statements

Example:

```
#include <stdio.h>
int main()
{
  int num,i=1;
  printf("Enter the number whose table you want to print?");
  scanf("%d",&num);
  table:
  printf("%d x %d = %d\n",num,i,num*i);
  i++;
  if(i<=10)
  goto table;
}
```

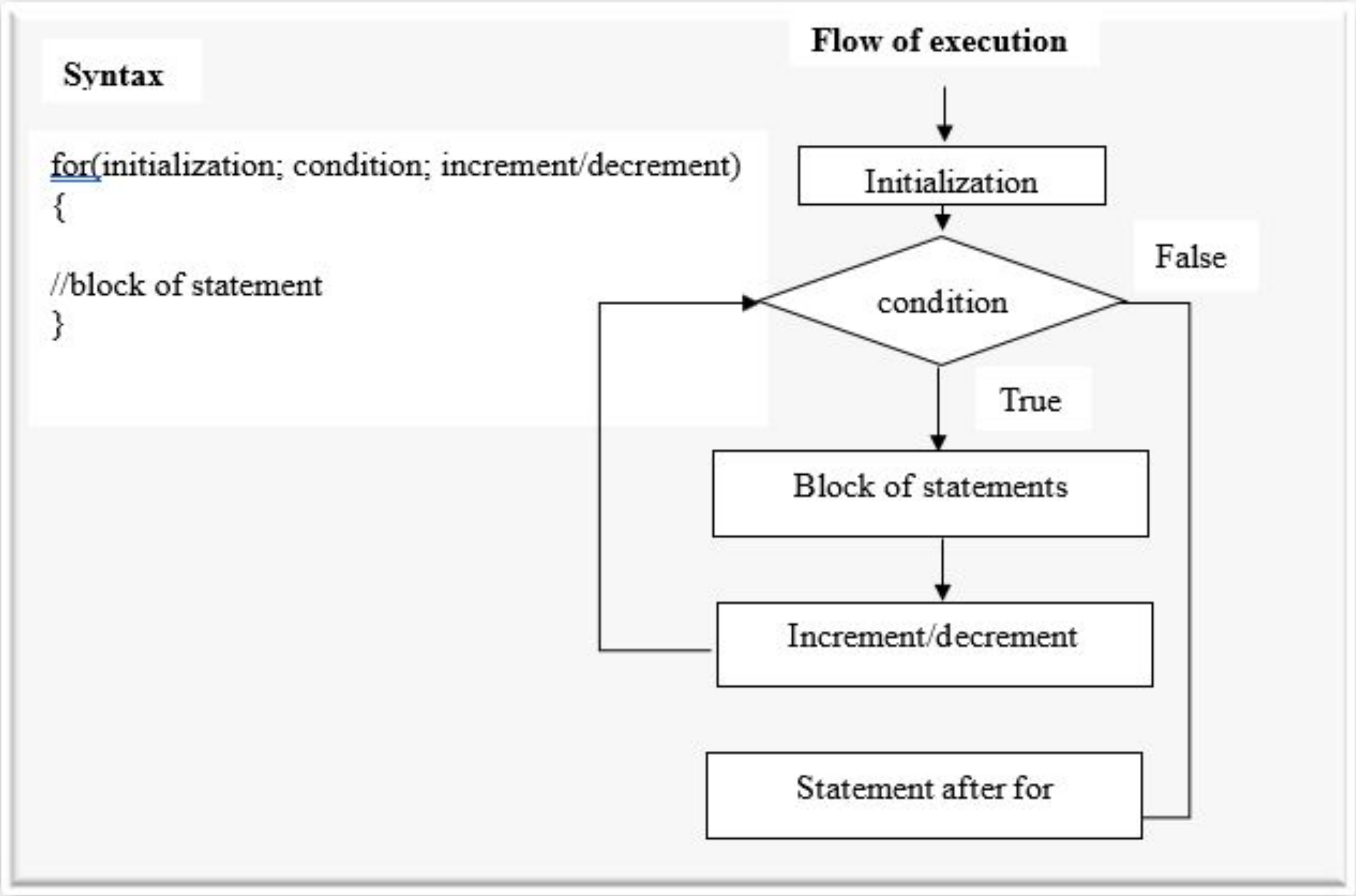
If the condition is true, then control is transferred to label

```
Enter the number whose
table you want to print?7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

Looping Control Statements

for

The for statement is used to repeatedly execute a single statement or a block of statements as long as the given condition is TRUE. The for loop consists of three parts: initialization, condition, increment or decrement.



Syntax

```
for(initialization; condition; increment/decrement)
{

//block of statement
}
```

Flow of execution

Looping Control Statements

Example:

```
#include<stdio.h>
void main()
{
int i,num=5;
for(i=1;i<=10;i++)
{
printf("%d*%d=%d\n",i,num,(i*num));
}
}
```

```
1*5=5
2*5=10
3*5=15
4*5=20
5*5=25
6*5=30
7*5=35
8*5=40
9*5=45
10*5=50
```

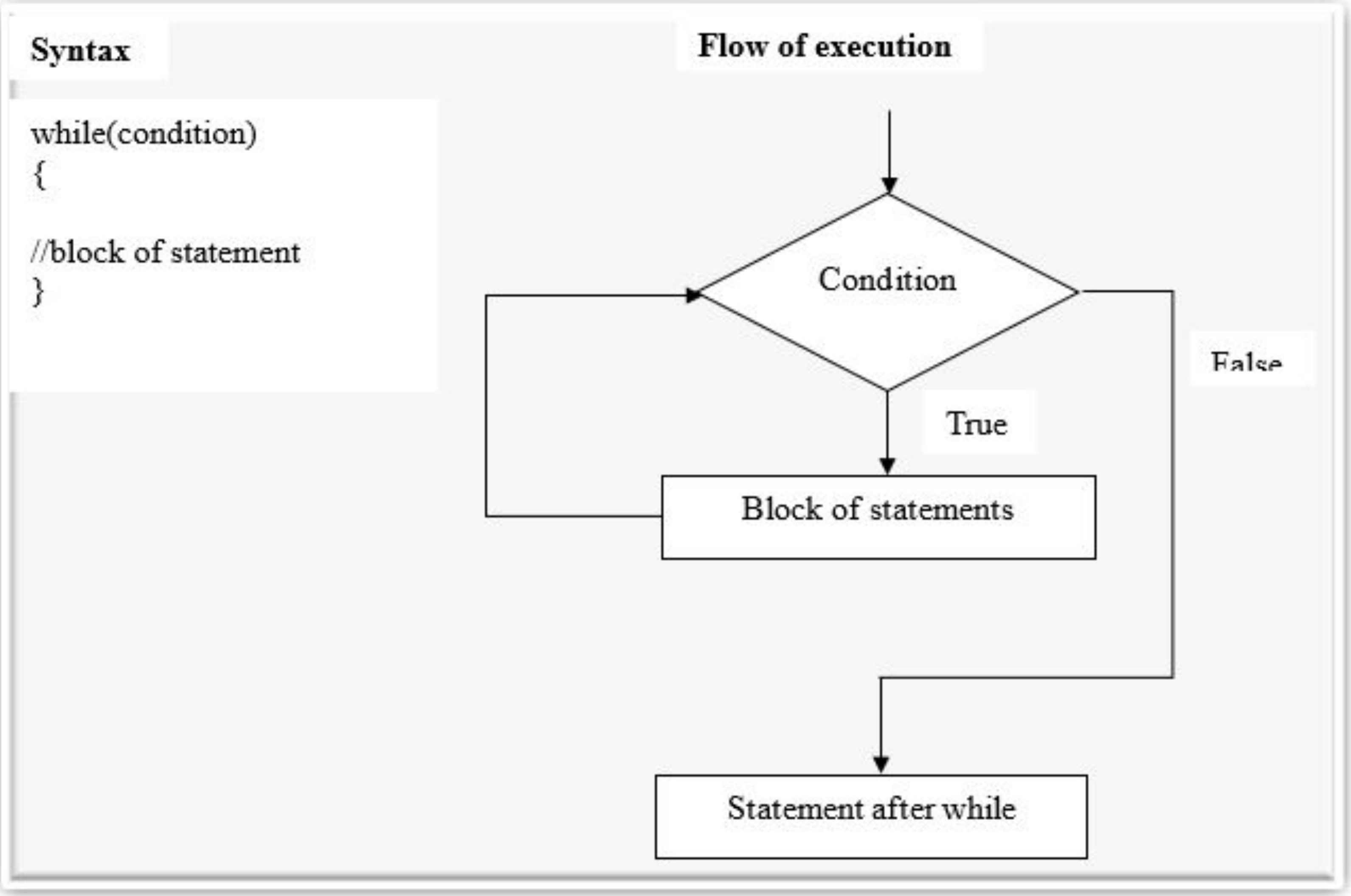
Looping Control Statements

While

It is a method for executing statements multiple times. The condition is checked in the entry level of the while loop. While the expression evaluates to true, the control will enter a loop. When the condition is false, the program control moves to the line following the while loop. If the number of iterations is not fixed, a while loop is recommended.

Looping Control Statements

While



Syntax

```
while(condition)
{

//block of statement
}
```

Flow of execution

Looping Control Statements

Example:

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

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

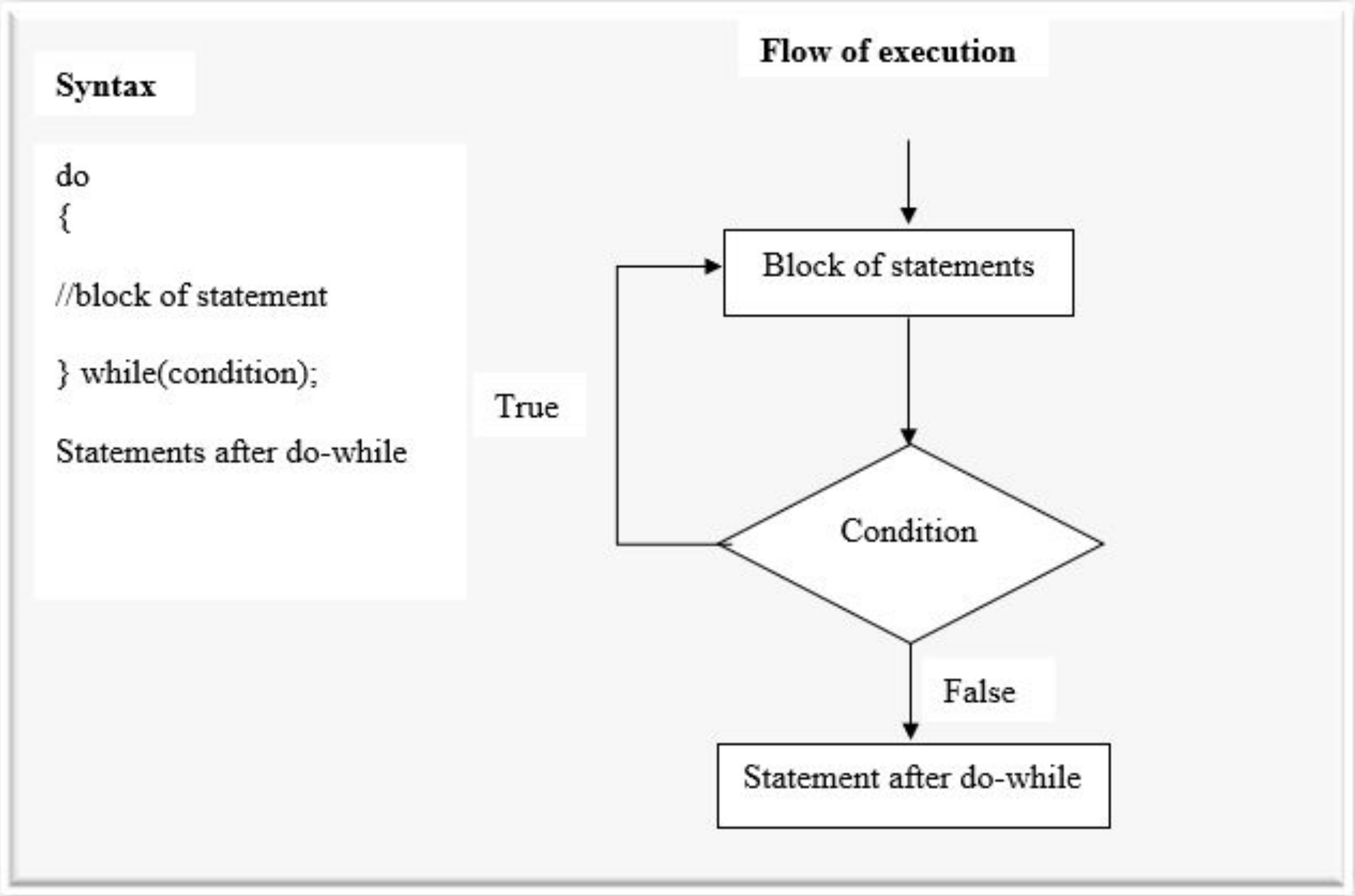
Looping Control Statements

do-while

The do while loop is a control flow statement that executes a portion of the program at least once and then depends on the given boolean condition for further execution. If the number of iterations is not fixed and the loop must be executed at least once, the do-while loop is recommended. It is a bottom tested loop that checks the condition at the bottom of the loop. The exit control looping statement is another name for the do-while statement.

Looping Control Statements

do-while



Syntax

```
do
{

//block of statement

} while(condition);

Statements after do-while
```

Flow of execution

Looping Control Statements

Example:

```
#include<stdio.h>
void main()
{
int i=18;
do
{
printf("i=%d: \n",i);
i=i+1;
}while(i<20);
}
```

```
i=18:
i=19:
```

Looping Control Statements

nested for

The nested for loop means any type of loop which is defined inside the 'for' loop.

Syntax:

```
for (initialization; condition; update)
{
    for(initialization; condition; update)
    {
        // inner loop statements.
    }
    // outer loop statements.
}
```

Looping Control Statements

Example:

```
#include <stdio.h>
int main()
{
   int n;// variable declaration
   printf("Enter the value of n :");
   scanf("%d",&n);
   // Displaying the n tables.
   for(int i=1;i<=n;i++)  // outer loop
   {
       for(int j=1;j<=3;j++)  // inner loop
       {
           printf("%d\t",(i*j)); // printing the value.
       }
       printf("\n");
   }
}
```

Output:

```
Enter the value of n :3
1       2       3
2       4       6
3       6       9
```

Looping Control Statements

nested while loop

The nested while loop means any type of loop which is defined inside the 'while' loop.

Syntax:

```
while(condition)
{
    while(condition)
    {
        // inner loop statements.
    }
// outer loop statements.
}
```

Looping Control Statements

Example:

```
#include <stdio.h>
int main()
{
    int end = 5;
    printf("Pattern Printing using Nested While loop");
    int i = 1;
    while (i <= end) {
        printf("\n");
        int j = 1;
        while (j <= i) {
            printf("%d ", j);
            j = j + 1;
        }
        i = i + 1;
    }
    return 0;
}
```

Output:

```
Pattern Printing using Nested
While loop
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Introduction to Arrays

One Dimensional(1D) Array Declaration and Initialization

In C language, arrays are referred to as structured data types. An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations.

- ✔ **finite means data range must be defined.**
- ✔ **ordered means data must be stored in continuous memory addresses.**
- ✔ **homogenous means data must be of similar data type.**

Advantages of Arrays

In one go, we can initialise storage for more than one value. Because you can create an array of 10, 100 or 1000 values.

They make accessing elements easier by providing random access. By random access we mean you can directly access any element in an array if you know its index.

Sorting and searching operations are easy on arrays.

Introduction to Arrays

One Dimensional(1D) Array Declaration and Initialization

Disadvantages of Arrays

Due to its fixed size, we cannot increase the size of an array during runtime. That means once you have created an array, then it's size cannot be changed.

Insertion and deletion of elements can be costly, in terms of time taken.

Declaring Arrays in C

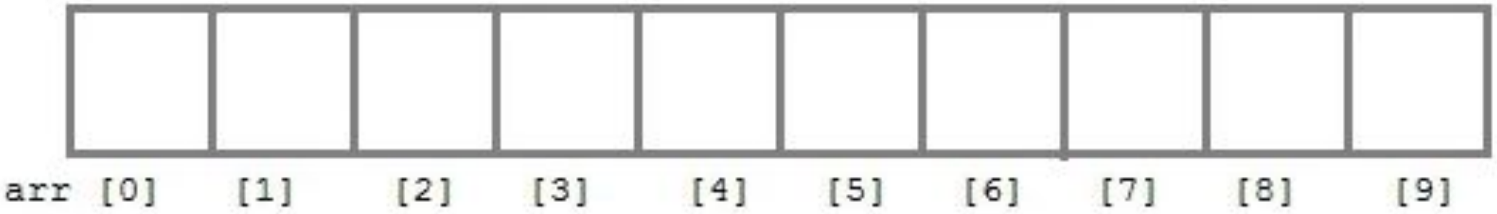
```
data-type variable-name[size];
```

/* **Example of array declaration** */

```
char a[5];    /* char type value array */
float ar[9];  /* float type value array */
int arr[10];  /* int type value array */
```

Introduction to Arrays

One Dimensional(1D) Array Declaration and Initialization



Initialization of Array in C

After an array is declared it must be initialized. Otherwise, it will contain **garbage** value(any random value). An array can be initialized at either **compile time** or at **runtime**. That means, either we can provide values to the array in the code itself, or we can add user input value into the array.

data-type array-name[size] = { list of values };

int marks[4] = { 67, 87, 56, 77 };

Introduction to Arrays

One Dimensional(1D) Array Declaration and Initialization
Example:

```
#include<stdio.h>
void main()
{
    int i;
    int arr[] = {2, 3, 4};      // Compile time array initialization
    for(i = 0 ; i < 3 ; i++)
    {
        printf("%d\t",arr[i]);
    }
}
```

Output:

```
2 3 4
```

Introduction to Arrays

Runtime Array initialization in C

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large arrays, or to initialize arrays with user specified values.

To input elements in an array, we can use a for loop or insert elements at a specific index.

For example, to insert element at specific index,

```
scanf("%d", &arr[3]); // will insert element at index 3, i.e. 4th position
```

Introduction to Arrays

Example:

```
#include<stdio.h>
void main(){

    int arr[5];
    printf("Enter array elements:"");
    for(int i = 0; i < 5; i++)
        scanf("%d", &arr[i]);

    printf("Array elements are:"");
    for(int i = 0; i < 5; i++)
        printf("%d ", arr[i]);
    int sum = 0;
    for(int i = 0; i < 5; i++)
        sum += arr[i];
    printf("Sum =%d", sum);
}
```

Output:

```
Enter array elements:3 2 4 1 5
Array elements are:3 2 4 1 5
Sum =15
```

Introduction to Arrays

Accessing, Indexing and Operations with 1D Arrays

To access all elements

```
for(int i = 0; i < 10; i++) // use index to access values from the array
    printf("%d", Arr[i]);
```

To **access and print elements at specified index**,

```
printf("%d", Arr[0]); //prints first element of the array
printf("%d", Arr[5]); //prints sixth element of the array
```

Arrays in C are indexed starting at 0, as opposed to starting at 1. The first element of the array above is point[0]. The index to the last value in the array is the array size minus one. In the example above the subscripts run from 0 through 5. C does not guarantee bounds checking on array accesses

Introduction to Arrays

Accessing, Indexing and Operations with 1D Arrays

```
char y;

int z = 9;

char point[6] = { 1, 2, 3, 4, 5, 6 };

//examples of accessing outside the array. A compile error is not always raised

y = point[15];

y = point[-4];

y = point[z];
```

Introduction to Arrays

Operations on Arrays in C

There are a number of operations that can be performed on an array which are:

1. Traversal
2. Copying
3. Reversing
4. Sorting
5. Insertion
6. Deletion
7. Searching
8. Merging

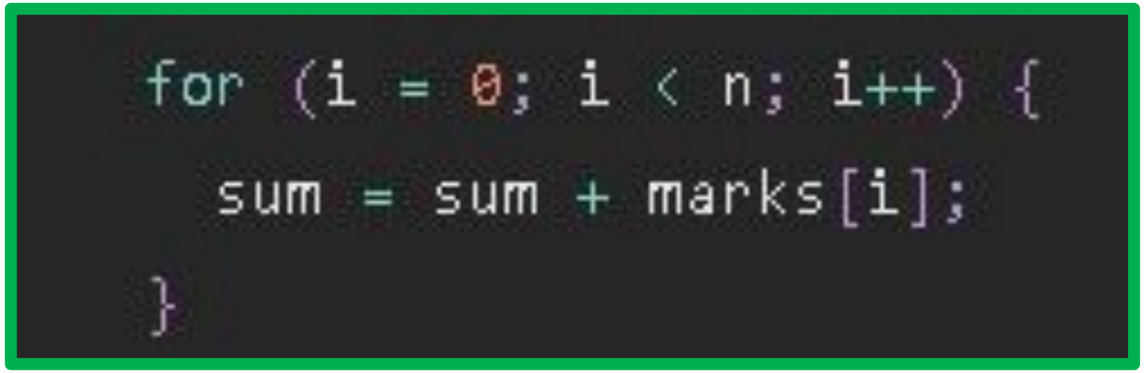
Introduction to Arrays

Operations on Arrays in C

There are a number of operations that can be performed on an array which are:

1. **Traversal**

Traversal means accessing each array element for a specific purpose, either to perform an operation on them , counting the total number of elements or else using those values to calculate some other result.



```
for (i = 0; i < n; i++) {
   sum = sum + marks[i];
}
```

Introduction to Arrays

Operations on Arrays in C

There are a number of operations that can be performed on an array which are:

2. Copying elements of an Array

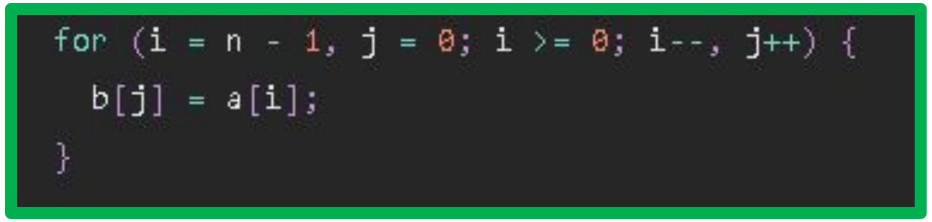
// Copying data from source array A to destination array 'b

```
 for (i = 0; i < num; i++) {
   arr2[i] = arr1[i];        }
```

3. Reversing an elements of an Array

Reversing an array means that the sequence of elements of array will be reversed.

For instance if your array ‘A’ has two elements : A[0] = 1; A[1] = 2; then after reversal A[0] = 2 and A[1] = 1.



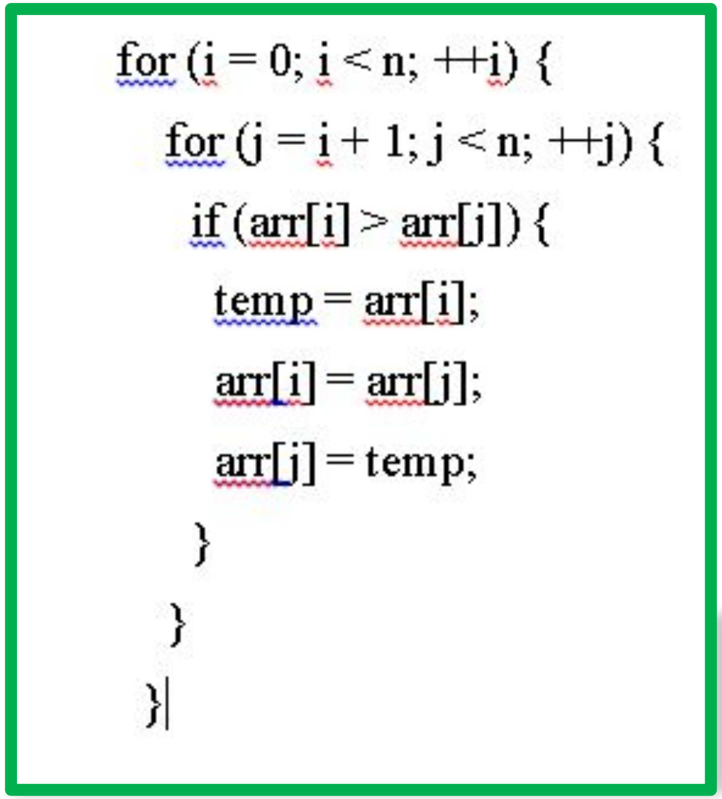
```
for (i = n - 1, j = 0; i >= 0; i--, j++) {
  b[j] = a[i];
}
```

Introduction to Arrays

Operations on Arrays in C

4. Sorting elements of an Array

Sorting elements if array means to order the elements in ascending or descending order – usually in ascending order.



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

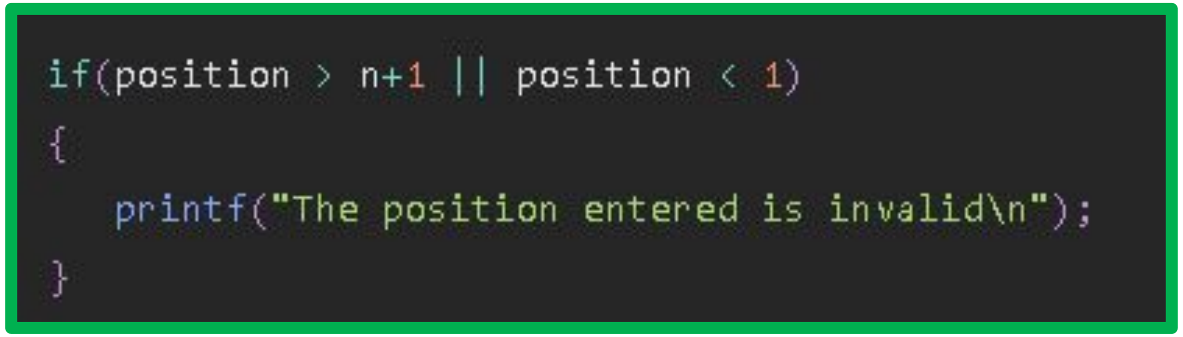
Introduction to Arrays

Operations on Arrays in C

5. Insertion

```
printf("Enter the value to insert\n");
  scanf("%d", &value);
  for (i = n - 1; i >= position - 1; i--)
     array[i+1] = array[i];
  array[position-1] = value;  //inserting value at the required location
```

Position invalid



```
if(position > n+1 || position < 1)
{
   printf("The position entered is invalid\n");
}
```

Important:

- Insertion can be done at any position
- Check the entered position is valid or not
- If position is valid, the element is inserted at required position

Introduction to Arrays

Operations on Arrays in C

6. Deletion

```
for (i = position - 1; i < n - 1; i++)
        array[i] = array[i+1];
```

Position invalid

```
if (position >= n+1 || position < 0)
printf("Deletion not possible as entered location is invalid.\n");
```

Introduction to Arrays

Array programs- 1D-*(Example-1)*

```
#include <stdio.h>
int main() {
    int arr[5] = {1,3,5,7,9};
    printf("The elements of arrays are :");
    for(int i=0; i<5; i++){
    printf(" %d",arr[i]);
    }
    return 0;
}
```

Output:

```
The elements of arrays are : 1 3 5 7 9
```

Introduction to Arrays

Array programs- 1D-*(Example-2)*

```
#include <stdio.h>
int main() {
    int arr[12] = {1,3,5,7,9,10,12,14,23,22,33,35};
    printf("The even elements in arrays are :");
    for(int i=0; i<12; i++){
    if(arr[i]%2==0) printf(" %d",arr[i]);
}
return 0;
}
```

Output:

```
The even elements in arrays are : 10 12 14 22
```

Introduction to Arrays

Array programs- 1D-*(Example-3)*

```
#include <stdio.h>
void  main()
{
    int arr[10];
    int i;
       printf("\n\nRead and Print elements of an array:\n");
       printf("----------------------------------------\n");
      printf("Input 10 elements in the array :\n");
    for(i=0; i<10; i++)
    {
           printf("element - %d : ",i);
         scanf("%d", &arr[i]);
    }
```

Introduction to Arrays

Array programs- 1D-*(Example-3)*

```
printf("\nElements in array are: ");
    for(i=0; i<10; i++)
    {
        printf("%d  ", arr[i]);
    }
    printf("\n");
}
```

Output:

```
Read and Print elements of an array:
--------------------------------------------
Input 10 elements in the array :
element - 0 : 1
element - 1 : 2
element - 2 : 3
element - 3 : 4
element - 4 : 5
element - 5 : 7
element - 6 : 8
element - 7 : 9
element - 8 : 1
element - 9 : 1
Elements in array are: 1  2  3  4  5  7  8  9  1  1
```

Introduction to Arrays

Array programs- 1D-*(Example-4)*

```
#include <stdio.h>
void main()
{
    int a[100];
    int i, n, sum=0;
       printf("\n\nFind sum of all elements of array:\n");
       printf("--------------------------------------\n");
       printf("Input the number of elements to be stored in the array :");
       scanf("%d",&n);
       printf("Input %d elements in the array :\n",n);
       for(i=0;i<n;i++)
        {
             printf("element - %d : ",i);
             scanf("%d",&a[i]);
           }
```

Introduction to Arrays

Array programs- 1D-*(Example-4)*

```
    for(i=0; i<n; i++)
    {
        sum += a[i];// sum=sum+a[i]
    }
    printf("Sum of all elements stored in the array is : %d\n\n", sum);
}
```

Output:

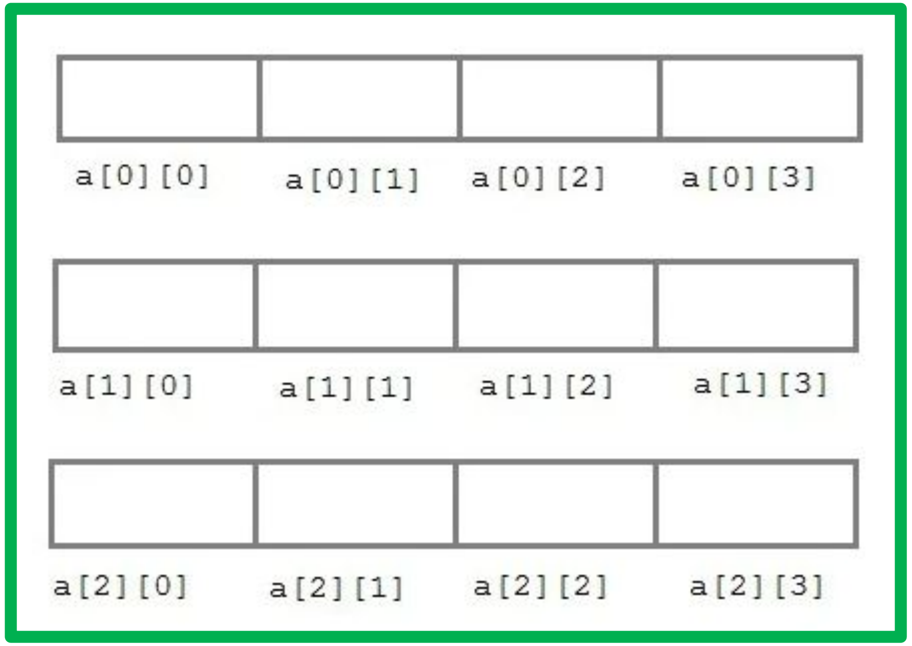
```
Find sum of all elements of array:
----------------------------------------------
Input the number of elements to be stored in the array :3
Input 3 elements in the array :
element - 0 : 2
element - 1 : 5
element - 2 : 8
Sum of all elements stored in the array is : 15
```

Introduction to Arrays

Initializing and Accessing 2D Arrays

Declaration

data-type array-name[row-size][column-size];
double arr[5][5];
int a[3][4];



a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

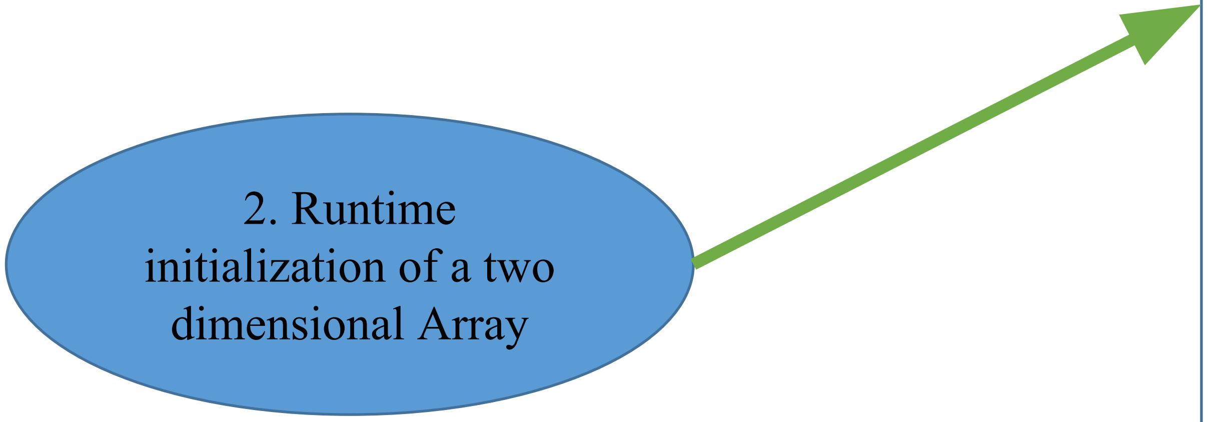
Introduction to Arrays

1.Initializing and Accessing 2D Arrays
Compile-time initialization of a two dimensional Array

int arr[][3] = { {0,0,0}, {1,1,1}};

char a[][2] = {{'a', 'b'},{'c', 'd'}};

int arr1[2][2] = {1, 2, 3, 4};



2. Runtime initialization of a two dimensional Array

```
#include<stdio.h>
void main()
{
    int arr[3][4];
    int i, j, k;
    printf("Enter array elements:\n");
    for(i = 0; i < 3;i++)
    {
        for(j = 0; j < 4; j++)
        {
            scanf("%d", &arr[i][j]);
        }
    }
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 4; j++)
        {
           printf("%d", arr[i][j]);
        }
    }
}
```

Introduction to Arrays

Array programs- 2D-*(Example-1)*

```
#include<stdio.h>
int main(){
  /* 2D array declaration*/
  int arr[2][3];
  /*Counter variables for the loop*/
  int i, j;
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
      printf("Enter value for disp[%d][%d]:", i, j);
      scanf("%d", &arr[i][j]);
    }
  }
```

Introduction to Arrays

```
//Displaying array elements
  printf("Two Dimensional array elements:\n");
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
      printf("%d ", arr[i][j]);
      if(j==2){
        printf("\n");
      }
    }
```

Output:

```
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:3
Enter value for disp[1][1]:2
Enter value for disp[1][2]:1
Two Dimensional array elements:
1  2  3
3  2  1
```

Introduction to Arrays

Array programs- 2D-*(Example-2)*

```
#include <stdio.h>
void main()
{
 int arr1[50][50],brr1[50][50],crr1[50][50],i,j,n;
       printf("\n\nAddition of two Matrices :\n");
       printf("------------------------------\n");
        printf("Input the size of the square matrix (less than 5): ");
       scanf("%d", &n);
      /* Stored values into the array*/
       printf("Input elements in the first matrix :\n");
       for(i=0;i<n;i++)
        {
           for(j=0;j<n;j++)
           {
                 printf("element - [%d],[%d] : ",i,j);
                 scanf("%d",&arr1[i][j]);
           }          }
```

Introduction to Arrays

Array programs- 2D-*(Example-2)*

```
    printf("Input elements in the second matrix :\n");
    for(i=0;i<n;i++)
     {
        for(j=0;j<n;j++)
        {
            printf("element - [%d],[%d] : ",i,j);
            scanf("%d",&brr1[i][j]);
        }
     }
 printf("\nThe First matrix is :\n");
for(i=0;i<n;i++)
  {
   printf("\n");
   for(j=0;j<n;j++)
        printf("%d\t",arr1[i][j]);
  }
```

Output:

```
EntAddition of two Matrices :
------------------------------
Input the size of the square matrix (less than
5): 2
Input elements in the first matrix :
element - [0],[0] : 1
element - [0],[1] : 2
element - [1],[0] : 3
element - [1],[1] : 4
Input elements in the second matrix :
element - [0],[0] : 5
element - [0],[1] : 6
element - [1],[0] : 7
element - [1],[1] : 8
```

Introduction to Arrays

Array programs- 2D-*(Example-2)*

```
printf("\nThe Second matrix is :\n");
  for(i=0;i<n;i++)
    {
      printf("\n");
      for(j=0;j<n;j++)
      printf("%d\t",brr1[i][j]);
    }
for(i=0;i<n;i++)
       for(j=0;j<n;j++)
            crr1[i][j]=arr1[i][j]+brr1[i][j];
   printf("\nThe Addition of two matrix is : \n");
   for(i=0;i<n;i++){
       printf("\n");
       for(j=0;j<n;j++)
            printf("%d\t",crr1[i][j]);
   }
   printf("\n\n"); }
```

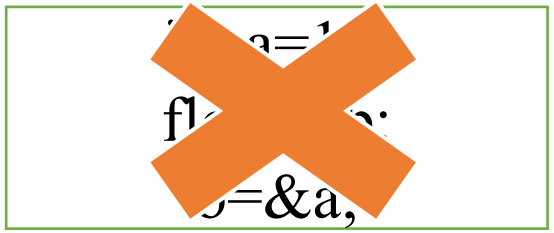
```
he First matrix is :
1       2
3       4
The Second matrix is :
5       6
7       8
The Addition of two matrix is :
6       8
10      12
```

Pointers

Introduction

- Pointers are special variables which contain address of any other variable.
- Pointer always contain address.
- Pointers are derived data type.(Because pointer is derived from the fundamental data types).
- Pointer is mainly used for dynamic memory allocation.

For example, an integer variable holds an integer value . **An** integer pointer holds the address of a integer variable.



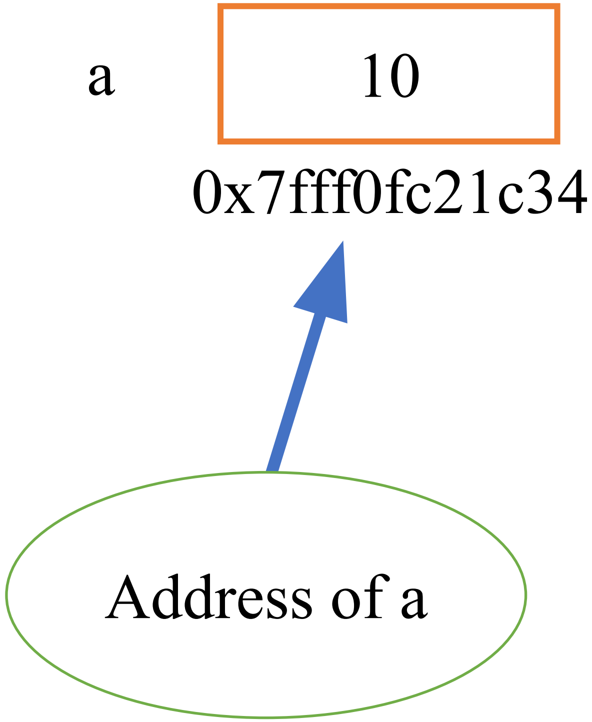
```
a=1
fl     b;
b=&a;
```

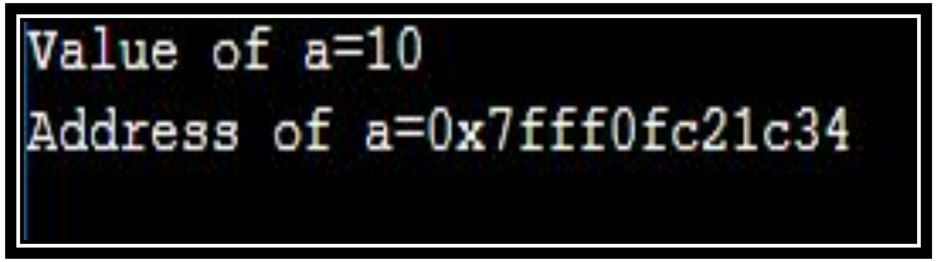
Here b is the pointer which contains address of the variable whose data type is float. It can hold address of float variable only.

Introduction

Example:

```
#include <stdio.h>

int main()

{

   int a=10;

   printf("Value of a=%d\n",a);

   printf("Address of a=%p",&a);

    return 0;

}
```





```
Value of a=10
Address of a=0x7fff0fc21c34
```

The %p is used to print the pointer value, and %x is used to print hexadecimal values.

How to declare pointer?

Syntax:

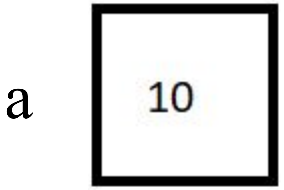
```
data_type *pointer_name;
or
data_type* pointer_name;
or
data_type  *  pointer_name;
```

The data type of the pointer and the variable to which the pointer variable is pointing must be the same.

How to initialize pointer variable?

Syntax:

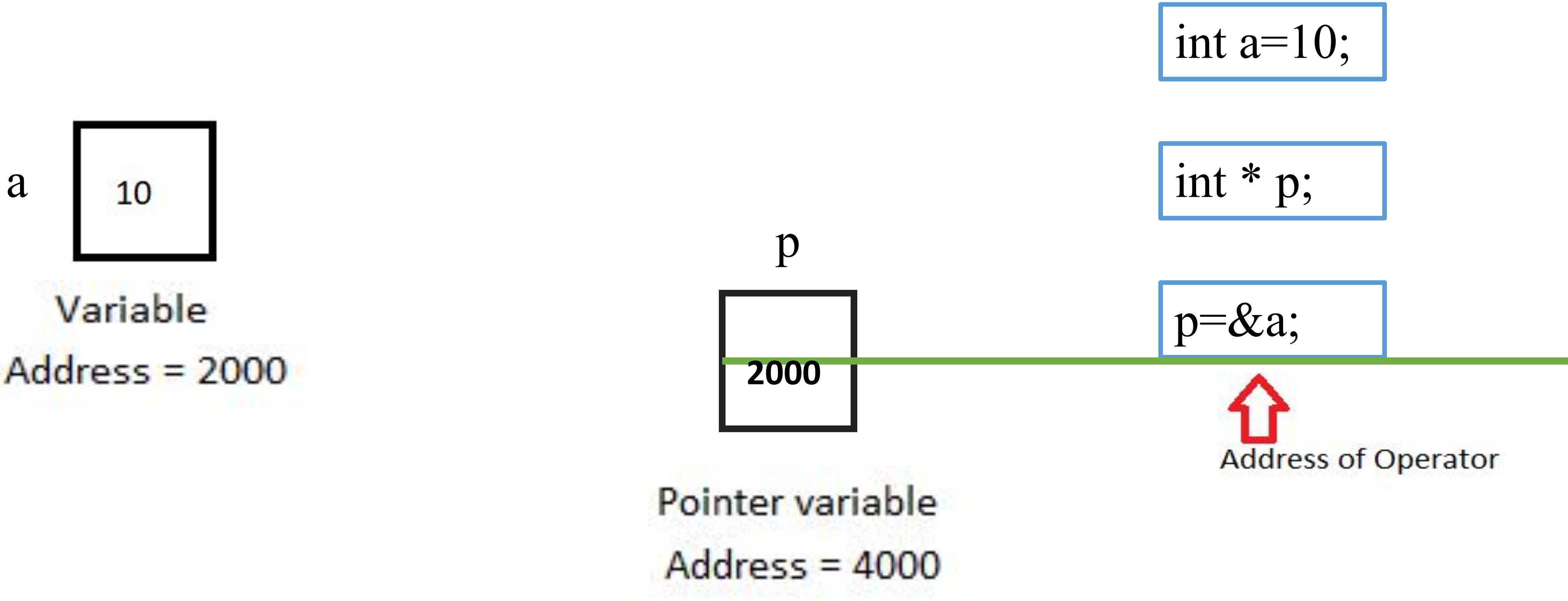
Pointer Initialization is the process of **assigning address** of a **variable** to a **pointer variable.** It contains the address of a variable of the same data type. In C language address operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.



```
int a=10;
```

```
int * p;
```

How to initialize pointer variable?



```
int a=10;

int * p;

p=&a;
```

a

10

Variable

Address = 2000

p

2000

Pointer variable

Address = 4000

Address of Operator

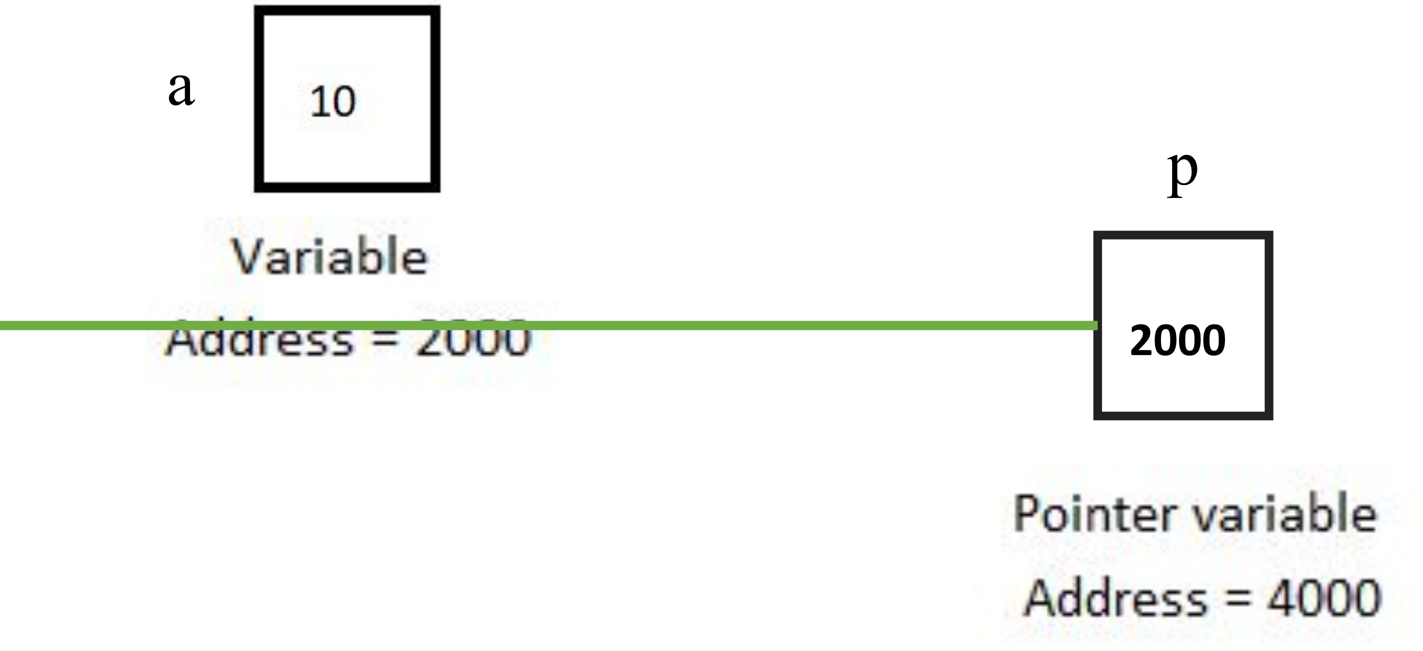
How to declare and initialize pointer variable in the same line?

```
int a=10,*p=&a;
```

```
int *p=&a, a=10;
```

✖

How will you access the value stored in address 2000?



a

10

Variable

Address = 2000

p

2000

Pointer variable

Address = 4000

Indirection (or) dereferencing operator

& (address of) and * (Indirection Operator)

The & operator is used to find the address of the variable.

The * operator is used to access the value of the variable.

```
#include <stdio.h>
int main()
{
   int a=10;
   int *p;
   p=&a;
   printf("Address of a=%p\n", p);
   printf("Value of a=%d",*p);
    return 0;
}
```

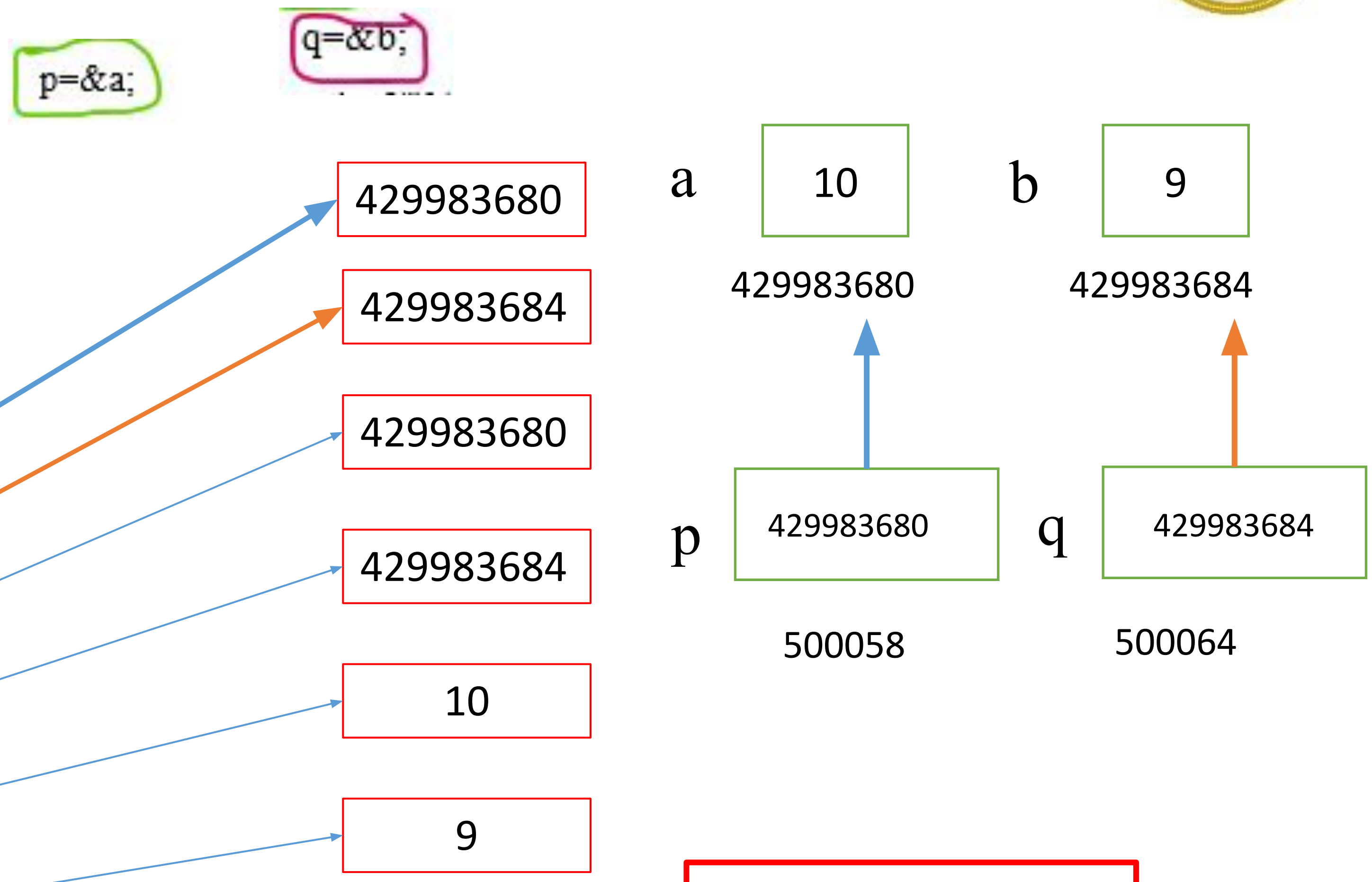
To print the address of a

To access the value of a

& (address of) and * (Indirection Operator)

Example:

```
#include <stdio.h>
int main()
{
  int a=10,b=9;
   int *p,*q;
   p=&a;
   q=&b;
   printf("%u\n",&a);
   printf("%u\n",&b);
   printf("%u\n",p);
   printf("%u\n",q);
   printf("%d\n",*p);
   printf("%d",*q);
   return 0;
}
```



p=(&a)
p=(429983680)
Value at 429983680
=>10

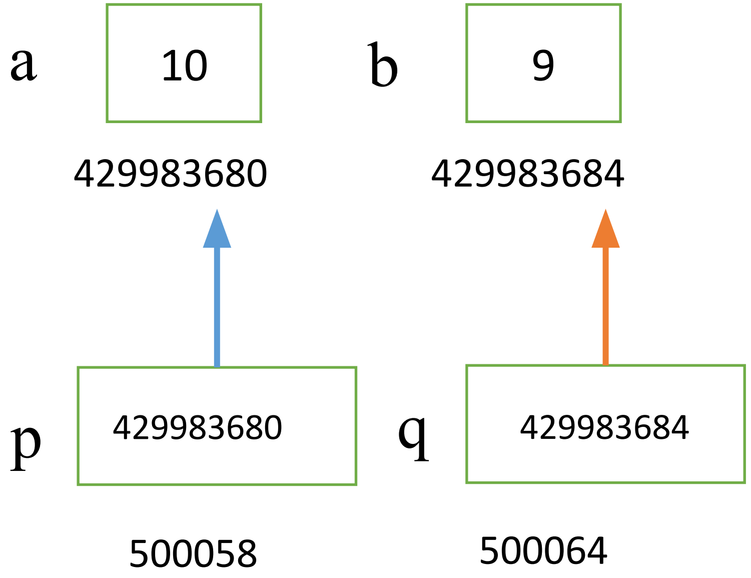
& (address of) and * (Indirection Operator)

Example:

```
#include <stdio.h>
int main()
{
  int a=10,b=9;
   int *p,*q;
   p=&a,&b;
   printf("%u\n",&a);
   printf("%u\n",&b);
   printf("%u\n",p);
   printf("%u\n",q);
   printf("%d\n",*p);
   return 0;
}
```

The precedence of = symbol is higher than comma operator.
In this case, address of a is assigned to p.
&b will be discarded.

```
429983680
429983684
429983680
500064
10
```



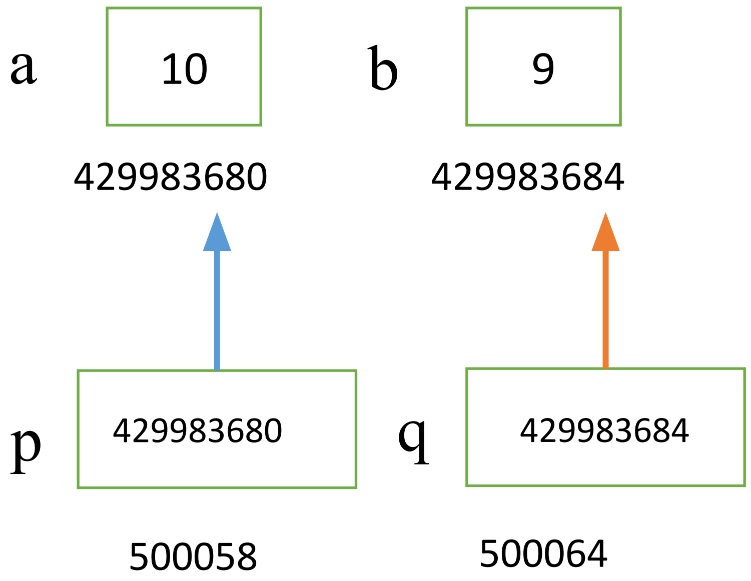
& (address of) and * (Indirection Operator)

Example:

```
#include <stdio.h>
int main()
{
  int a=10,b=9;
   int *p,*q;
   p=(&a,&b);
   printf("%u\n",&a);
   printf("%u\n",&b);
   printf("%u\n",p);
   printf("%u\n",q);
   printf("%d\n",*p);
   return 0;
}
```

Here, brackets are having higher precedence than assignment = operator. The first operant will be evaluated and discarded. Then second operand is evaluated and address of b is assigned to p.

```
429983680
429983684
429983684
500064
9
```



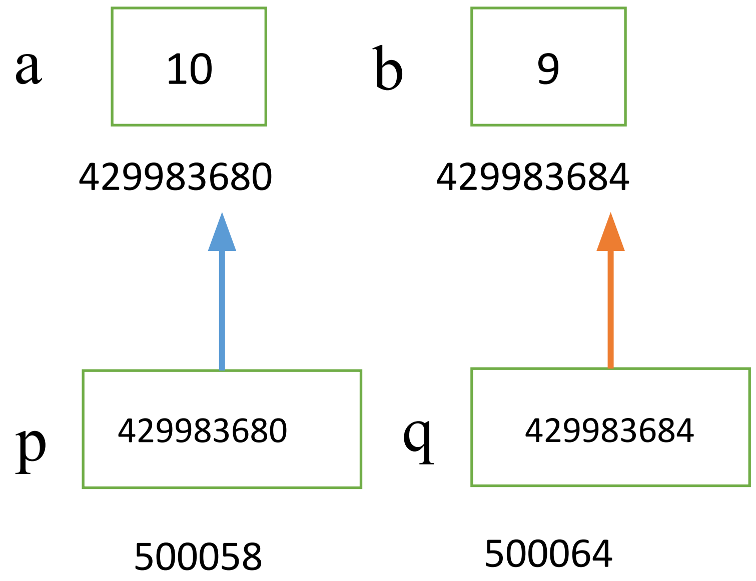
& (address of) and * (Indirection Operator)

Example:

```
#include <stdio.h>
int main()
{
  int a=10,b=9;
   int *p,*q;
   p=&a;
    p=&b;
   printf("%u\n",&a);
   printf("%u\n",&b);
   printf("%u\n",p);
   printf("%u\n",q);
   printf("%d\n",*p);
   return 0;
}
```

Initially address of a is assigned to p. In turn address of b is assigned to pointer variable p.

```
429983680
429983684
429983684
500064
9
```



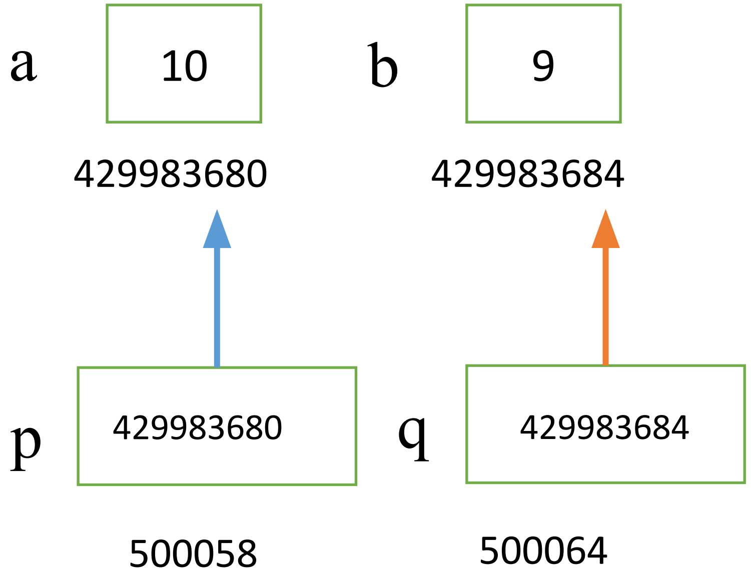
& (address of) and * (Indirection Operator)

Example:

Initially address of a is assigned to p. In turn address of b is assigned to pointer variable p.

```
#include <stdio.h>
int main()
{
   int a=10,b=9;
    int *p,*q;
    p=&a;
     q=&a;
    printf("%u\n",&a);
    printf("%u\n",&b);
    printf("%u\n",p);
    printf("%u\n",q);
    printf("%d\n",*p);
    return 0;
}
```

```
429983680
429983684
429983680
429983680
10
```



& (address of) and * (Indirection Operator)

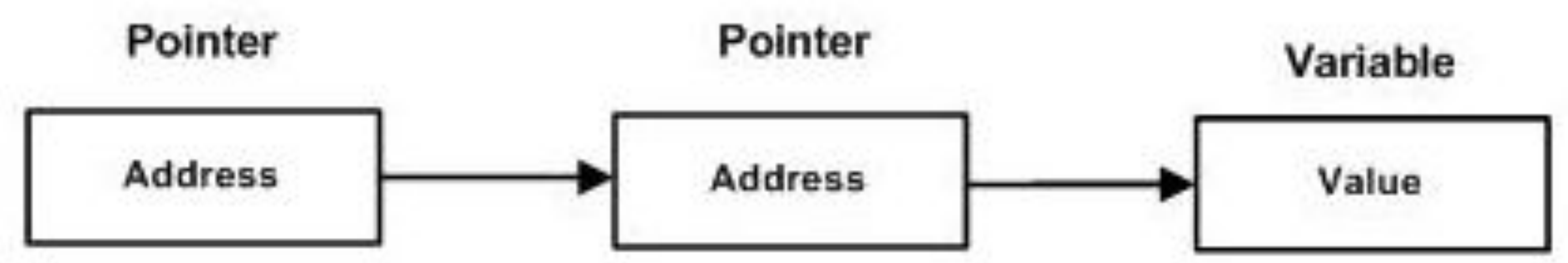
Example:

```
#include <stdio.h>
int main()
{
  int a=10,b=9;
  int c;
  int *p,*q;
  p=&a;
  q=&b;
   c=*p;
  printf("Value of a=%d\n",a);
  printf("Value of a=%d\n",*p);
  printf("%d",c);
     return 0;
}
```

```
Value of a=10
Value of a=10
10
```

Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



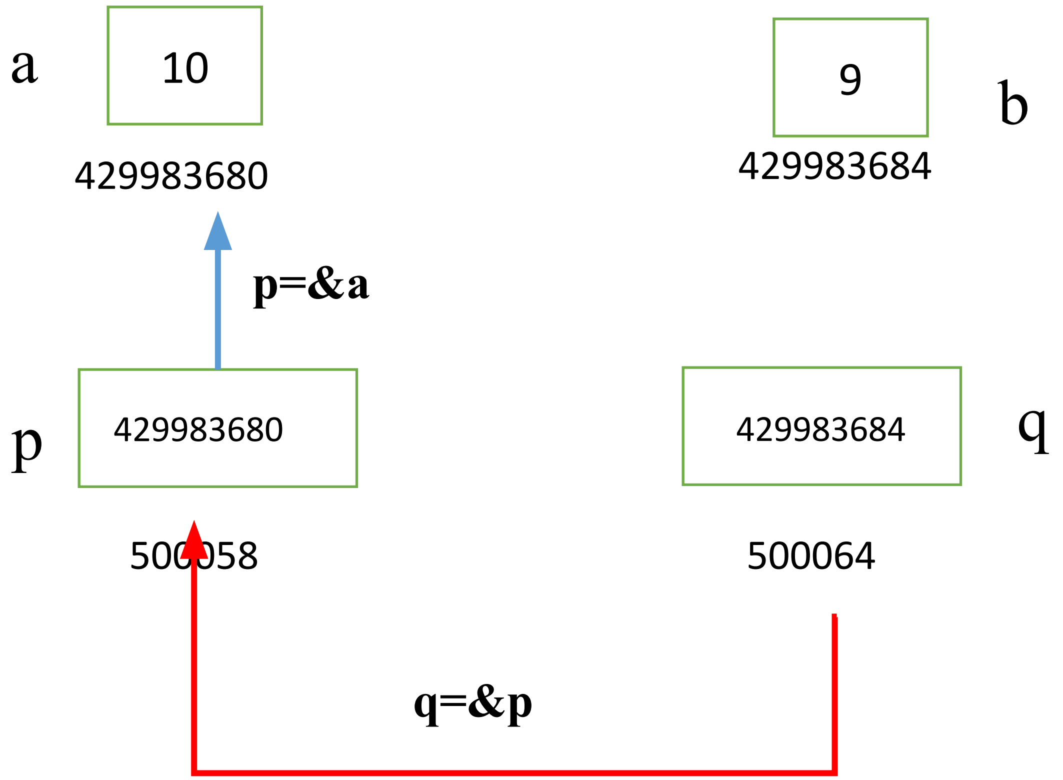
Second level pointer store the address of first level pointer. The first level pointer holds the address of actual variable.

Pointer to Pointer

For example, the following declaration declares a pointer to a pointer of type int –

```
int **var;
```

** pointer is the special variable, used to store address of another pointer variable.



a

10

429983680

9

b

429983684

p=&a

p

429983680

500058

429983684

q

500064

q=&p

```
int a=10;
int *p;
int **q;
p=&a;
q=&p;
```

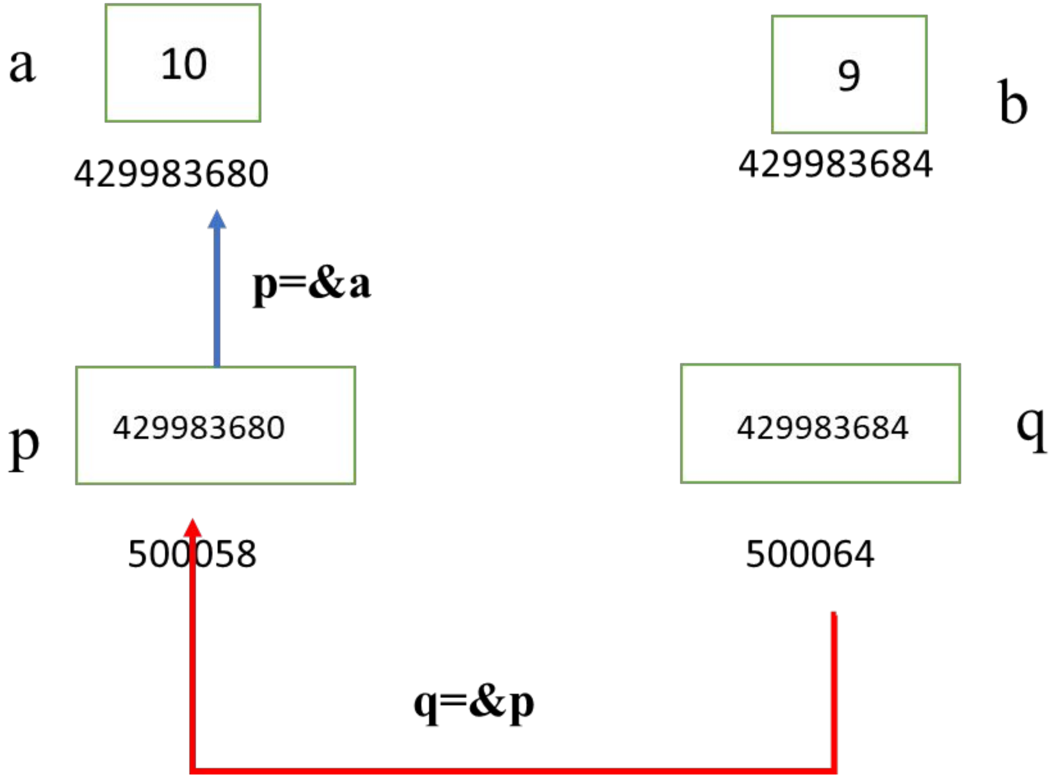
Pointer to Pointer

Example:

```
#include <stdio.h>
int main()
{
   int a=10,b=9;
   int *p;
   int **q;
   p=&a;
   q=&p;
   printf("%d\n",**q);
   printf("%d\n",*p);
   printf("%d\n",*q);
   printf("%d\n",p);
   printf("%d\n",a);
    return 0;
}
```

**q=*(*(q))=*(*(&p))=*(*(50008))=*(429983680)=10

```
10
10
819183264
819183264
10
```



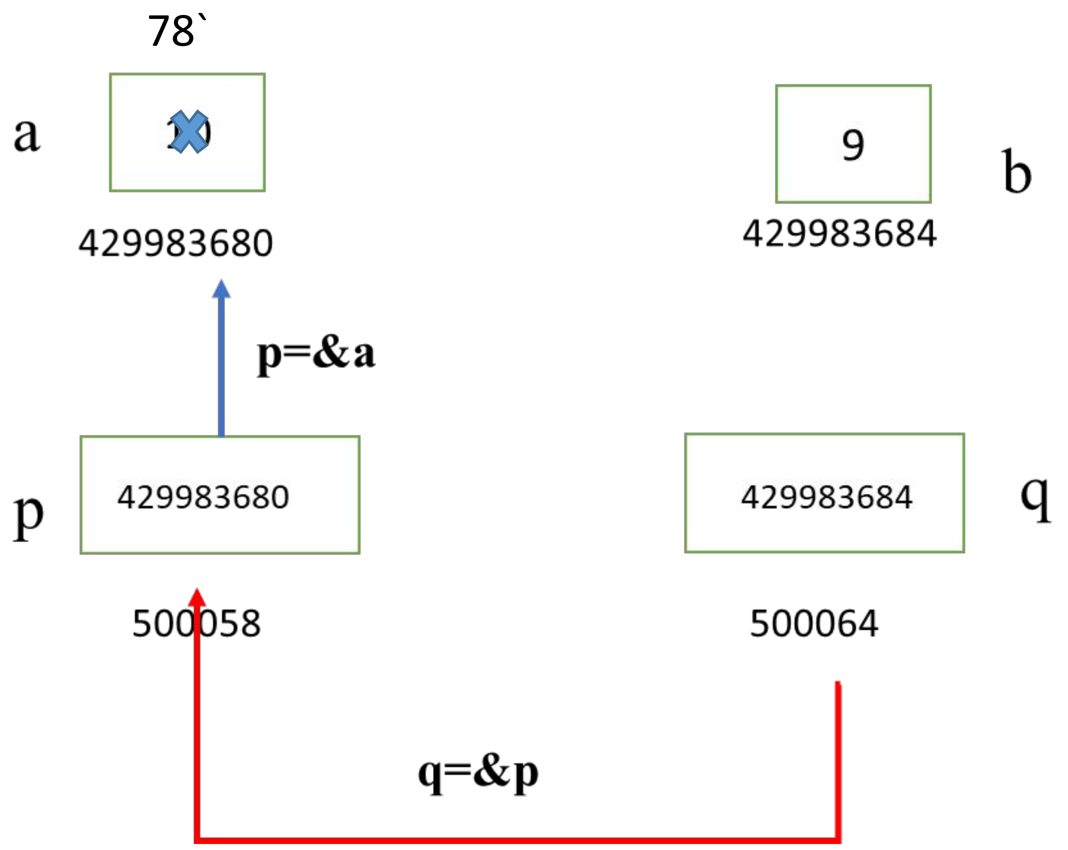
printf("%d%d%d",a,*p,q);**

Pointer to Pointer

Example:

```
#include <stdio.h>

int main()
{
   int a=10,b=9;
   int *p;
   int **q;
   p=&a;
   q=&p;
   printf("%d\n",**q);
   printf("%d\n",*p);
   printf("%d\n",*q);
   printf("%d\n",p);
   printf("%d\n",a);
   **q=78;
   printf("%d",a);
    return 0;
}
```

```
10
10
-1288442720
-1288442720
10
78
```



Void Pointers

The void pointer in C is a pointer that is not associated with any data types. It points to some data location in the storage. This means that it points to the address of any variable. It is also called the general purpose pointer.

For example, if we declare the int pointer, then this int pointer cannot point to the float variable or some other type of variable, i.e., it can point to only int type variable. **To overcome this problem, we use a pointer to void.** A pointer to void means a generic pointer that can point to any data type. We can assign the address of any data type to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

Syntax:

void *pointer_name;

Example:

void *ptr;

In the above declaration, the void is the type of the pointer, and 'ptr' is the name of the pointer.

Void Pointers

Let us consider some examples:

```
int i=9;         // integer variable initialization.
int *p;          // integer pointer declaration.
float *fp;          // floating pointer declaration.
void *ptr;          // void pointer declaration.
p=fp;          // incorrect.
fp=&i;          // incorrect
ptr=p;          // correct
ptr=fp;          // correct
```

The size of the void pointer is the same as the size of the pointer of character type.

```
void *ptr;
char *cp;
    printf("size of void pointer = %d\n\n",sizeof(ptr));  // 8
    printf("size of character pointer = %d\n\n",sizeof(cp));  //8
```

Void Pointers

Dereferencing void pointer

```
#include <stdio.h>
int main()
{
   int a=90;
   void *ptr;
   ptr=&a;   // void pointer cannot be dereferenced
   printf("Value which is pointed by ptr pointer : %d",*ptr);
   return 0;
}
```

The above program will generate error

```
  printf("Value which is pointed by ptr pointer : %d",*ptr); //Incorrect
  printf("Value which is pointed by ptr pointer : %d",*(int*)ptr);   //correct
```

Void Pointers

A null pointer is a pointer that does not point to any memory location and hence does not hold the address of any variables. It stores the base address of the segment.

The null pointer can be defined in two ways

int *pointer_var=NULL:

(or)

int *pointer_var=0;

Applications of NULL pointer

a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.

b) To pass a null pointer to a function argument when we don't want to pass any valid memory address.

c) To check for null pointer before accessing any pointer variable.

Void Pointers

```
#include <stdio.h>
int main()
{
    int *ptr;  //pointer is not initialized
   printf("Address: %d", ptr); // printing the value of ptr.
   printf("Value: %d", *ptr); // dereferencing the illegal pointer
   return 0;
}
```

The above program generate error when dereferencing of the uninitialized pointer variable.

How to avoid the above problem?

```
#include <stdio.h>
int main()
{
    int *ptr=NULL;
   if(ptr!=NULL)
   {
      printf("value of ptr is : %d",*ptr);
   }
   else
   {
      printf("Invalid pointer");
   }
 return 0;
}
```

Pointer Based Array Manipulation

How to access elements of an array using a pointer?

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

Thank You