



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# C / C++ Course Slides

# Content

---

**Module 1** - [ C,C++ -Introduction]

**Module 2** - [C,C++ - Fundamentals of C

**Module 3** - [C Language Programming with C]

**Module 4** - [C,C++ - File and Error Handling,      Debugging]

**Module 5** - [C,C++ - C++ (Basics of C++) ]

**Module 6** - [C,C++ - Programming with C++]

**Module 7** - [C,C++ - Exception Handling and Templates]



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# **Module - 1**

# **[C,C++ -Introduction]**

# What is Program

- Program- It is a set of Instructions

**Ex. 1- Instructions to your Pet**

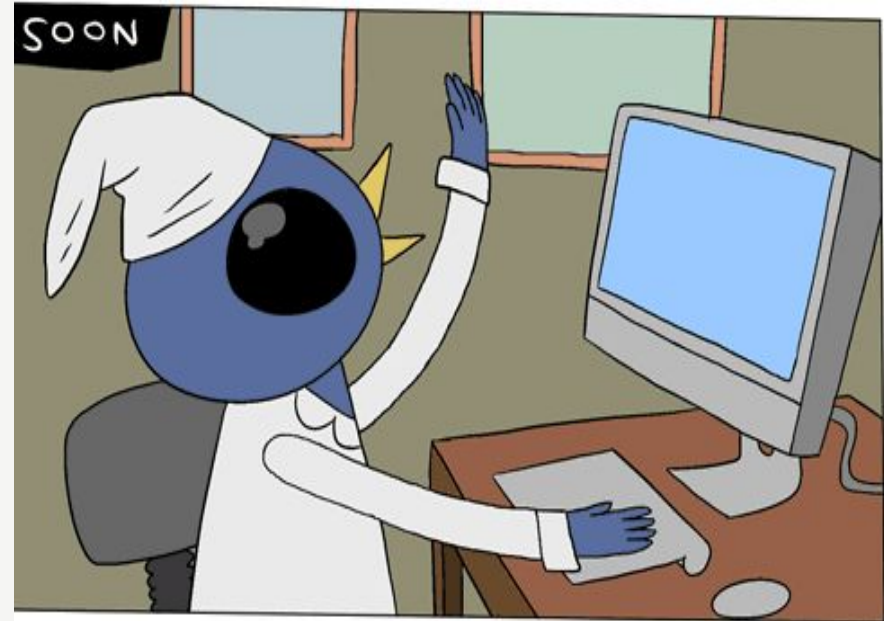
**Ex. 2- Starting your Computer**

```
>Hello world!  
>_
```

# What is Programming

- Programming- To create a Program.

**Ex. 1- Using Keyboard & Mouse**



# Types of Programming Language

- Procedural Programming
  - Ex.- C Language



# Types of Programming Language

- Object Oriented Programming
  - Ex.- C++ Language



# Types of Programming Language

- Logical Programming
  - Ex.- Prolog Language





# Types of Programming Language

- Functional Programming
  - Ex.- Python Language



# Why to Learn C Programming?

- Easy to learn
- Structured language
- Produces efficient programs
- Compiled on various computer platforms



# Facts related to C

- **Facts related to C**

- Invented to write UNIX(Operating System).
- Successor of B language.
- Most widely used System Programming Language.
- Linux OS and MySQL Database are made on it.

# Hello World Program using C

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

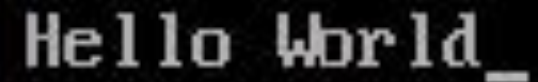
```
int main()
```

```
{
```

```
printf("Hello World \n");
```

```
return 0;
```

```
}
```



```
Hello World_
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# **Module - 2**

## **[C,C++ - Fundamentals of C ]**



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Setting up C Environment

# What is a Text Editor

- Application used to type your Program

**Ex. - Windows Notepad, VS Code,  
Notepad++ etc.**

```
>Hello world!  
>_
```

# What is a C Compiler

- Program written by us needs to be converted into machine understandable code.

**Ex. - GCC Compiler**

```
F:\>tcc
tcc version 0.9.23 - Tiny C Compiler - Copyright (C) 2001-2005 Fabrice Bellard
usage: tcc [-v] [-c] [-o outfile] [-Bdir] [-bench] [-ldir] [-Dsym[=val]] [-Usym]
        [-Wwarn] [-g] [-b] [-bt N] [-ldir] [-llib] [-shared] [-static]
        [infile1 infile2...] [-run infile args...]

General options:
-v          display current version
-c          compile only - generate an object file
-o outfile  set output filename
-Bdir       set tcc internal library path
-bench      output compilation statistics
-run        run compiled source
-fflag      set or reset (with 'no-' prefix) 'flag' (see man page)
-Wwarning   set or reset (with 'no-' prefix) 'warning' (see man page)
-w          disable all warnings

Preprocessor options:
-ldir       add include path 'dir'
-Dsym[=val] define 'sym' with value 'val'
-Usym       undefine 'sym'

Linker options:
-ldir       add library path 'dir'
-llib       link with dynamic or static library 'lib'
-shared     generate a shared library
-static     static linking
-rdynamic   export all global symbols to dynamic linker
-r          relocatable output

Debugger options:
-g          generate runtime debug info
-bt N       show N callers in stack traces

F:\>_
```



# Downloading the Compiler

Steps to Download the Dev C/C++.

- Open this link in the browser -  
<https://sourceforge.net/projects/embarcadero-devcpp/>



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# First Program in C

# Writing a 'Hello World' Program

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

# Code Example

C program basically consists of –

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

# Code Example

- *#include <stdio.h>*  
is a preprocessor command,  
which tells compiler to include files  
beforehand.

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

# Code Example

- *int main()* is the main function where the program execution begins.

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

# Code Example

- `/*...*/` will be ignored by the compiler. Such lines are called comments in the program.

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

# Code Example

- *printf(...)* is a function which causes the message "Hello, World!" to be displayed on the screen.

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```



# Code Example

- return 0; terminates the main() function and returns the value 0.

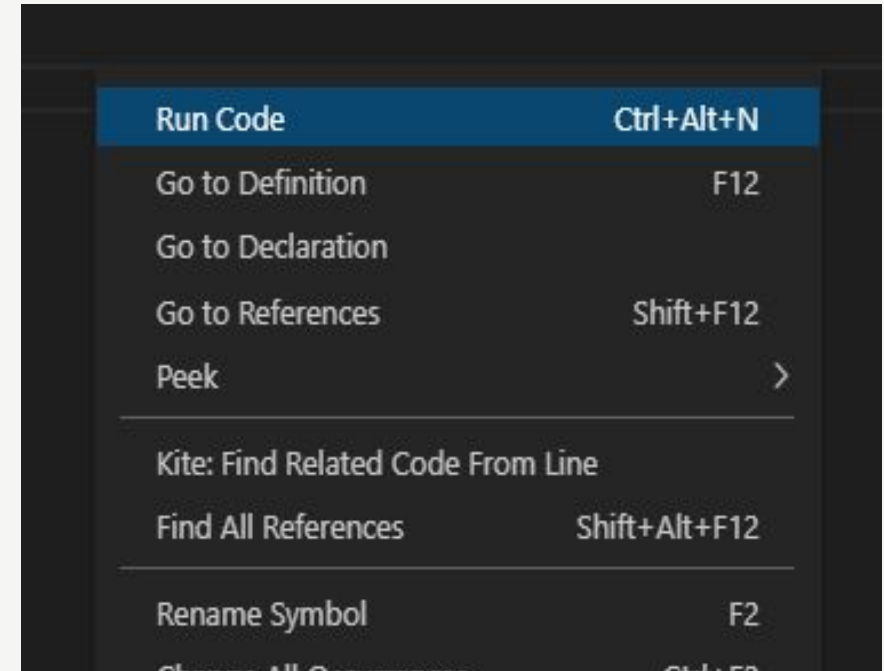
```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

# Code Compilation

- Save the file with .c extension and right click on the VS Code screen to compile.



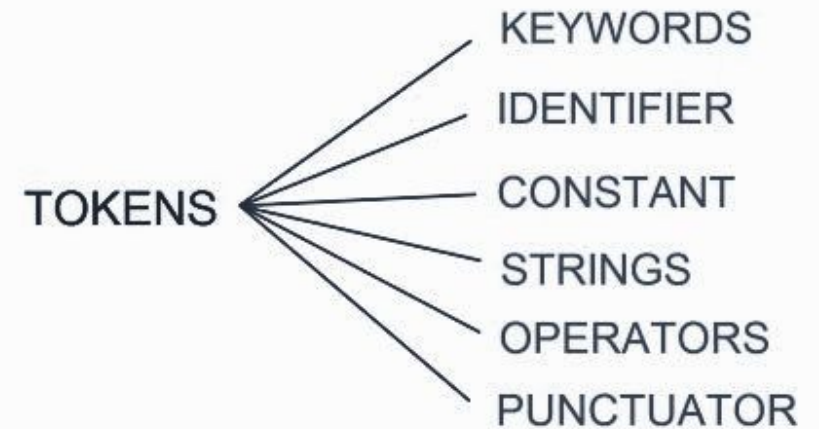


# Basic Syntax

# What are Tokens?

- C program consists of various tokens.

**Ex. - Keyword, Identifier, Constant, String Literal, Operators, or a Symbol.**



# Identifiers

- Identifier is a name used to identify a variable, function, or any other user-defined item.

Ex. - `num1`, `getchar()`, `sum`, `ab_c` etc.

```
num1 = 3;
```

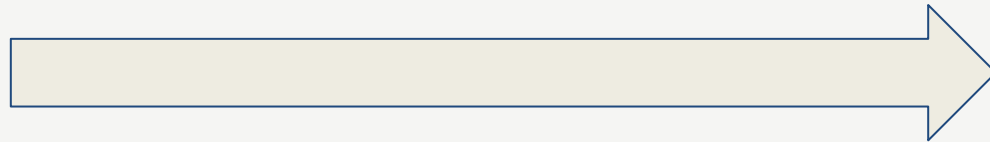
# Identifiers

- Rules for writing the names of Identifiers
  - An identifier starts with a letter A to Z, a to z, or an underscore '\_'.
  - Followed by zero or more letters, underscores, and digits (0 to 9).

```
num1 = 3;
```

# Keywords

- Some Reserved keywords names given in the next slide cannot be used as identifier names.





# Keywords

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



# Strings

- Strings in C are always represented as a set of characters having null character '\0' at the end of the string.

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0



# Operators in C

- Operators in C is a special symbol used to perform the functions.

Ex. - **+, -, \*, /, ==, ++, -- etc**

```
a = b + c;
```

# Special Characters in C

- Various Punctuators are used as a part of C Syntax.

Ex. - [ ], ( ), { }, #, \* etc.

```
return 0;
```

# Constants in C

- A constant is a value assigned to the variable which will remain the same throughout the program

```
const int abc = 5;
```



# Data Types in C

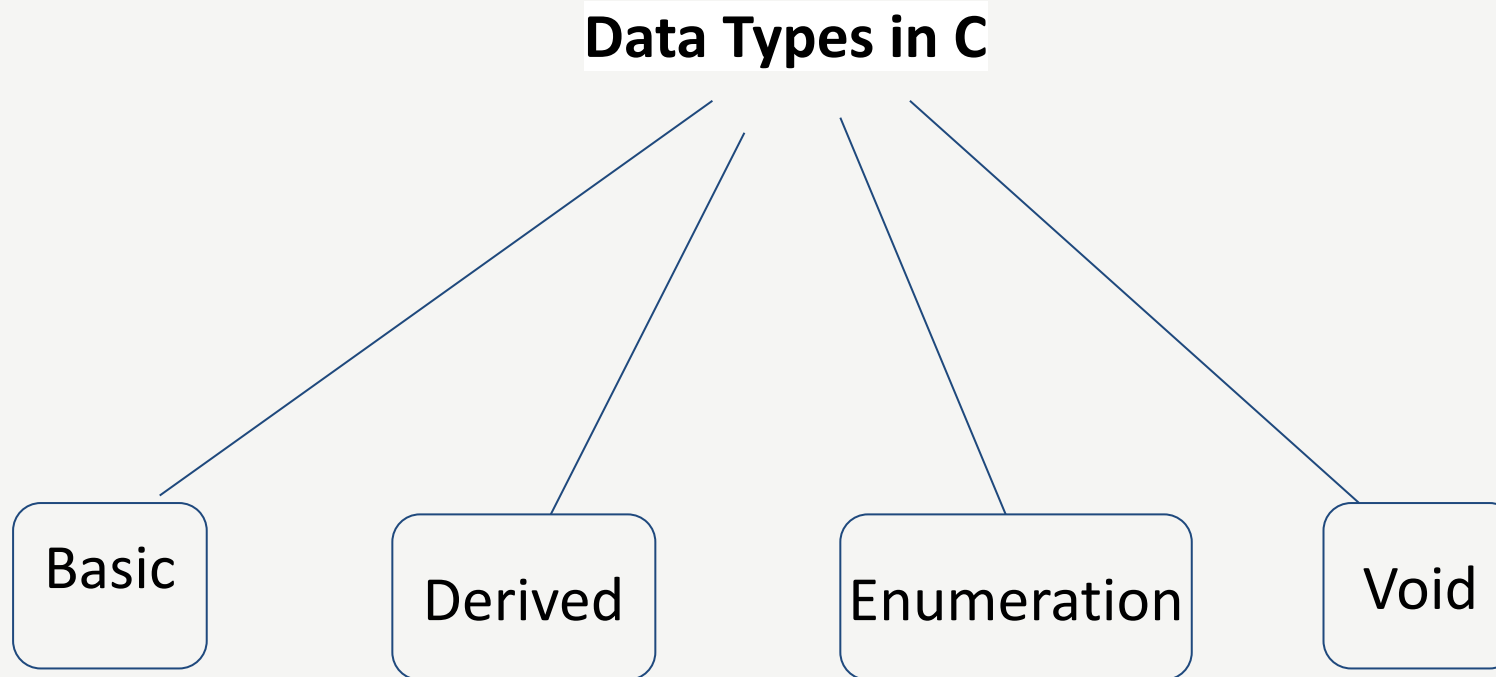
# What are Data Types?

- A data type specifies what type of data a variable can store such as integer, floating, character, etc.

Ex. - **int, float, char** etc.

```
int res = 5;
```

# Types of Data Types



# Basic Data Types

- The basic data types are integer-based and floating-point based.

```
float zab1 = 5.34;
```

```
int res = 5;
```

```
char b = 'G';
```





# Basic Data Types

Data Types	Memory Size	Range
short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10 byte	

# Derived Data Types

- Arrays
- Pointers
- Union
- Structure

grades

90	80	56	100
----	----	----	-----

```
int * p = &n;
```



# Void

- Void is an empty data type that has no value.

```
void num();
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

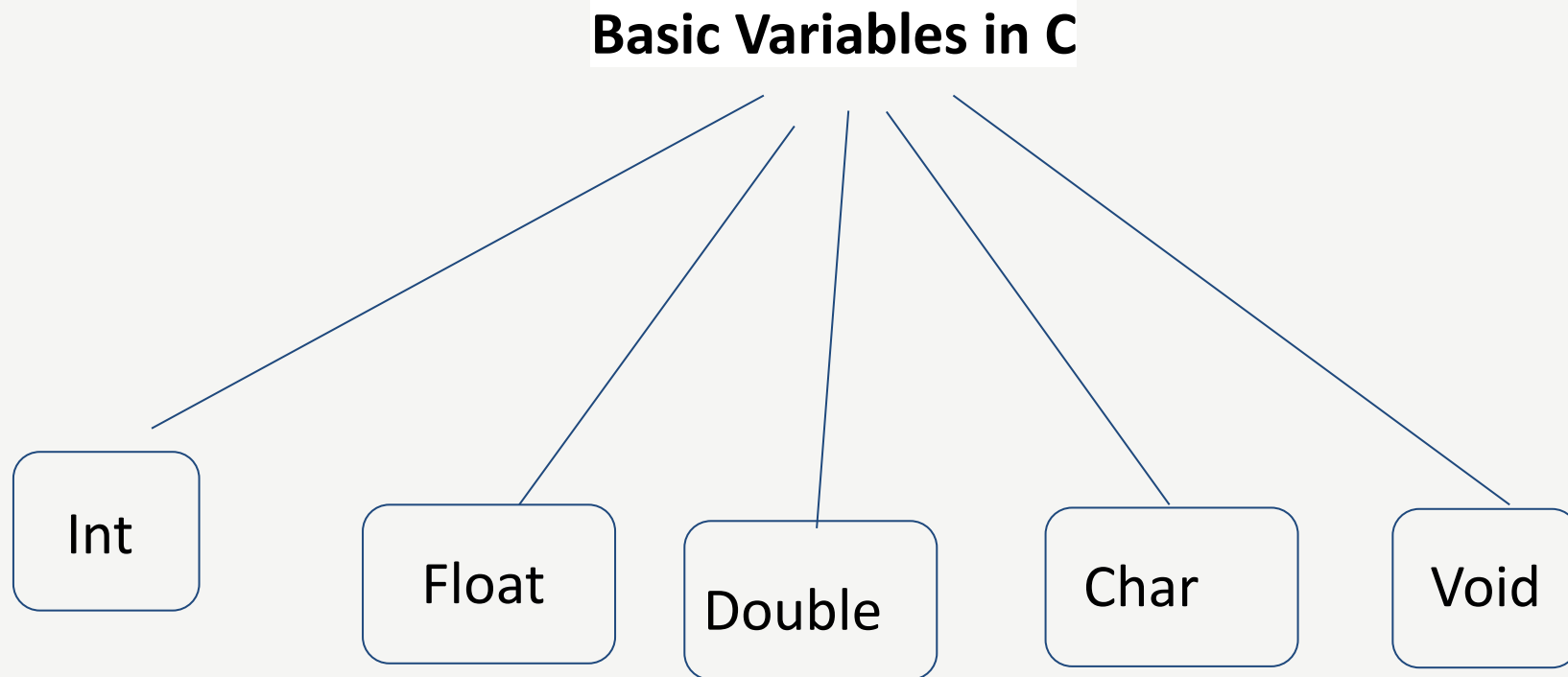
# Variables Used in C

# Deep Dive in Variables

- A variable is a name given to a storage area that our programs can use to store values.
- The name of a variable can be composed of letters, digits, and the underscore character.

```
int res = 5;
```

# Basic Types of Variables



# Variable Declaration

- A variable declaration tells the compiler where and how much storage to create for the variable.

```
float f, salary;  
double d;
```

# Variable Definition

- A variable definition assigns a value in the variable.

```
i = 43;
```





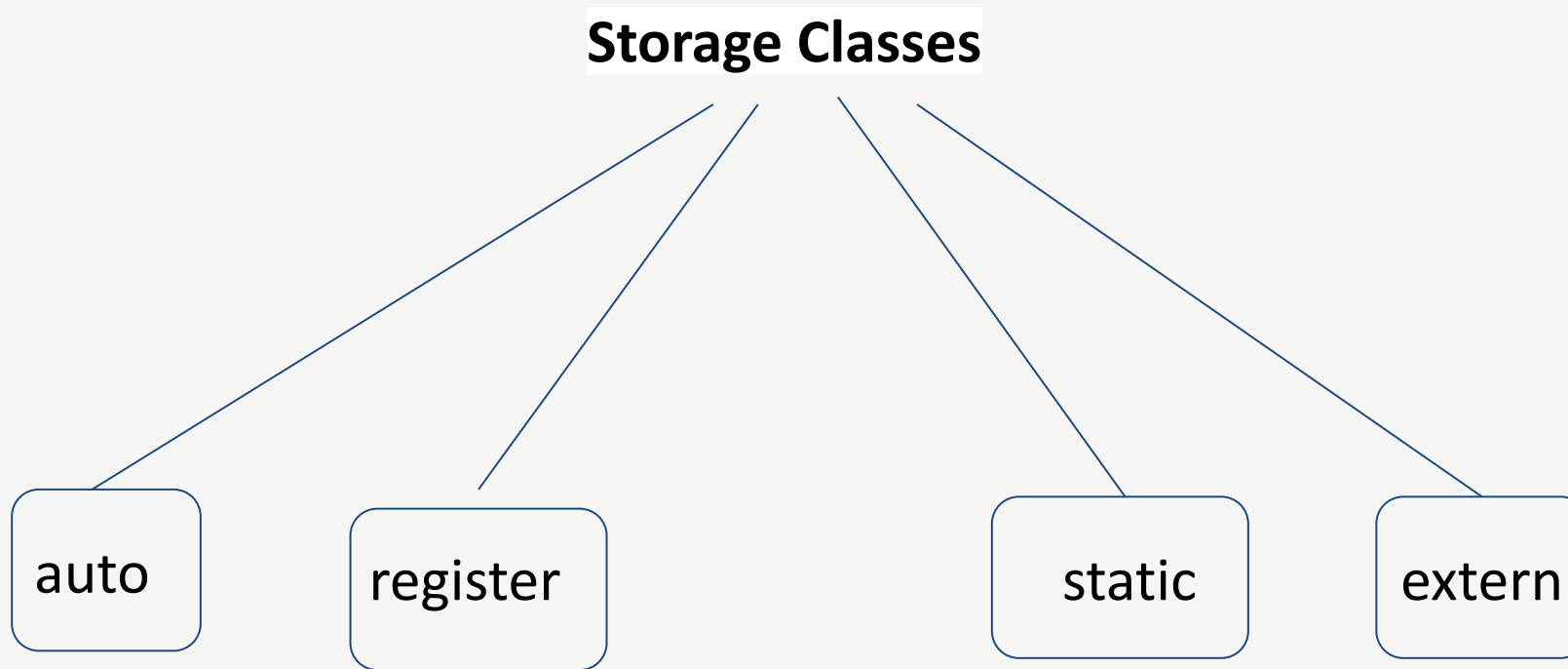
# Storage Classes

# What is a Storage Class?

- Storage class defines the scope (visibility) and lifetime of a variable within a C Program

```
auto int res = 5;
```

# Types of Storage Classes



# Auto Storage Class

- The auto storage class is the default storage class for all local variables

```
int month; auto int days;
```

# Register Storage Class

- When we want a variable to load faster we declare it in register storage class
- it sometimes creates the variable in the CPU register

```
register int zum = 43;
```

# Static Storage Class

- Variables which need to retain their values even outside their scopes are declared under static class

```
static int june = 43;
```

# Extern Storage Class

- Variables which are declared in a scope/block can be used in another block; are created using extern
- extern also helps to share variables from one file to another

```
extern int area;
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# **printf() and scanf()**



# What are printf() & scanf()

- The printf() and scanf() functions are used for input and output in C language.

```
printf("enter a number:");
```

# printf( ) function

- The printf() function is used for output. It prints the given statement to the console.

```
printf("format string",argument_list);
```

# Format Specifiers

- Format specifiers define the type of data to be printed on standard output.

SPECIFIER	USED FOR
%c	a single character
%s	a string
%hi	short (signed)
%hu	short (unsigned)
%Lf	long double
%n	prints nothing

# Format Specifiers

SPECIFIER	USED FOR
%d	a decimal integer (assumes base 10)
%i	a decimal integer (detects the base automatically)
%f	a floating point number for floats
%u	int unsigned decimal
%e	a floating point number in scientific notation
%E	a floating point number in scientific notation
%%	the % symbol

# scanf( ) function

- The scanf() function is used for input. It reads the input data from the console.

```
scanf("format string",argument_list);
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

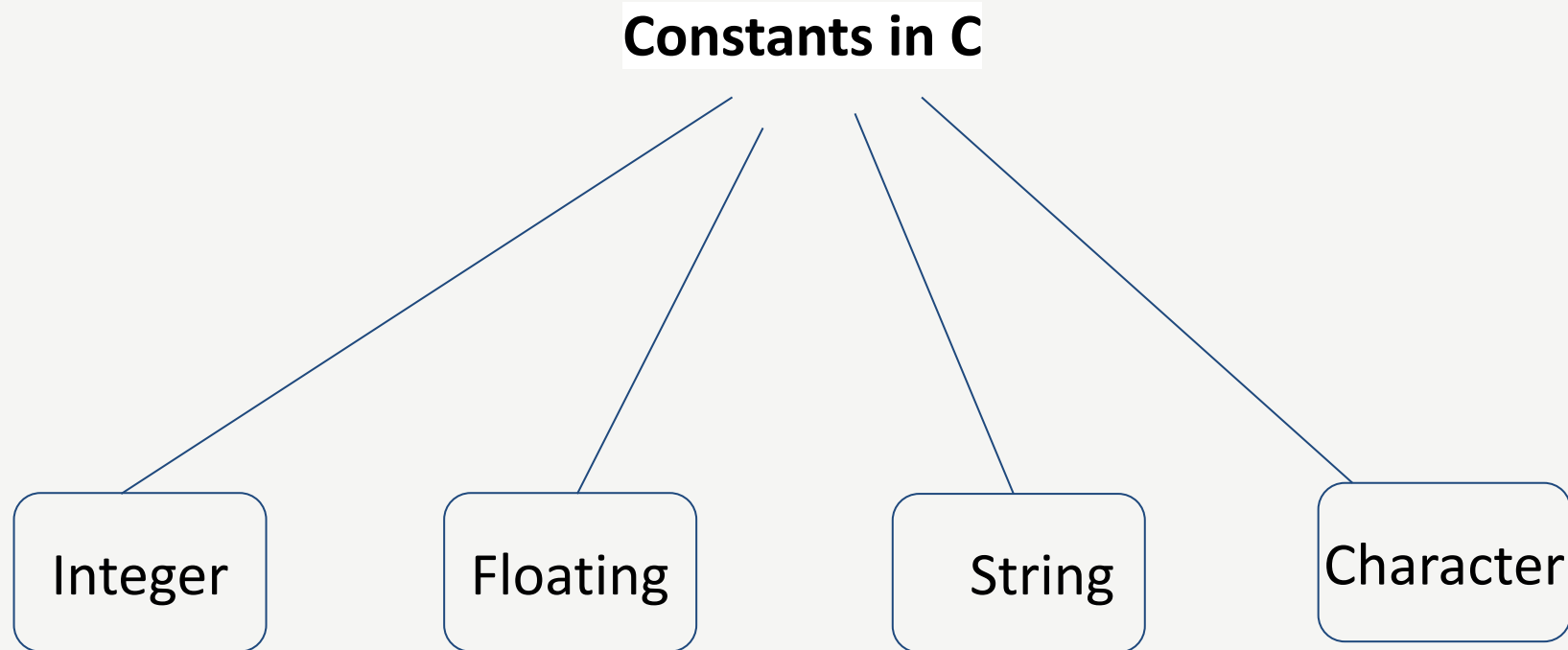
# What are Constants?

# What are Constants?

- Constants are the fixed value in program . Which means that we cannot change it's value.

```
const int res = 5;
```

# Types of Constants in C





# Defining Constants

- Two ways to define a constant-
  - Using `#define`
  - Using `const` Keyword

# #define preprocessor

- Using #define preprocessor
- Syntax : #define variable\_name value

Ex. - **#define salary 40000;**

# const keyword

- Using const keyword
- Syntax : `const data_type variable_name = value ;`

**Ex. - `const int a = 10;`**



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# **Module – 3**

## **[C Language Programming with C]**



**TOPSTECHNOLOGIES**

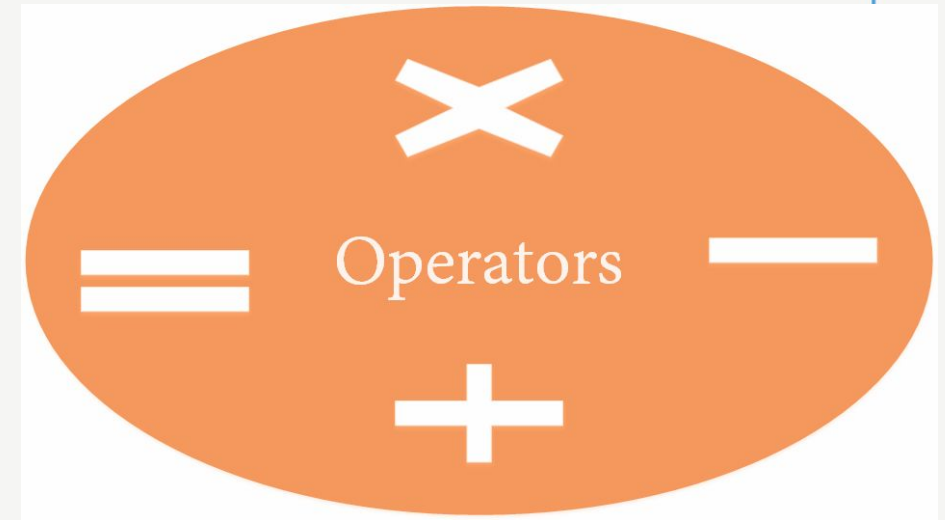
Training | Outsourcing | Placement | Study Abroad

# Operators in C

# What are Operators?

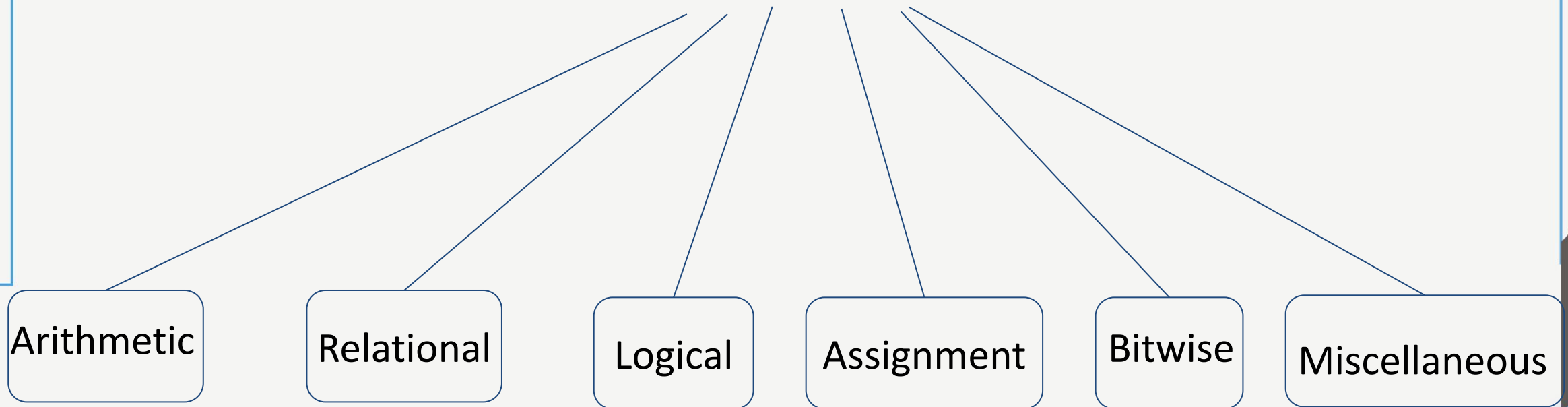
- A symbol that takes one or more operands such as variables, expressions or values and operates on them to give an output.

Ex. - `=, +, -, /, *, ==, ++, --, %, etc.`



# Types of Operators

## Operators in C



# Arithmetic Operators

Operator	Function	Example
+	Addition	var=a+b
-	Subtraction	var=a-b
*	Multiplication	var=a*b
/	Division	var=a/b
%	Modulo	var=a%b
++	Increment	var++
--	Decrement	var--



# Relational Operators

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

# Logical Operators

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5)    (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

# Assignment Operators

- The Basic type of Assignment operator is '='.
- There are other derived operators

Ex. - **\*=, -=, /=, += etc.**

**c += 15;**



**c = c + 15;**

# Bitwise Operators

Operators	Meaning of operators
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

# Miscellaneous Operators

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Decision Making in C

# What is Decision Making?

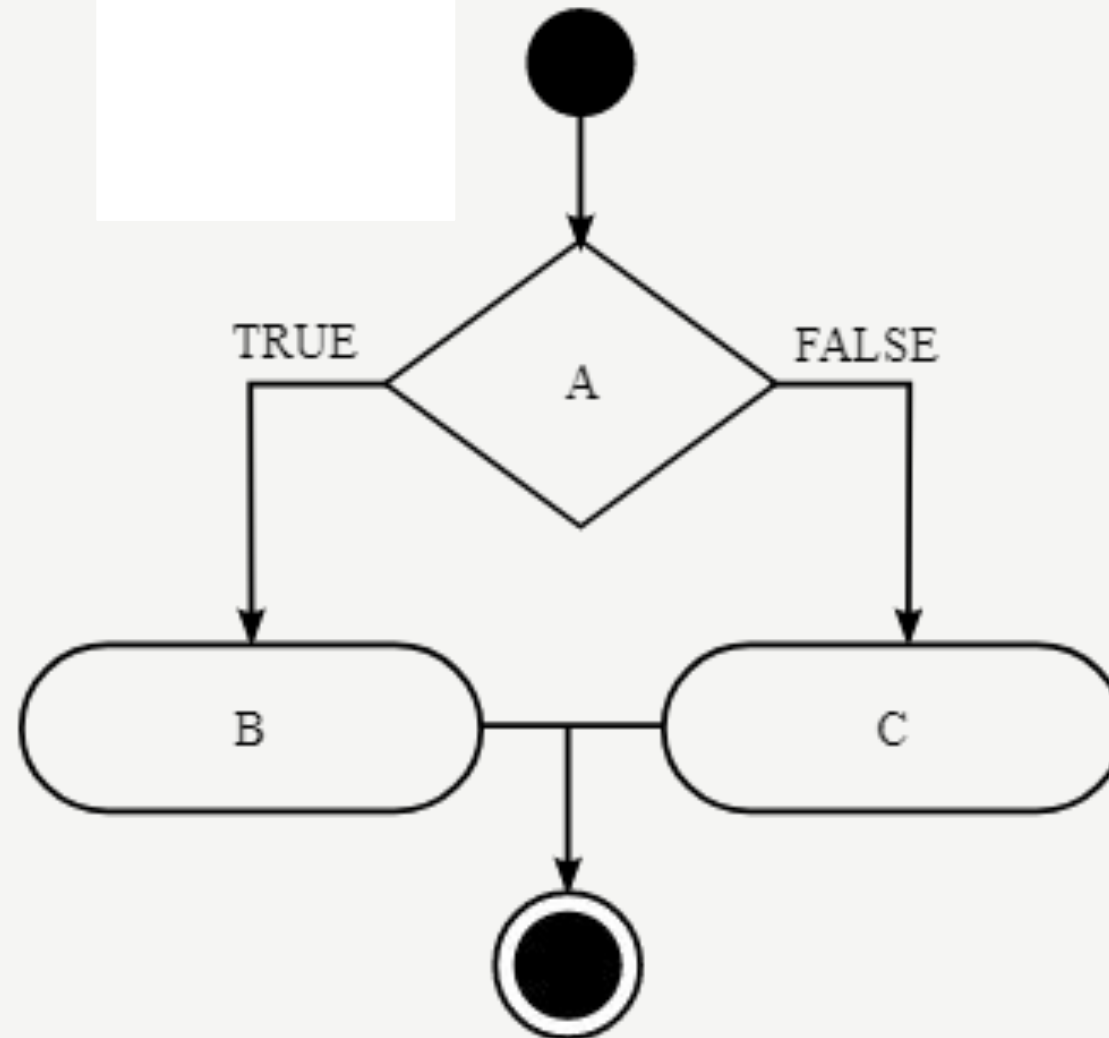
- Real life situations where we have to take condition based decisions by asking 'if' questions.

**Ex. - if age is above 18, I am allowed to drive vehicles or else not allowed.**





# Flow of Decision Making





# Decision Making Statements

- Following types of Statements:
  - if
  - if else
  - Nested if
  - Switch
  - Nested switch

# If Statements

- if statement is the basic decision making statement
- Used to decide whether a certain statement or block of statements will be executed or not

# If Statements

- **Syntax :**

```
if( condition )
```

```
{
```

```
    statement_1 ; // true block statements
```

```
}
```

```
statement x ;
```

# If else Statements

- if else statement allows selecting any one of the two available options depending upon the output of the test condition

# If else Statements

- **Syntax :**  
if (condition )  
{  
    statements; // true statement  
}  
else  
{  
    statements ; // false statement  
}

# Nested If Statements

- Nested if statement is simply an if statement embedded with an another if statement

# Nested If Statements

- **Syntax :**

```
if (condition1)
{
    statements ;           // executes when condition1 is true
    if ( condition2)
    {
        statements ;       // executes when condition2 is true
    }
}
```

# Switch Statements

- Switch case statements are a substitute for long if statements that compare a variable to several integer values



# Switch Statements

- **Syntax :**

```
switch ( n)
{
case 1 :           // executed when n = 1
break;
case 2 :           // executed when n = 2
break ;
default :          // executed when n doesn't match any case
}
```

# Nested Switch Statements

- Nested Switch Statements occurs when a switch statement is defined inside another switch statement.

# Nested Switch Statements

- **Syntax :**

```
switch(ch1)
{
    case 'A': printf ("\n This A is part of outer switch ");
    switch (ch2)
    {
        case 'A ': printf ("\n This A is part of inner switch ");
        break;
        case ' B' :
            }
        break ;
        case ' B' :
    }
}
```



# Loops in C

# What are Loops?

- A loop statement allows us to execute a statement or group of statements multiple times based on a condition

**Ex. - printing 1 to 100 on the output screen**



# Types of Loops

- **Entry Controlled loops-** A condition is checked before executing the loop. It is also called as a pre-checking loop.
- **Exit Controlled loops-** A condition is checked after executing the loop. It is also called as a post-checking loop.

# Entry controlled Loops

1. **For Loops-** It is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
2. **While Loops-** It repeatedly executes a target statement as long as the given condition is true.

# For Loops

- Used to efficiently write a loop that needs to execute a specific number of times.

## Syntax :

```
for ( initialization ; test condition ; increment )  
{  
    Body of loop  
}
```

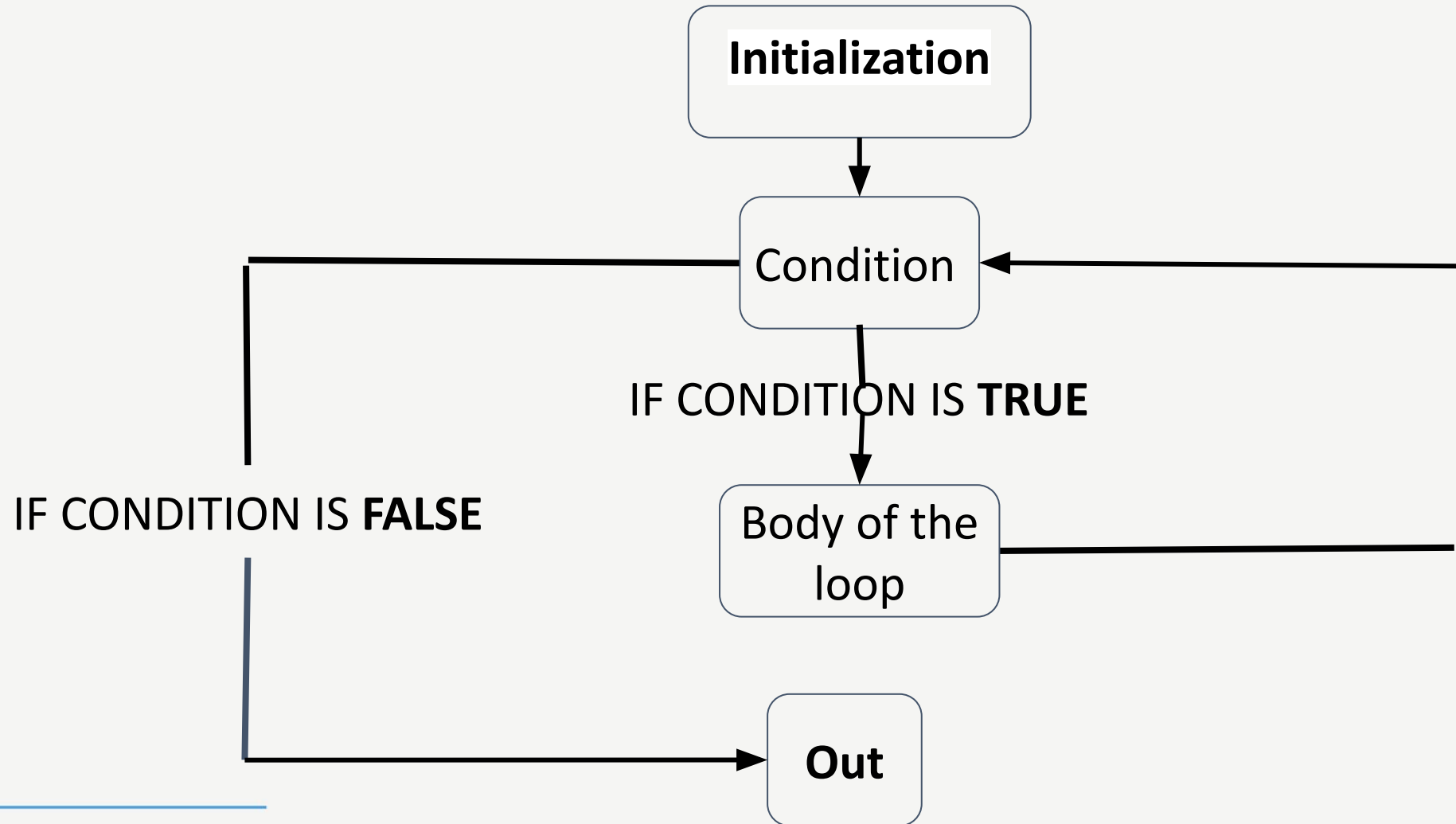


# For Loops

- All for loops are executed in following sequence :
  - Step 1 : It executes initialization statements
  - Step 2 : It checks the condition;
    - if true then go to Step 3
    - other wise Step 4
  - Step 3: Executes loop and go to Step 2
  - Step 4 : Out of the Loop



# For Loop Flow Chart



# While Loops

- It repeatedly executes a target statement as long as the given condition is true .

## Syntax :

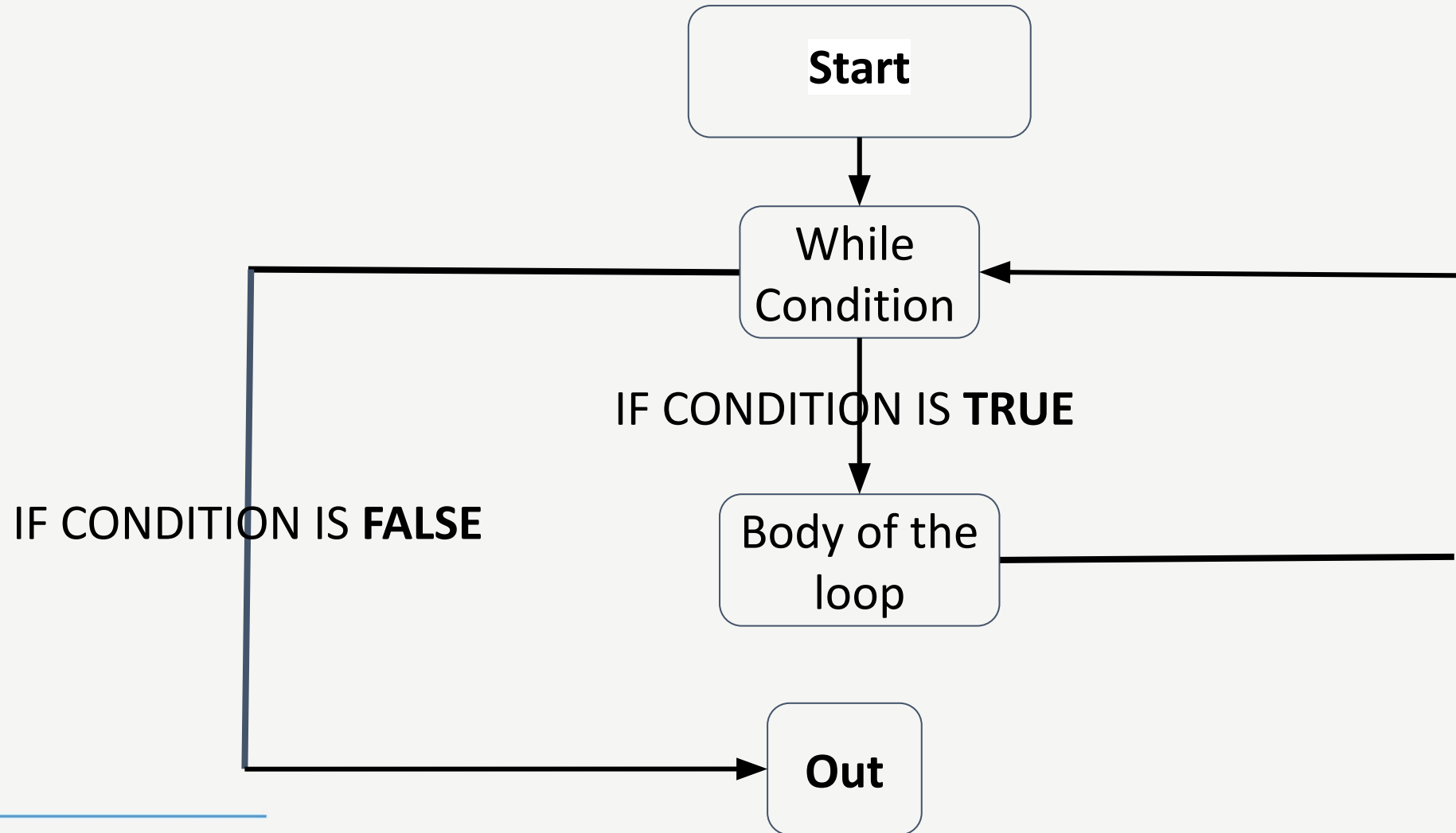
```
while (condition )  
{  
    Body of loop  
}
```

# While Loops

- All while loops are executed in following sequence :
  - Step 1 : It checks the test condition
    - if true then go to Step 2
    - other wise to Step 3
  - Step 2 : Executes body of loop and go to Step 1.
  - Step 3 : Other statements of program.



# While Loop Flow Chart



# Exit controlled Loops

1. **Do-while Loops-** Do-while loop is similar to while loop , except the fact that it will execute once even if condition is false.

# Do-while Loops

- **Syntax :**

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

# Do-while Loops

- All do while loops are executed in following sequence :

Step 1 : Executes the body of loop and go to Step 2 .

Step 2 : It checks the test condition

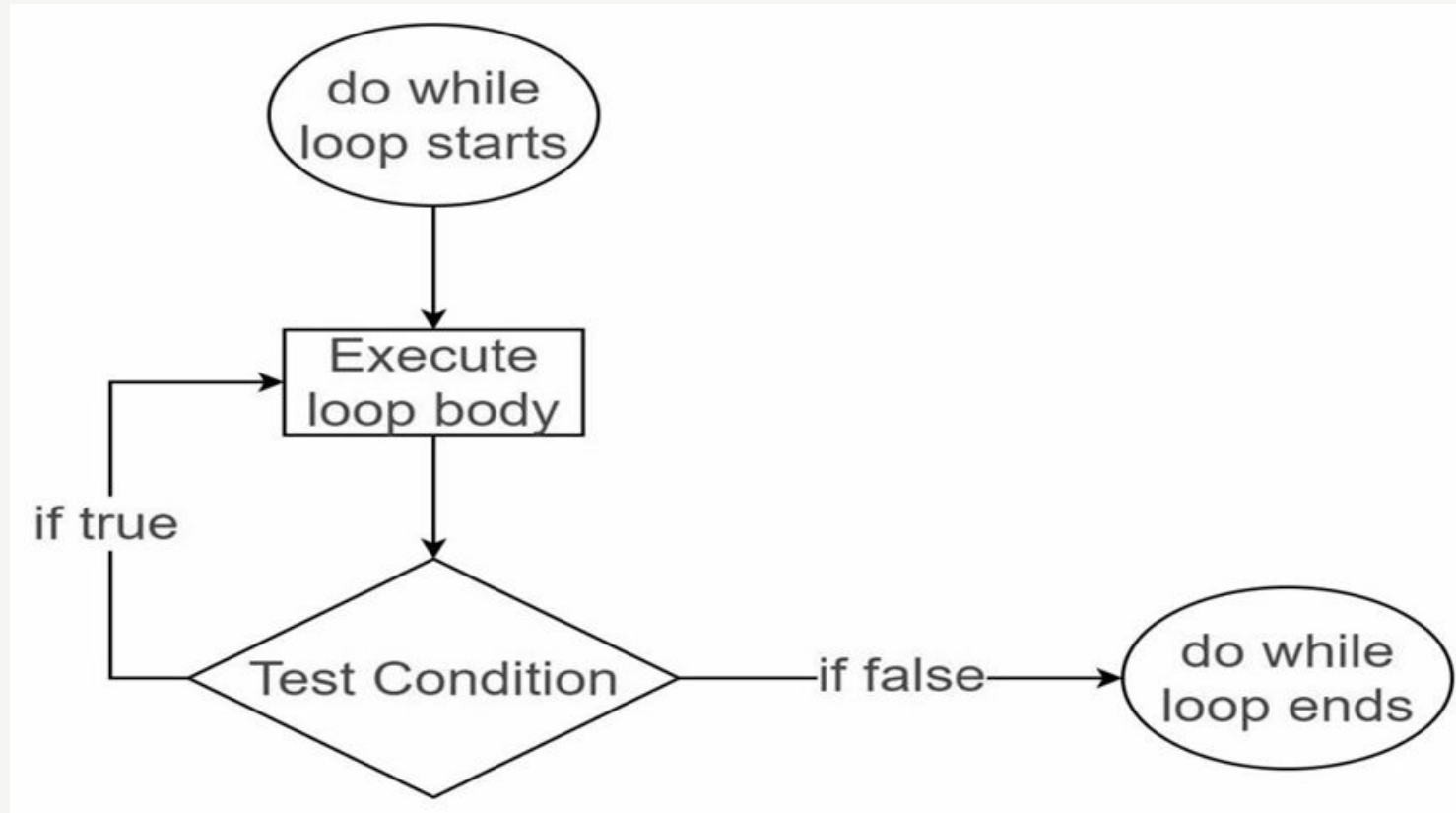
if true then go to Step 1

otherwise go to Step 3.

Step 3: Other statements of the program .



# Do-while Loops



# Nested Loops

- In C we can use one loop inside another loop .

- **Syntax:**

```
for(initialization ; test condition ; increment )  
{  
    for(initialization ; test condition ; increment )  
    {  
        statements ;  
    }  
    statements ;  
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Function and Parameters

# Types of Parameters

## Parameter in Functions

Call by  
Value

Call by  
Reference

# Call by Value

- A method of passing parameters, where it copies the actual value into formal parameter
- Changes made to the parameter inside the function have no effect on the actual parameters

# Call by Reference

- A method of passing arguments which copies the address of an argument into formal parameter.
- Changes made to the parameter affect the passed argument.



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Working with Functions

# What are Functions?

- A function is a set of statements that take inputs, do some specific computation and produces output

Ex. - `main( ), sum( ), swap( )` etc.

```
int num1( );
```



# Parts of Functions

- **FUNCTIONS-**
  - Function Definition
  - Function Call
  - Function Declaration

```
int num1( );
```

# Functions Definition

- Syntax of Function Definition-

```
return_type function_name( parameter list )  
{  
    body of the function  
}
```

# Components in Definition

**Return\_type** : It is data type of value which function will return. If function does not return any value data type will be void().

```
return_type function_name( parameter list )  
{  
    body of the function  
}
```

# Components in Definition

**Function\_name** : It is the name given to the function by the programmer.

```
return_type function_name( parameter list )  
{  
    body of the function  
}
```

# Components in Definition

**Parameter list :** This list refers to type, order and number of parameters of the function. A function can have no parameters also.

```
return_type  function_name( parameter list )  
{  
    body of the function  
}
```

# Components in Definition

**Body of function** : It is collection of statements that define the working of the function.

```
return_type function_name( parameter list )  
{  
    body of the function  
}
```

# Function Declaration

- It tells the compiler about the function name and how to call the function.

- Syntax :

```
return_type function_name ( parameter_list ) ;
```

# Function Declaration

- Only type is required in function declaration , we can skip the parameter name
- Example:  
`int add ( int , int );`



# Function Call

- To use a function, we have to call that function to perform the given task.

# Function Call

- When a program calls a function, the compiler gets redirected towards the function definition
- Function Call simply pass the required parameters along with the function name

# What are Function Parameters

- Parameters are the variables that are taken as input to perform the function

```
int max(int num1, int num2);
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Scope of a Variable

# What is Scope?

- Scope is the part of the program where a defined variable can have its existence and beyond that it cannot exist

# Three Scopes

- **Three places where variables can be declared-**
  - Local Variables : Declared inside a block
  - Global Variables : Declared outside all functions
  - Formal Parameter : Declared in function definition

# Local Variables

```
#include<stdio.h>

void main ()
{
    int x , y , z ;      // local variable
    x=10 ;
    y =20 ;
    z =x + y ;
    printf(" %d  %d  %d " , x, y, z ) ;
}
```

# Global Variables

```
#include<stdio.h>

int z;                      // global variable

void main ()
{
    int x , y ;             // local variable
    x=10 ;
    y =20 ;
    z =x + y ;
    printf(" %d  %d  %d " , x, y, z ) ;
}
```





# Formal Parameters

```
void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Array of Pointers

# What is an Array of Pointers?

- An array where the elements would be pointers that will store addresses of some other variables.
- **Syntax:**

```
int *ptr[MAX];
```

# What is an Array of Pointers?

- with the help of array of pointers we can use it to point to each element of another array

# Pointer to Pointers

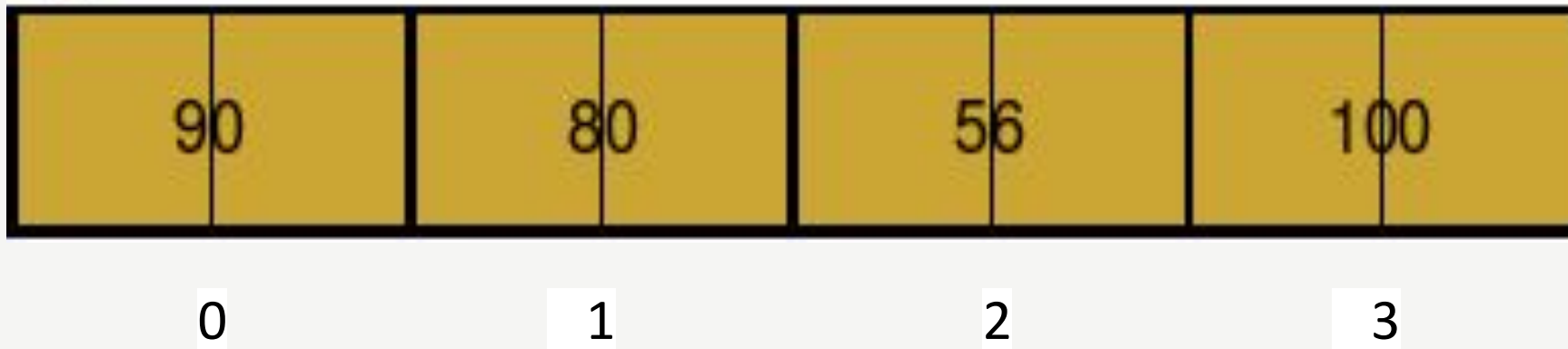
- Concept where a pointer stores the address of another pointer in it, which eventually is pointing to another variable
- **Syntax:**

```
int **var;
```

# Arrays

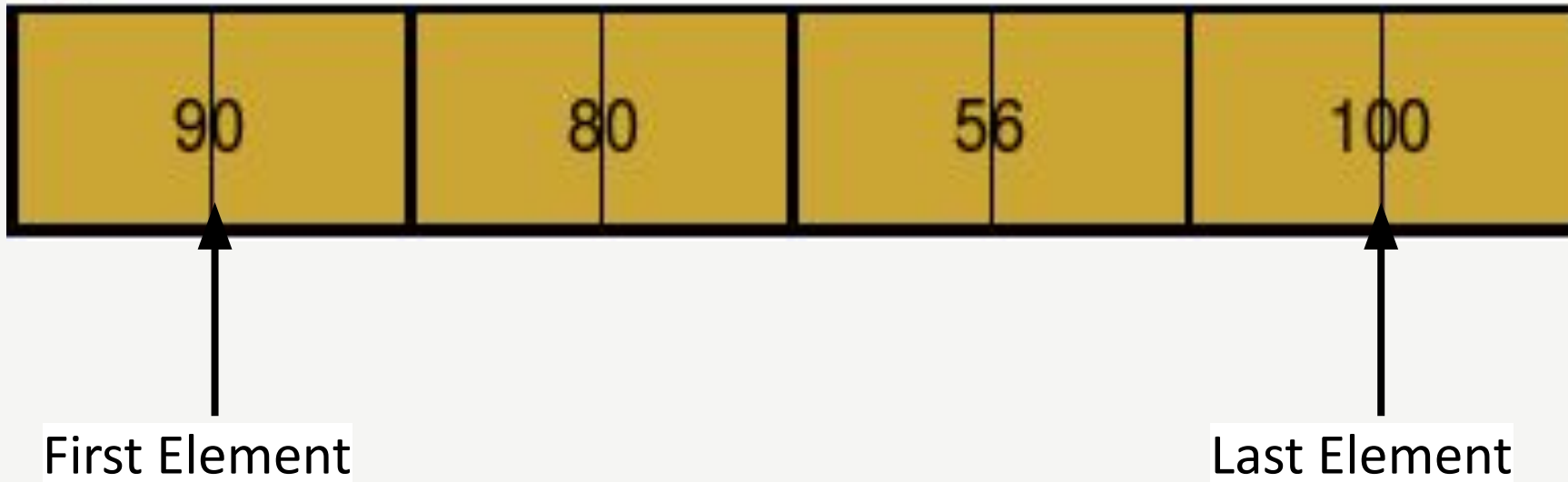
# What are Arrays?

- An array is used to store a collection of data, and it is often used as a collection of variables of the same type.



# What are Arrays?

- All arrays consist of contiguous memory locations.





# Declaring Arrays

- In declaration we specify the type of element and size of the array element

- **Syntax :**

`data_type array_name [ size ] ;`

**Ex. -** `int roll[20] ;`

# Initializing Arrays

- We can initialize an array in C either one by one or using single statement

Ex. - `double balance [ ] = { 1000.0, 2.0, 3.4, 7.0, 50.0};`

OR

`balance[4] = 50.0 ;`



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Accessing Arrays

# Accessing Array Elements

- An element is accessed by placing the index of the element within the square brackets after the name of the array

Ex. - `double income = balance[9];`

# Accessing Struct Members

- The individual members of structure can be accessed using the member access operator (.)
- It connects the structure variable and the structure member.
- **Syntax:**  
`structure_variable.structure_member`



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Types of Arrays

# Types of Arrays

- Single/ One Dimensional Array
- Multi Dimensional Array



# Single Dimensional

```
int roll[20] ;
```





**TOPS**TECHNOLOGIES

Training | Outsourcing | Placement | Study Abroad

# Multi Dimensional

```
int roll [20][12]...;
```

# Multi Dimensional

- C programming language provides us multi-dimensional array.
- Out of which we will discuss Two Dimensional Array



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Multidimensional Arrays

# What are they?

- Multidimensional arrays are arrays of arrays
- **General Form-**

```
data_type array_name[size1][size2]....[sizeN];
```



# Example

```
int x[2][3][4] =  
{  
    { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },  
    { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }  
};
```



**TOPSTECHNOLOGIES**

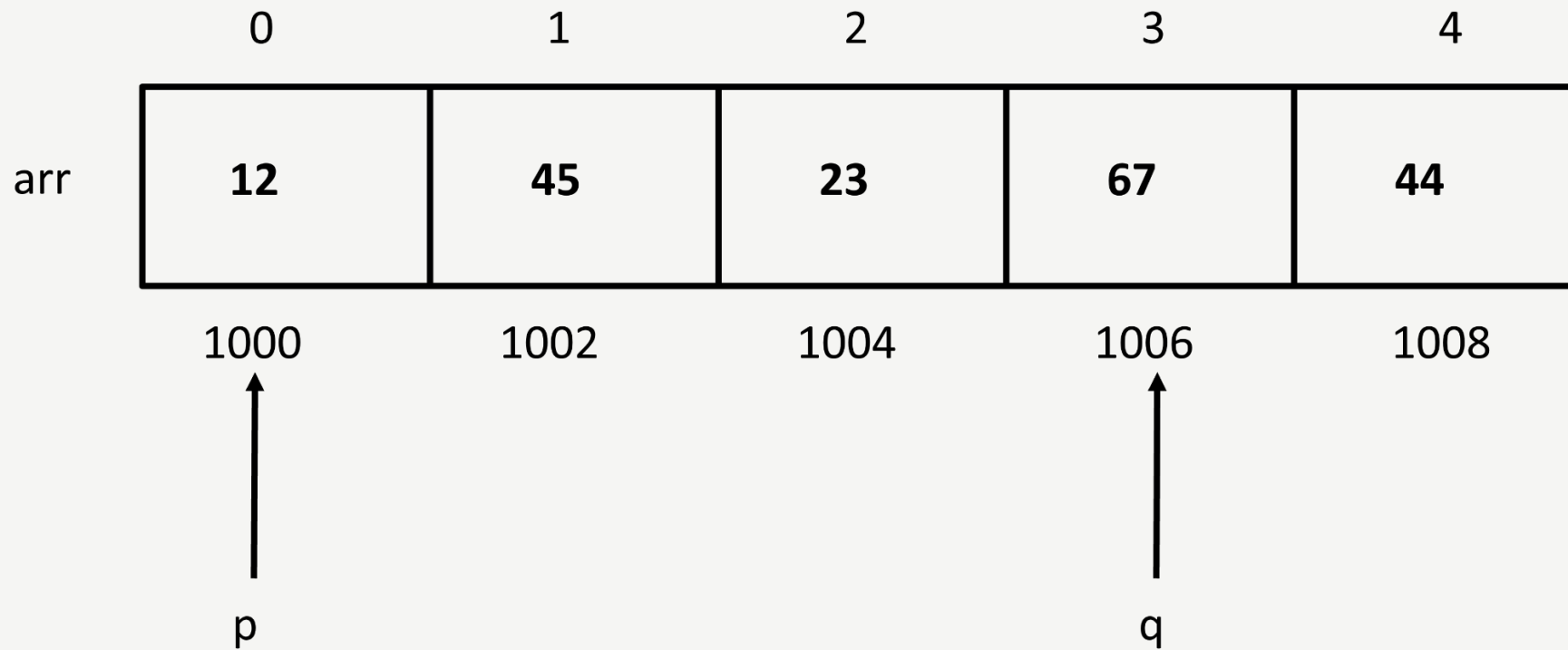
Training | Outsourcing | Placement | Study Abroad

# Pointer Arithmetic

# Pointer Declaration

- ```
int main()  
{  
    int arr[5] = {12, 45, 23, 67, 44};  
    int *p, *q;  
  
    p = &arr[0];  
    q = &arr[3];  
}
```

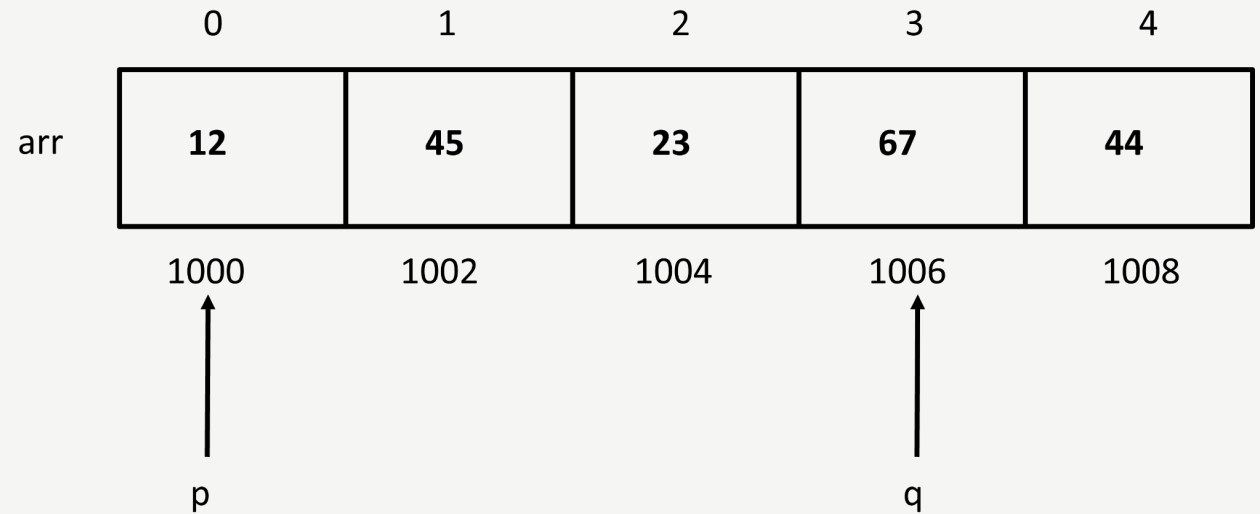
# Representation of Array





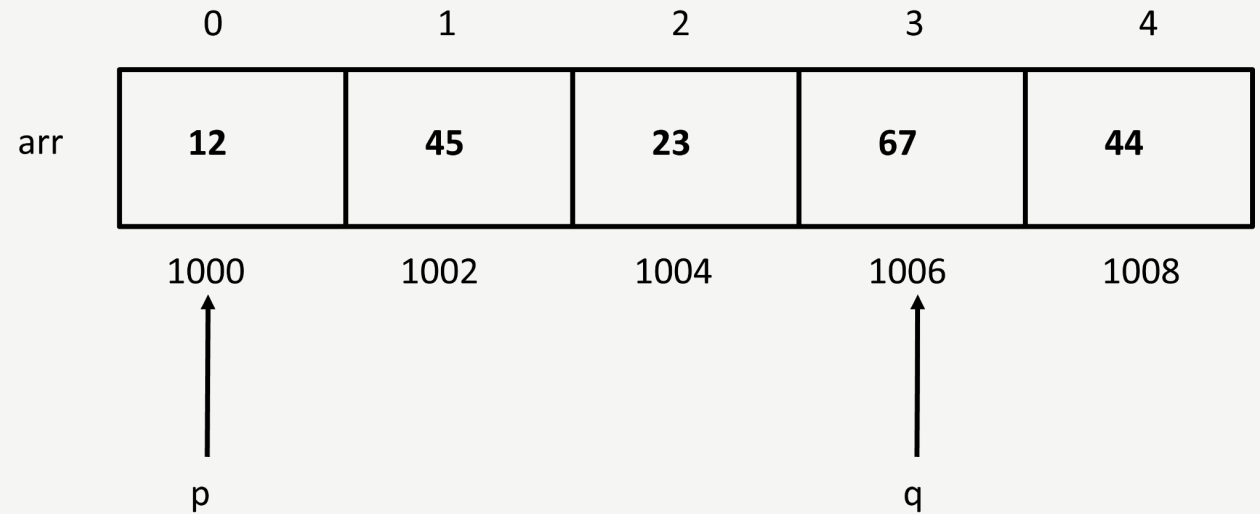
# Pointer Increment

**p++;**



# Pointer Decrement

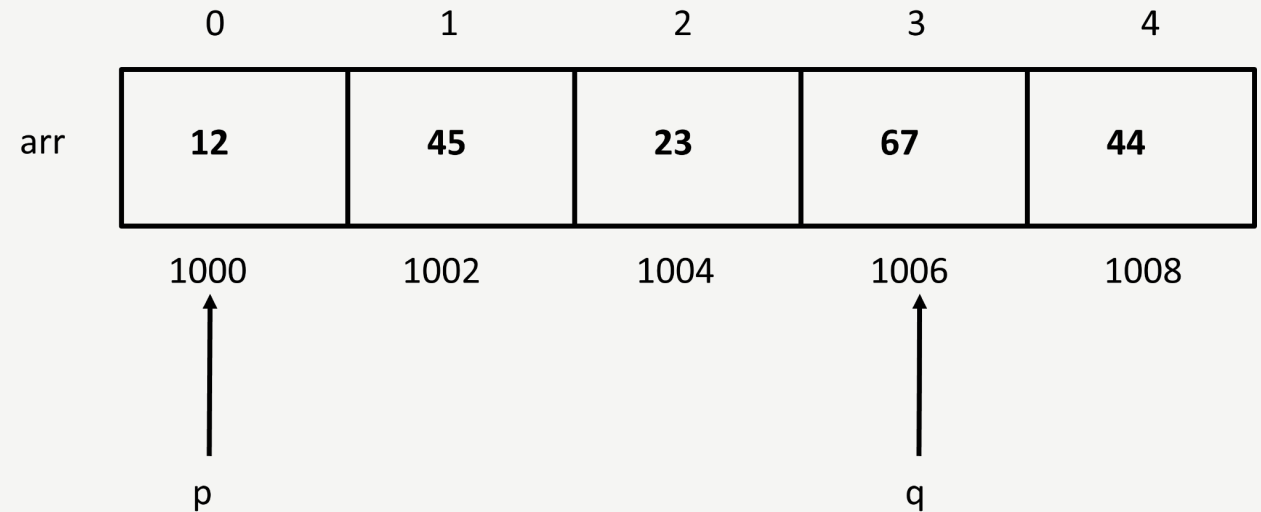
**q--;**



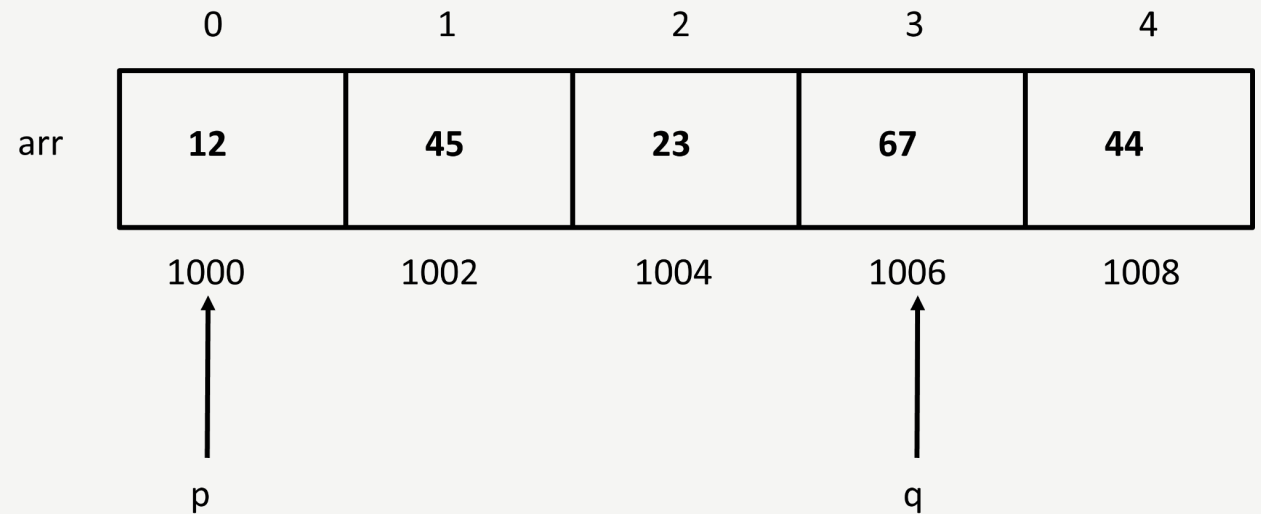
# Constant Addition/Subtraction

**$p = p + k;$**

**$q = q - k;$**



```
l = q - p;
```





# Pointers

# What are Pointers?

- Pointers are the variables that contain the address of the another variable within the memory
- **Syntax :**  
`data_type *pointer_name ;`

# Steps to use Pointers

- Define a pointer variable
- Assign the address of variable to the pointer
- Access the value at the address available in the pointer variable

# Pointer Declaration and Initialization

```
int roll = 7 ;
```

```
int *optr = &roll;
```



Type of the variable whose address is to be stored





# Strings

# What are Strings?

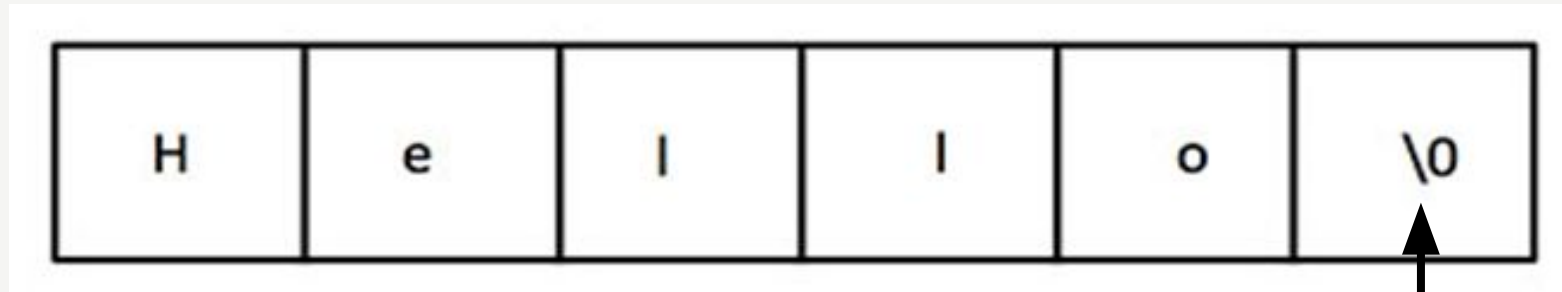
- A string is an array of characters stored in a consecutive memory locations
- The ending character is always the null character '\0'. It acts as string terminator.

- **Syntax :**

```
char string_name [ length ] ;
```

# Strings

- The compiler automatically places the '\0' at the end of the string when we initializes the array



Null Character

# String Functions

| Function | What It Does                                                    |
|----------|-----------------------------------------------------------------|
| strcpy() | Copies one string to another.                                   |
| strlen() | Returns the length of a string, not counting the NULL character |

# String Functions

| Function | What It Does                                                        |
|----------|---------------------------------------------------------------------|
| strcmp() | Compares two strings. If the strings match, the function returns 0. |
| strcat() | Appends one string to another, creating a single string out of two. |



# Structures used in C

# What are Structures?

- It is a collection of logically related data items of different data types grouped under a single name
- It is a user defined data type

# Structure Definition

- The structure definition template is terminated with a semicolon
- Members of the structure are enclosed in { }

- **Syntax :**

```
struct structure_name
{
    data_type  member1;
    data_type  member2;
    .....
};
```





# Unions

# What are Unions?

- Union can use same memory for multiple purposes
- Union can store different data types in the same memory location
- Union have many members, but only 1 member can have a value at a time

# Union Definition

- We start with Union keyword
- The union definition is similar to structures except the keyword

- **Syntax :**

```
union union_name
{
    data_type member1;
    ..      ..      ..
} var1,var2,... ;
```

# Accessing Members of union

- With the help of member access operator ( . ) , we can access any member of the union
- **Syntax :**  
`union_variable.union_name`



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Typedef Function

# What is Typedef?

- typedef is used to provide some meaningful names to the already existing datatypes
- **Syntax :**  
`typedef existing_name alias_name;`



# What is Typedef?

- Example:

```
typedef int vector ;
```

- Here we declare vector as a new name to the type int with the help of typedef .

# What is Typedef?

- Now we can create variables of type vector in the following way :

**vector x, y, z;**

- instead of :

**int x, y, z ;**



# typedef vs #define

- typedef is limited to giving names to types only.
- #define can also be used to define other things; like you can define 2 as Two

# **Module – 4**

## **[C,C++ – File and Error Handling, Debugging]**



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# File I/O Functions

# fprintf()

- similar to printf() function where it helps to print formatted output on the file



# fscanf()

- similar to scanf() function where it helps to consider formatted input from the file

# rewind()

- It refreshes the file stream and bring it to the start of the file



# fseek()

- It brings the file pointer to a specific location in the file



# ftell()

- It prints the specific location of the file pointer





# getw()

- It is used to extract the integer from the file



# putw()

- It is used to put an integer to the file



# feof()

- Used to check whether the file pointer has reached the end of the file



# I/O on Files

# I/O on Files

- A file represents a sequence of bytes, regardless of it being a text file or a binary file
- Through C, we can converse with external text files by input and output functions

# Opening Files

- We use fopen() function to create a new file or to open an existing file
- **Prototype:**

```
FILE *fopen ( const char * filename, const char *mode );
```

# Opening Files

- File can be opened in various modes-
  - **r** : Opens an existing text file for reading purpose
  - **w**: Opens a text file for writing. If does not exist a new file is created

# Opening Files

- `w+` : Opens a text file for both reading and writing . It first truncates the file to zero length if it exists , otherwise creates a file.
- `a` : Opens a text file for writing in appending mode.
- `r+` : Opens a text file for both reading and writing .



# Closing Files

- To close a file, we will use `fclose( )` function
- **Prototype:**

```
int fclose ( FILE *fp) ;
```

# Writing a File

- The simplest function to write strings to stream is:

```
int fputs ( const char *s , FILE *fp ) ;
```

# Reading a File

- The simplest function to read strings from stream is:  
`char *fgets(char *buf, int n , FILE *fp );`

# Input and Output

- C programming treats all the devices as files
- Few files are automatically opened when a program executes to provide access to the keyboard and screen

# Input and Output

- The file pointers are the means to access the file for reading and writing purpose .

| Standard File   | File Pointer | Device   |
|-----------------|--------------|----------|
| Standard input  | stdin        | Keyboard |
| Standard output | stdout       | Screen   |

# getchar and putchar

- **getchar( )**: It reads only single character at a time and is used in the loop when we want to read more than one character
- **putchar( int c )**: It returns the a single character at a time and is used in the loop when we want to display more than one character on the screen.

# gets and puts

- **char \*gets(char \*s )**: This function reads a line from stdin until EOF or new line is encountered
- **int puts (const char \*s)** : This function writes the string to stdout.



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Preprocessor & Header File



# What is a Preprocessor?

- C Preprocessor is just a tool and it instructs the compiler to do required pre-processing before the actual compilation

# Preprocessor Directive

| SR.NO | DIRECTIVE AND DESCRIPTION                                |
|-------|----------------------------------------------------------|
| 1.    | #define – Defining with alias                            |
| 2.    | #undef – Undefines the #define                           |
| 3.    | #include – Inserts a particular header from another file |

# Header Files

- A header file is a file with extension .h which contains C function declarations and macro definitions
- it is imported by using #include command
- We can also create our own header files



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Handling the Errors

# What is an Error?

- Whenever we make a mistake in the code, and compiler doesn't understand it, it gives an error

**Ex. Syntax Error, Logical Error etc.**

# What is an errno?

- errno is a global variable, which can be used to identify which type of error was encountered during function execution, based on its value.

# **perror( ) & strerror( ) Function**

- perror() and strerror() functions are used to display the text message of errors generated

# **perror( ) Function**

- The perror() function displays the string we pass to it, followed by colon and then the text of the current errno value



# strerror( ) Function

- strerror() function returns a pointer to the textual representation of the current errno value



# Recursive Functions

# What is Recursion?

- It is the process by which a function calls itself repeatedly, until some specific condition has been satisfied
- This process is used in place of repetitive computations in which each action is stated in terms of a previous result.

# Recursive Function

- **Syntax :**

```
data_type function_name ( argument list )  
{  
    .....  
    function_name(. . .) ;  
    .....  
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Module – 5

## [C,C++ – C++ (Basics of C++) ]

# Introduction to C++

C plus plus written as C++ is a general-purpose programming language which was developed as an extension of the existing C language to which includes object-oriented concepts.

C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories USA in early 1980's, as an extension to the C language.

The language was updated 4 major times in 2011, 2014, 2017, and 2020 to C++11, C++14, C++17, C++20.

It is a middle-level language which has an advantage of programming low-level like : drivers and kernels apart from this it also support even higher-level applications like: GUI, games, desktop applications etc.

Syntax and code structure of both C++ and C are almost equivalent.

# Advantages of C++

It is most popular programming languages.

Various operating systems use C++ for development, GUI, and also for embedded systems.

It is an object-oriented programming language which provides accurate structure to programs and also allows code to be reused, which drastically lowers the cost of development.

It is portable and hence used for developing applications which can adapt to various platforms.

# **Procedure Oriented Programming & Object Oriented Programming**

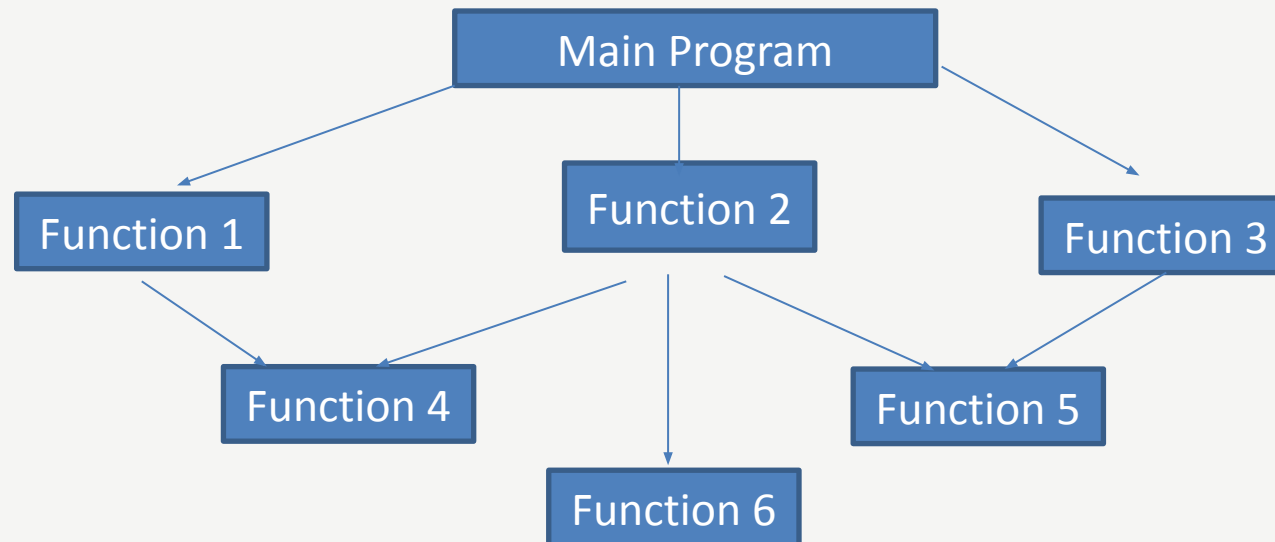


# Procedure Oriented Programming

In Procedure Oriented programming , the problem is viewed as sequence of things to be done such as reading, calculating and printing.

The number of functions are return to accomplish such task, i.e. the focus is on functions.

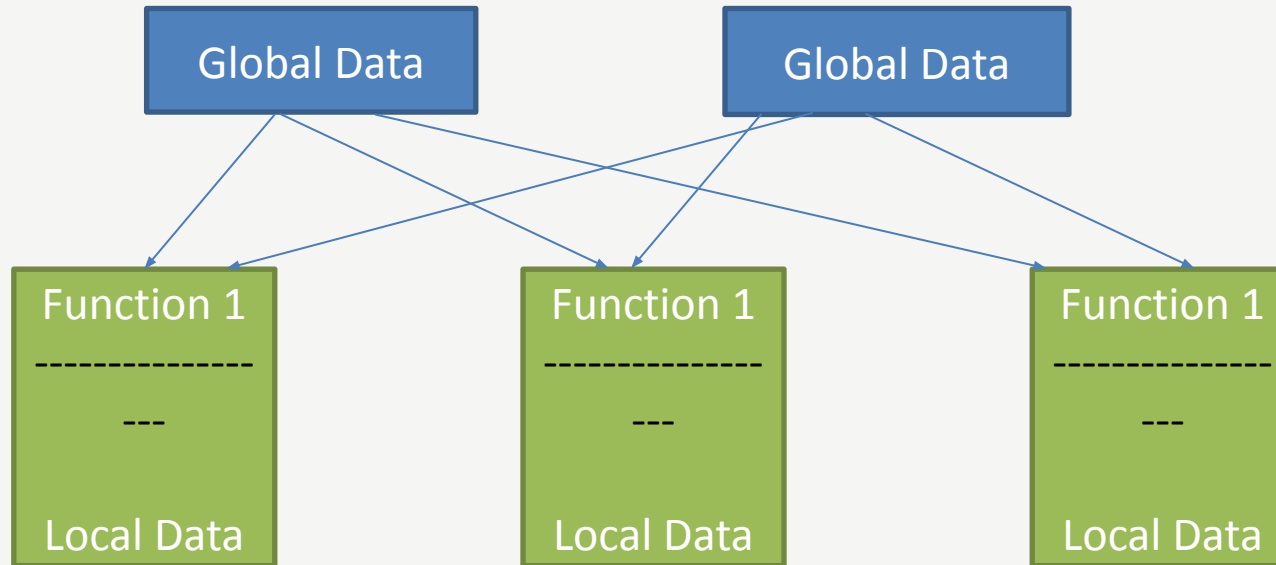
The typical program structure for procedure programming is shown below :



# Procedure Oriented Programming

Characteristics of Procedure Oriented Programming.

- ❖ Emphasis is on doing things
- ❖ Large programs are broken into small known as functions
- ❖ Most of the function share global data.
- ❖ Data move openly around the system from function to function.
- ❖ Follows Top Down approach in program design



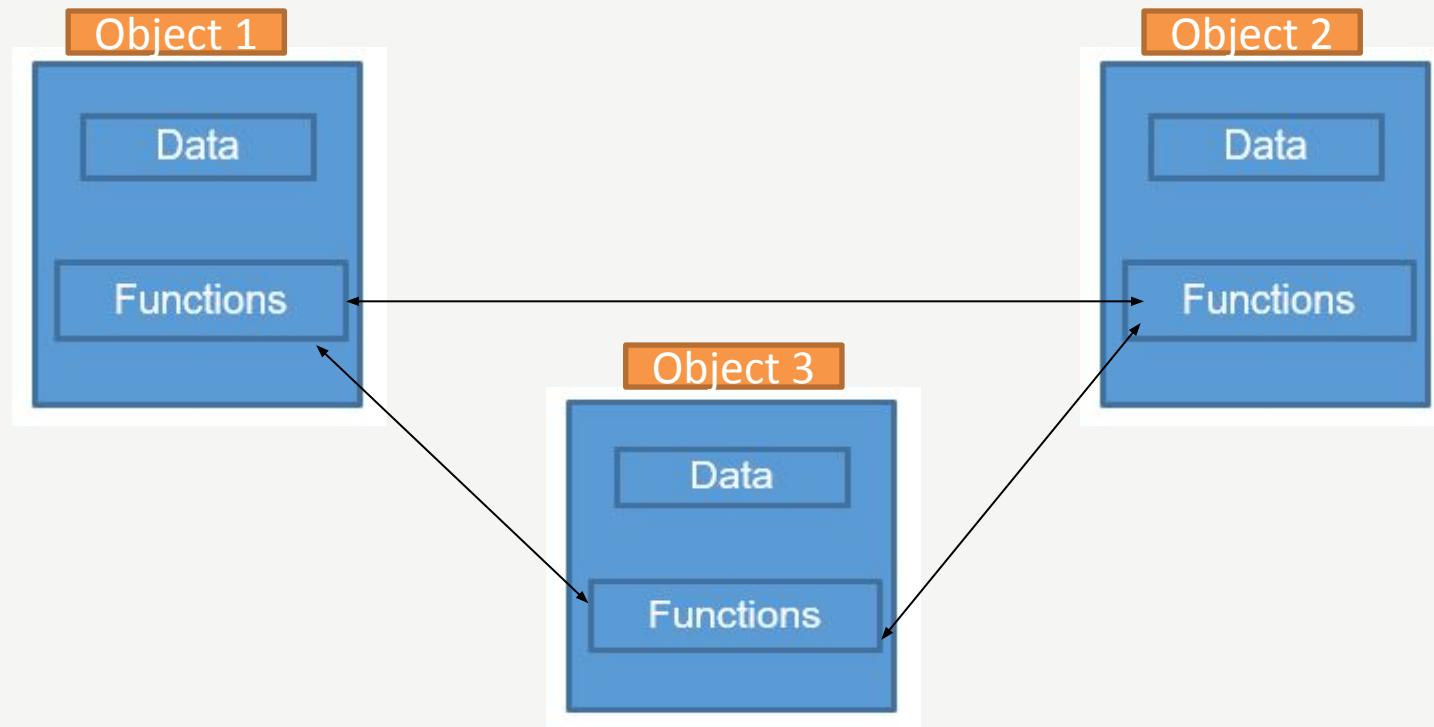
# Object Oriented Programming

In order to remove some of the flaws of POP, OOP came into existence.

OOP treats data as critical element in program development and does not allow it to flow freely around the system.

It ties the data more closely to the function that operates on it.

Object Oriented Programming allows decomposition of program into a number of entities called objects and then builds data and function around these objects.



# Object Oriented Programming

Characteristics of Object Oriented Programming:

- ❖ Emphasis on data rather than procedure.
- ❖ Program are divided into objects.
- ❖ Data is hidden and cannot be accessed by external functions.
- ❖ Objects may communicate with each other through functions.
- ❖ Follows bottom up approach in program design.



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# **Basics Concepts of Object Oriented Programming**

# Basic Concepts of OOP

Some of the basic concepts of object oriented programming are:

- ❖ Objects
- ❖ Classes
- ❖ Data abstraction and encapsulation
- ❖ Polymorphism
- ❖ Inheritance
- ❖ Dynamic Binding



Some of the basic concepts of object oriented programming are:

### ❖ **Objects:**

Object are variable of class or in other words we can say that it is the memory space of all the data members and member function of class.

There can be many object associated with class, for example if Mobile Phone is a class, then iPhone, One+, Oppo, Vivo, Samsung are objects.

## ❖ **Classes:**

We just mentioned above that object contain data, and code to manipulate that data.

The entire set can be made user defined data type with the help of a class.

## ❖ **Data abstraction and encapsulation:**

Abstraction: Refers to act of representing essential features and hiding the background details.

Classes use the concept of abstraction, as it contains both data members and member functions,

## ❖ **Encapsulation:**

Wrapping up of data and functions together into a single unit is known as encapsulation, and the single unit is known as class.

## ❖ **Polymorphism:**

Polymorphism means ability to take more than one forms.

An operation may exhibit different behaviors in different instances.

Operator overloading and Function overloading are example of polymorphism.

## ❖ Inheritance:

Inheritance is the process by which objects of one class acquire the properties of objects of another class.

Inheritance provide the concept of data reusability.

### **Types of Inheritance:**

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Tokens in C++

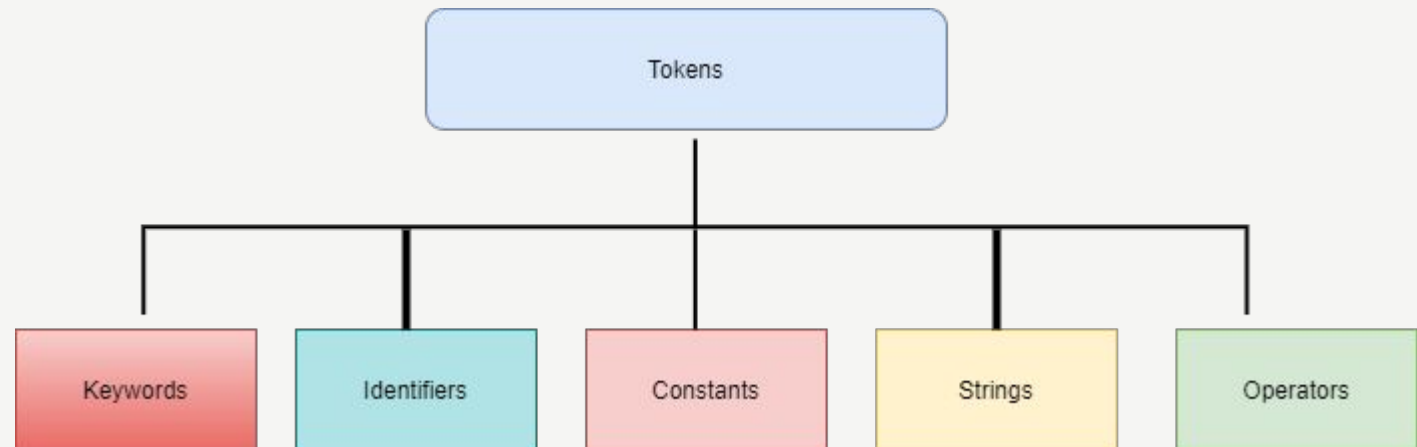
**Token:** When the compiler is processing the source code of a C++ program, each group of characters separated by white space is called a token.

Tokens are the smallest individual units in a program.

A C++ program is written using tokens.

It has the following tokens:

- ❖ Keywords
- ❖ Identifiers
- ❖ Constants
- ❖ Strings
- ❖ Operators



# Keywords in C++

Keywords (also known as reserved words) have special meaning to the C++ compiler and are always written or typed in short(lower) cases.

Keywords are words that the language uses for a special purpose, such as **float**, **int**, **private**, etc. It cannot be used for a variable name or function name.

Below is the table for the complete set of C++ keywords.

**Note:** The keywords not found in ANSI C are shown in bold.

| C++ Keyword   |               |                 |                 |            |               |                  |              |
|---------------|---------------|-----------------|-----------------|------------|---------------|------------------|--------------|
| auto          | else          | <b>operator</b> | <b>template</b> | <b>asm</b> | double        | <b>new</b>       | switch       |
| break         | enum          | <b>private</b>  | <b>this</b>     | case       | extern        | <b>protected</b> | <b>throw</b> |
| <b>catch</b>  | float         | <b>public</b>   | <b>try</b>      | char       | for           | register         | typedef      |
| <b>class</b>  | <b>friend</b> | return          | union           | const      | goto          | short            | unsigned     |
| continue      | if            | signed          | <b>virtual</b>  | default    | <b>inline</b> | sizeof           | void         |
| <b>delete</b> | int           | static          | volatile        | do         | long          | struct           | while        |





# Datatypes in C++

| Name        | Bytes* | Description                                                                                                                                                                                                                                                      | Range*                                                         |
|-------------|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|
| char        | 1      | character or integer 8 bits length.                                                                                                                                                                                                                              | signed: -128 to 127<br>unsigned: 0 to 255                      |
| short       | 2      | integer 16 bits length.                                                                                                                                                                                                                                          | signed: -32768 to 32767<br>unsigned: 0 to 65535                |
| long        | 4      | integer 32 bits length.                                                                                                                                                                                                                                          | signed: -2147483648 to 2147483647<br>unsigned: 0 to 4294967295 |
| int         | *      | Integer. Its length traditionally depends on the length of the system's Word type, thus in MSDOS it is 16 bits long, whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes). | See short, long                                                |
| float       | 4      | floating point number.                                                                                                                                                                                                                                           | 3.4e + / - 38 (7 digits)                                       |
| double      | 8      | double precision floating point number.                                                                                                                                                                                                                          | 1.7e + / - 308 (15 digits)                                     |
| long double | 10     | long double precision floating point number.                                                                                                                                                                                                                     | 1.2e + / - 4932 (19 digits)                                    |
| bool        | 1      | Boolean value. It can take one of two values: true or false NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it. Consult section <a href="#">bool type</a> for compatibility information.                                 | true or false                                                  |
| wchar_t     | 2      | Wide character. It is designed as a type to store international characters of a two-byte character set. NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it.                                                              | wide characters                                                |

## How Do You Declare Variable Data Types in C++?

To declare the data type of the variable to be used in C++, a definition must be made, as follows:

`<datatype> <name of variable>;`

Here is how you declare a variable data type in C++ code:

```
int age;  
float price;  
char letter;
```

It is possible to change the content of a variable by assigning a specific value anywhere in the program.

`<datatype> <name of variable> = <value>;`

Here is how that would look in code:

```
int age = 26;  
float price = 32.95;  
char letter = "f";
```

# Constant in C++

In C++, we can create variables whose value cannot be changed. For that, we use the `const` keyword. Here's an example:

```
const int SPEED_OF_LIGHT = 299792458;
```

```
SPEED_OF_LIGHT = 2500 // Error! SPEED_OF_LIGHT is a constant.
```

Here, we have used the keyword `const` to declare a constant named `SPEED_OF_LIGHT`.

If we try to change the value of `SPEED_OF_LIGHT`, we will get an error.

Constants are program components whose value does not change from the beginning to the end of the program. Constants with the following data types can be used in C++:

- ❖ Integer Constants
- ❖ Character Constants
- ❖ String Constants
- ❖ Real Constant

For example, let's take constant PI, there are two ways to declare this constant.

1. By using the const keyword in a variable declaration which will reserve a storage memory.

```
#include <iostream>
int main() {
    const double pi = 3.14;
    cout<<pi;
    //pi++; // will produce an error as constants cannot be changed
    return 0;}
```

2. By using the #define pre-processor directive which doesn't use memory for storage and without putting a semicolon character at the end of that statement

```
#include <iostream>
#define pi 3.14
int main() {
    cout<<pi;
    return 0;}
```

# Storage Class in C++

We already done with constants, and variables.

Variables are nothing but a fixed number of locations reserved under a specific name.

The mode in which a variable is assigned memory space is determined by its storage class.

The computer allows access to two locations:

- ❖ Memory
- ❖ CPU registers

Besides the location of the stored variable, a storage class determines:

- ✓ What is the default initial value, i.e. the value assumed when a variable has not been initialized.
- ✓ What is the scope of the variable, i.e. in which functions the value of the variable is available .

**storage\_class var\_data\_type var\_name;**

#C++ uses 5 storage classes, namely:

1. auto
2. register
3. extern
4. static
5. mutable



| Storage Class | Keyword  | Lifetime       | Visibility | Initial Value |
|---------------|----------|----------------|------------|---------------|
| Automatic     | auto     | Function Block | Local      | Garbage       |
| Register      | register | Function Block | Local      | Garbage       |
| Mutable       | mutable  | Class          | Local      | Garbage       |
| External      | extern   | Whole Program  | Global     | Zero          |
| Static        | static   | Whole Program  | Local      | Zero          |



## The auto Storage Class

The auto storage class is the default storage class for all local variables.

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

```
{  
    int mount;  
    auto int month;  
}
```

## The register Storage Class

The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word).

```
{  
    register int miles;  
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

**The Static Storage Class :** The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

```
#include <iostream>
using namespace std;
void LrnVrn() {
    static int a = 0; //static variable
    int b = 0; //local variable
    a++;
    b++;
    cout<<"a =" << a<<" and b =" <<b<<endl;
}
int main()
{
    LrnVrn();
    LrnVrn();
    LrnVrn();
}
```

**The extern Storage Class :** Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

```
int learnvern;  
Decleration and defintion
```

```
//LV.cpp  
#include<iostream>  
using namespace std;  
extern int a;  
int main()  
{  
  
    cout<<a;  
    return 0;  
}
```

```
extern int learnvern;  
Only Decleration
```

```
//LV1.cpp  
int a = 10
```

**The mutable Storage Class :** There may be a situation where we need to change one or more data members of class or structure through const function and do not want the function to update other members of class or structure . Above task can easily be performed using the mutable keyword. The keyword mutable is mainly used to allow a particular data member of const object to be modified.

```
#include <iostream>
using namespace std;

class LearnVern {
public:
    int a;

    // defining mutable variable
    // b which can be modified

    mutable int b;
    LearnVern()
    {
        a = 4;
        b = 10;
    }
};
```

```
int main()
{

    const LearnVern L;

    // trying to change the value
    L.b = 12;
    cout << L.b;

    // below lines
    // will throw error if uncommented
    // L.a = 56;
    // cout << L.a;
    return 0;
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# **Module – 6**

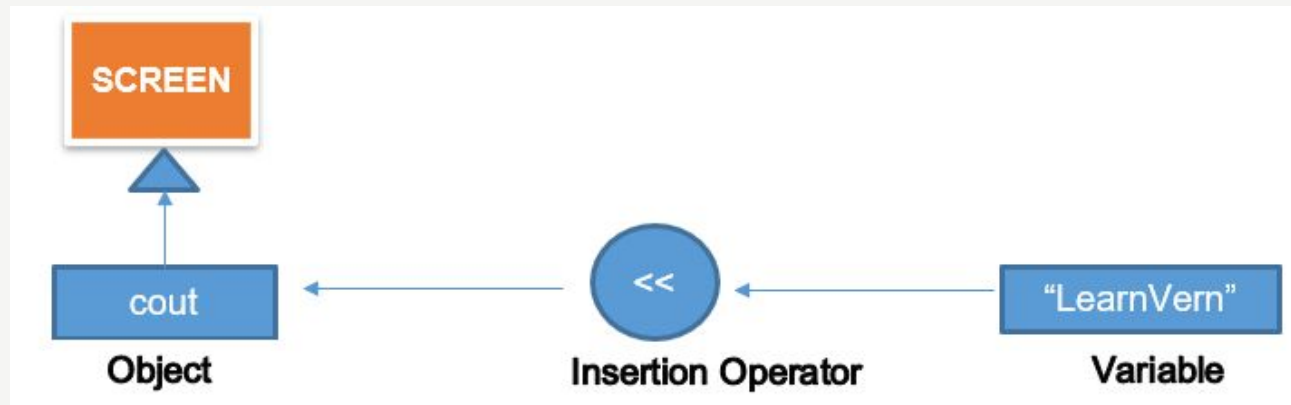
## **[C,C++ – Programming with C++]**

# Operators in C++

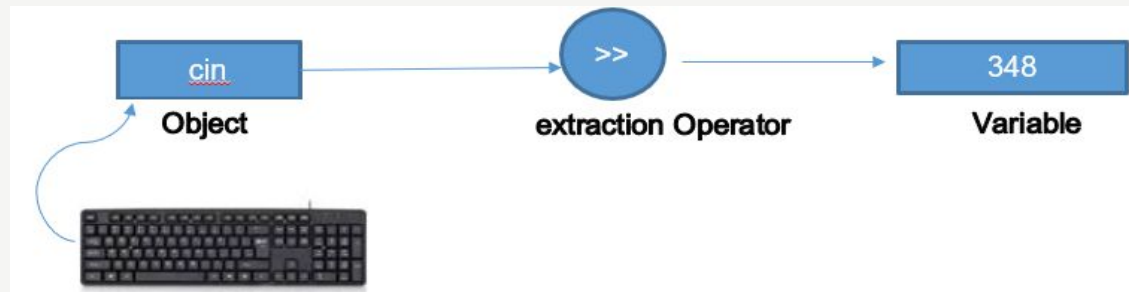
C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators.

- << insertion operator
- >> extraction operator
- :: scope resolution operator
- **delete** memory release operator
- **endl** line feed operator delete
- **new** memory allocation operator
- **setw** field width operator

**insertion operator(<<):** The operator << is called insertion or put to operator. It inserts the contents of variable on its right to the object in its left.



**Extraction operator(>>):** The operator >> is called extraction or get from operator. It extracts the values from keyboard and assign it to the variable on its right.



- In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator.
- :: variable-name, This operator allows access to the global version of a variable.

```
Output
inner block
    q = 30
    p = 40
::p = 20
outer block
    p = 30
::p = 20
```

```
#include <iostream>

using namespace std;
int p = 20; // global p

int main()
{
    int p = 30; // p redeclared, local to main
    {
        int q = p;
        int p = 40; // p declared again local to
                    inner block

        cout << "inner block \n";
        cout << "q =" << q << "\n";
        cout << "p =" << p << "\n";
        cout << "::p =" << ::p << "\n";
    }
    cout << "outer block \n"
    cout << "p =" << p << "\n";
    cout << "::p =" << ::p << "\n";

    return 0;
}
```



Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement:

```
cout << "LV =" << LV << endl << LV << endl << "LV1 = " << LV1 << endl << "LV2 = " << LV2 << endl;
```

Will produce three lines of output, one for each variable LV, LV1 and LV2 respectively.

If we assume the values of variables LV, LV1 and LV2 as 7, 14, and 2121 respectively, the output will appear as follows:

```
LV  = 7
LV1 = 14
LV2 = 2121
```

It is important to note that this form is not the ideal form of output. It should rather present output as:

```
LV  =    7
LV1 =   14
LV2 = 2121
```

Now the numbers are right-justified. This form of output is possible only if we can specify common field width for all the numbers and force them to be printed right-justified. The manipulator does this job.

The following statement:

```
cout << setw(5) << sum << endl;
```

The manipulator `setw(5)` specifies a field width 5 for printing the value of the variable value is right-justified within the field as shown below:

|  |  |   |   |   |
|--|--|---|---|---|
|  |  | 3 | 4 | 5 |
|--|--|---|---|---|

Program using MANIPULATORS

```
#include <iostream>
#include <iomanip> // for setw
using namespace std;
int main()
{
    int Length = 950, Breadth = 95, Height = 1045;
    cout << setw(10) << "Length" << setw(10) << Length << endl
         << setw(10) << "width" << setw(10) << width << endl
         << setw(10) << "Height" << setw(10) << Height << endl;
    return 0;
}
```

Output:

|        |      |
|--------|------|
| Length | 950  |
| Width  | 95   |
| Height | 1045 |



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# First C++ Program Using Class

---

# Inline Function

The main reason behind using functions in program is to save some memory space which is best in case when function is called many times.

Calling function is an expensive task, as it takes lot of time in executing series of instructions:

- Jumping to a function
- Saving Registers
- Putting arguments to stack
- Returning to calling function

Solution of the issue is to use macro definition, known as *macros*.

Preprocessor macros are used in C, they are not functions and their error checking is also not possible during compilation.

C++ has another solution of above problem, by using a new feature called *inline function*.

*An Inline function is a function which is expanded in a line when it is invoked.*

We can say that compiler replaces function call with the corresponding function code(similar to macro). *The inline function are defines as :*



```
inline returntype function_name()
{
    function body;
}
```

```
inline int square(int length)
{
    return (length * length);
}
```

# C++ program of inline function

```
#include<iostream>
using namespace std;
inline int sumf(int x, int y)
{
    return(x + y);
}

inline float divf(int x, int y)
{
    return(x / y);
}

int main()
{
    int p = 10;
    int q = 2;

    cout<<sumf(p,q)<<"\n";
    cout<<divf(p,q)<<"\n";
    return 0;
}
```

OUTPUT:  
12  
5.0000

# Simple C++ Program

Write a simple C++ program which takes two numbers as inputs from keyboard and display their sum and average on the screen.

```
// C++ program to calculate sum and average
// of two numbers
#include <iostream>
using namespace std;
int main()
{
    int num1, num2;
    int sum, avg;
    cout<<"Enter two numbers:";
    cin>> num1;
    cin>> num2;
    sum = num1 + num2;
    avg = sum/2;
    cout<<"SUM =" << sum<<"\n";
    cout<<"AVERAGE = " << avg <<"\n";
    return 0;
}
```

ANSI C++ Standard has added a new keyword namespace to define a scope that could hold global identifiers. The best example of namespace scope is the C++ Standard Library. All classes, functions and templates are declared within the namespace named std. That is why we have been using the directive ***using namespace std;*** in our program that uses standard library





# C++ Program using Class

---

Write a simple C++ using concept of class and object and explain it in detail.

**//Use of class in a C++ program.**

```
#include <iostream>
using namespace std;
class employee
{
    char ename[20];
    int  eage;

    public:
        void collectdata(void);
        void printdetail(void);
};

void employee :: collectdata(void)
{
    cout <<"Enter name of employee:";
    cin >> ename;
    cout <<"Enter age of employee:";
    cin >> eage;
}
```

```
void employee :: printdetail(void)
{
    cout << "\n Employee Name: " << ename;
    cout << "\n Employee age; " << eage;
}

int main()
{
    employee e;
    e.collectdata();
    e.printdetail();

    return 0;
}
```



# Function Overloading in C++

---

Overloading can be defined as, use of the same thing for different purposes. C++ permits both overloading of operators and functions.

This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as ***function polymorphism in OOP***.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type

of the arguments. An overloaded ***avg()*** function handles different types of data as shown in box:

```
// Declarations
int avg(int x, int y);
//declaration 1
int avg(int x, int y, int z);
//declaration 2
double avg(double a, double b); //declaration 3
double avg(int x, double y);
//declaration 4
double avg (double x, int y);
//declaration 5

// Function calls
cout <<avg (15, 100);
cout <<avg (15, 110, 115);
cout <<avg (18.5, 17.5);
cout <<avg (150, 12.0);
cout <<avg (2.75, 55);
```

Write a simple C++ using concept of class and object and explain it in detail.

**//Function overloading.**

```
#include <iostream>
using namespace std;
int volume(int);
int volume(int, int);
int volume(int, int, int);

int main()
{
    cout<< volume(15) << "\n";
    cout<< volume(5, 10) << "\n";
    cout<< volume(5, 10, 15) << "\n";

    return 0;
}

int volume(int length)
{
    return(length*length*length);
}
```

```
int volume(int radius, int height)
{
    return( 3.14*radius*radius*height);
}

int volume(int length, int bredth, int height)
{
    return(length*breadth*height);
}
```

# Call By Value & Call By Reference in C++

---

We can call function in two ways either using **Call by Value** or using **Call by Reference**. The basic difference between them is by the type of values passed to them as parameters.

The parameters passed to function are called **actual parameters** whereas the parameters received by function are called **formal parameters**.

**Call By Value:** In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.

**Call by Reference:** Both the actual and formal parameters refer to the same locations, so any changes made inside the function are actually reflected in actual parameters of the caller.

```
fun(int x, int y) //formal
argument
{
    z = x + y;
}

int main()
{
    int a = 10, b = 20;
    fun(a, b); // actual
argument

    return 0;
}
```

### //Call By Value

```
#include<iostream>
using namespace std;
void swapv(int x, int y);

int main()
{
    int a = 2, b = 4;
    swapv(a, b);
    cout<<"a ="<<a <<" b =" <<b;

    return 0;
}

void swapv(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
    cout<<"x ="<<x <<" y =" <<y;
}
```

### Output

x = 4 y = 2  
a = 2 b = 4

### //Call By Reference

```
#include<iostream>
using namespace std;
void swapr(int x, int y);

int main()
{
    int a = 2, b = 4;
    swapr(&a, &b);
    cout<<"a ="<<a <<" b =" <<b;

    return 0;
}

void swapr(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
    cout<<"x ="<<*x <<" y =" <<*y;
}
```

### Output

x = 4 y = 2  
a = 4 b = 2





# Classes and Objects in C++

---

**Structures Revisited:** One of the unique features of the C language is structures. Using structures we pack data of different types. It is a user-defined data type with a template that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations.

For example, consider the following declaration:

```
struct person
{
    int age;
    char name[10];
    float money;
} ;
```

The keyword `struct` declares ***person*** as a new data type that can hold three fields of different data types. *These fields are known as structure members or elements.*

### ***Structure Variable:***

The identifier ***person***, which is referred to as structure name, can be used to create variables of type ***person***.

**struct person A;**

```
struct person
{
    int age;
    char name[10];
    float money;
} ;
```

A is a variable of type ***person*** and has three member variables. Member variables can be accessed using the dot operator as follows:

```
strcpy (A.name, "Parul");
```

```
A.age = 19;
```

```
A.money = 595.5;
```

***Limitation of Structures*** : The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

The complex numbers a1, a2, and a3 can easily be assigned values using the dot operator but we cannot add two complex numbers or subtract one from the other.

For example,  
 $a3 = a1 + a2$ ; //is illegal in C programming.

```
struct complex
{
    float a;
    float b;
};

struct complex a1, a2, a3;
```

Other drawback of structures is that they do not permit data hiding, members of structure can be directly accessed by the structure variables by any function anywhere in their scope. i.e. the structure members are ***public*** by default.

**Extensions to Structures:** Apart from all the existing features of structure in C, C++ has expanded its capabilities further to suit its OOP philosophy. It provides a facility to hide the data which is one of the main principles of OOP.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

*The only difference between a structure and a class in C++ is that, by default, the members of a class are private, while, by default, the members of a structure are public.*

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type. Generally, a class specification has two parts:

- 1. Class declaration***
- 2. Class function definitions***

*The class declaration describes the type and scope of its members.*

*The class function definitions describe how the class functions are implemented.*

The general form of a class declaration is shown in box above:

```
class class_name
{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declaration;
};
```

The class declaration is similar to a struct declaration. These functions and variables are collectively called class members. The keywords ***private and public are known as visibility labels***. Note that these keywords are followed by a colon.

The class members that have been declared as ***private*** can be accessed only from ***within the class***.

On the other hand, ***public*** members can be accessed from ***outside the class also***.

The data hiding (using private declaration) is the key feature of object-oriented programming.

```
class LearnVern
{
    int a;
    int b;
    public:
        void getdata(int, int);
        void display();
};
```

***The use of the keyword private is optional.***

By default, the members of a class are private.

Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class.

- ❖ **Objects:** Object are variable of class or in other words we can say that it is the memory space of all the data members and member function of class. There can be many object associated with class, for example if Mobile Phone is a class, then iPhone, One+, Oppo, Vivo, Samsung are objects.

```
class LearnVern
{
    int a;
    int b;
public:
    void getdata(int, int);
    void display();
}0;
```

Or

```
Learnvern 0;
```



# **Access Modifiers (Public, Private & Protected)**


---

## ← Select privacy


### Who can see your post?



Your post will show up in Feed, on your profile and in search results.

Learn more about [post privacy](#).

☒  **Public**  
Anyone on or off Facebook

☐  **Friends**  
Your friends on Facebook

☐  **Friends except...**   
Don't show to some friends

☐  **Specific friends**   
Only show to some friends

☐  **Only me**  
Only me

# Access Modifiers

***Access Modifiers or Access Specifiers*** in a class are used to assign the accessibility to the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

**Note:** If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.

- 1. Public:** All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
- 2. Private:** The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.
- 3. Protected:** Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of its class unless with the help of friend class, the difference is that the class members declared as Protected can be accessed by any subclass(derived class) of that class as well.

```
#include<iostream>
using namespace std;
class LearnVern
{
    int a;
    int b;
public:
    int c;
};
int main()
{
    LearnVern L;
    L.a = 10;
    L.b = 20;    // Will produce error as both a & b are private
                //
    L.c = 30;    // Is legal as c is public member

    cout<<"c = "<<L.c;    //will give output c = 30
}
```

```
#include<iostream>
Using namespace std;
class LearnVern
{
    int a = 50;
public:
    int c;
    void display()
    {
        cout<< "\n"<<"a = "<<a; // will give output a = 50
    }
};
int main()
{
    LearnVern L;
    L.c = 30;
    cout<<"c = "<<L.c;    //will give output c = 30
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Defining member function inside and outside the class

---

**Member functions of Classes:** It is seen that the private data member of class cannot be accessed directly using object, but via member function of class which are generally declared in public section.

The member function can be defined inside or outside the Class.

1. Inside the class: ***Simple as other functions***
2. Outside the class: ***Using Scope resolution operator.***

```
class class_name
{
    private:
        variable declarations;

    public:
        function1 declaration;
        function2 declaration;
};
```



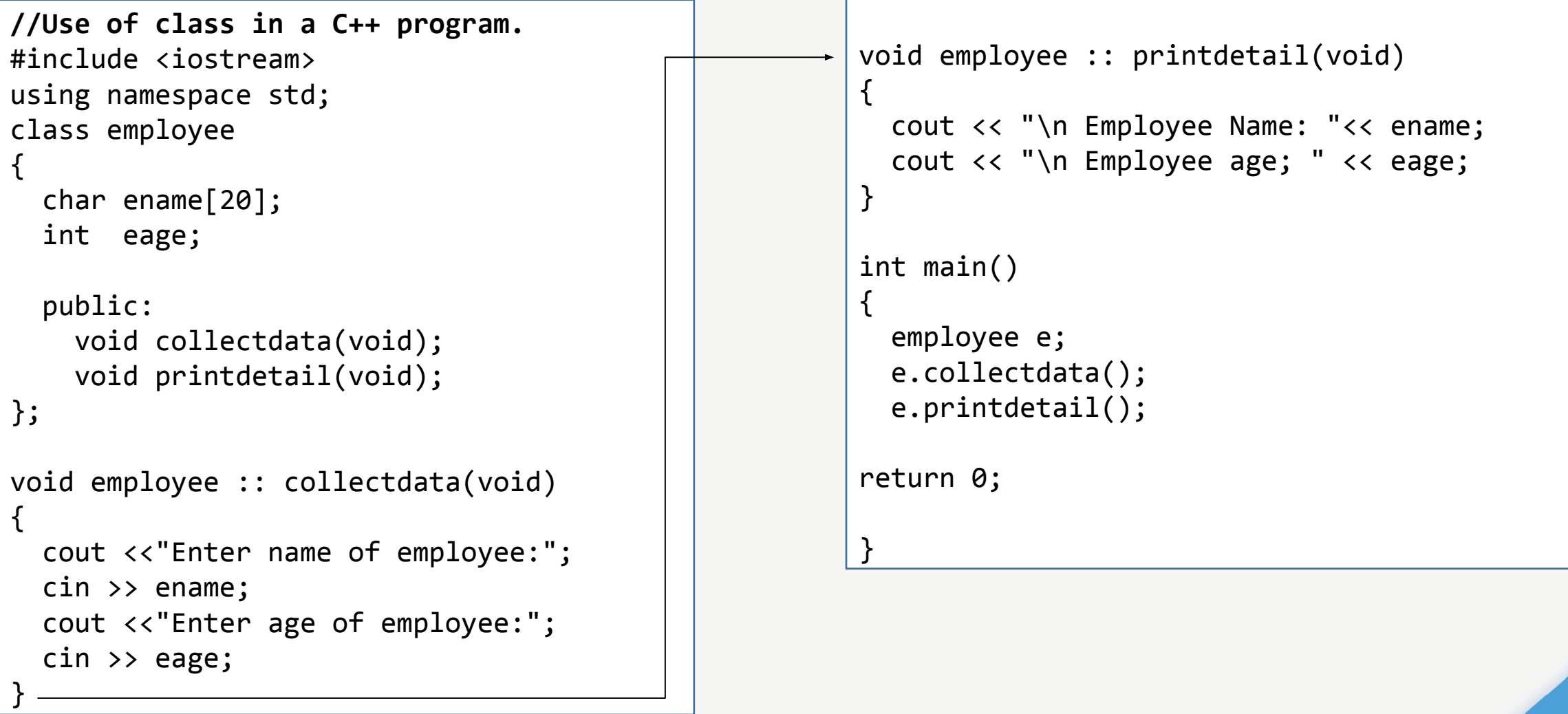
## C++ program where member functions are defined outside the class.

//Use of class in a C++ program.

```
#include <iostream>
using namespace std;
class employee
{
    char ename[20];
    int  eage;

    public:
        void collectdata(void);
        void printdetail(void);
};

void employee :: collectdata(void)
{
    cout << "Enter name of employee:";
    cin >> ename;
    cout << "Enter age of employee:";
    cin >> eage;
}
```



```
void employee :: printdetail(void)
{
    cout << "\n Employee Name: " << ename;
    cout << "\n Employee age; " << eage;
}

int main()
{
    employee e;
    e.collectdata();
    e.printdetail();

    return 0;
}
```

## C++ program where member functions are defined outside the class.

```
#include <iostream>
using namespace std;
class employee
{
    char ename[20];
    int  eage;

public:
    void collectdata(void)
    {
        cout << "Enter name of employee:";
        cin >> ename;
        cout << "Enter age of employee:";
        cin >> eage;
    }
```

```
void printdetail(void)
{
    cout << "\n Employee Name: " << ename;
    cout << "\n Employee age; " << eage;
}

};

int main()
{
    employee e;
    e.collectdata();
    e.printdetail();

    return 0;
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Static Data Members

---

***Static Data Members:*** A data member of a class can be qualified as static.

Characteristic of static member variable are equivalent to that of a C static variable.

Properties of static member variable are :

- ❖ It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- ❖ Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- ❖ It is visible only within the class, but its lifetime is the entire program.

***Note: Static variables are normally used to maintain values common to entire class.***



```
#include <iostream>
using namespace std;

class LearnVern
{
    static int count;
    int num;

public:
    void getvalue(int m)
    {
        num = m;
        count += 1;
    }

    void val_counter(void)
    {
        cout << "count: " << count << "\n";
    }
};

int LearnVern :: count;
```

**Note:** The type and scope of static member variable must be defined **outside the class** definition. As static members are stored separately rather than as part of object. Since they are associated with class itself rather than with any class object, they are also known as **class variable**.

```
int main()
{
    Learnvern x, y, z;
    cout << "BEFORE INTIALIZATION";

    x.val_counter();
    y.val_counter();
    z.val_counter();

    x.getvalue(10);
    y.getvalue(20);
    x.getvalue(30);

    cout << "AFTER INTIALIZATION";
    x.val_counter();
    y.val_counter();
    z.val_counter();

    return 0;
}
```

```
#include <iostream>
using namespace std;

class LearnVern
{
    static int count;
    int num;

public:
    void getvalue(int m)
    {
        num = m;
        count += 1;
    }

    void val_counter(void)
    {
        cout << "count: " << count << "\n";
    }
};

int LearnVern :: count;
```

```
int main()
{
    Learnvern x, y, z;
    cout<<"BEFORE INTIALIZATION";

    x.val_counter();
    y.val_counter();
    z.val_counter();

    x.getvalue(10);
    y.getvalue(20);
    x.getvalue(30);

    cout<<"AFTER INTIALIZATION";
    x.val_counter();
    y.val_counter();
    z.val_counter();

    return 0;
}
```

### OUTPUT

**BEFORE  
INITIALIZATION**

**count: 0**

**count: 0**

**count: 0**

**AFTER INITIALIZATION**

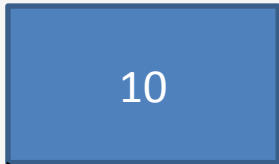
**count: 3**

**count: 3**

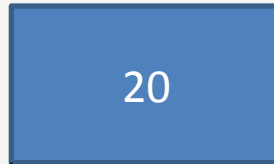
**count: 3**



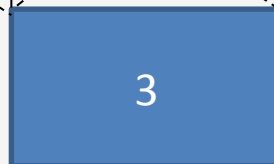
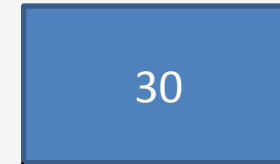
Object 1  
num



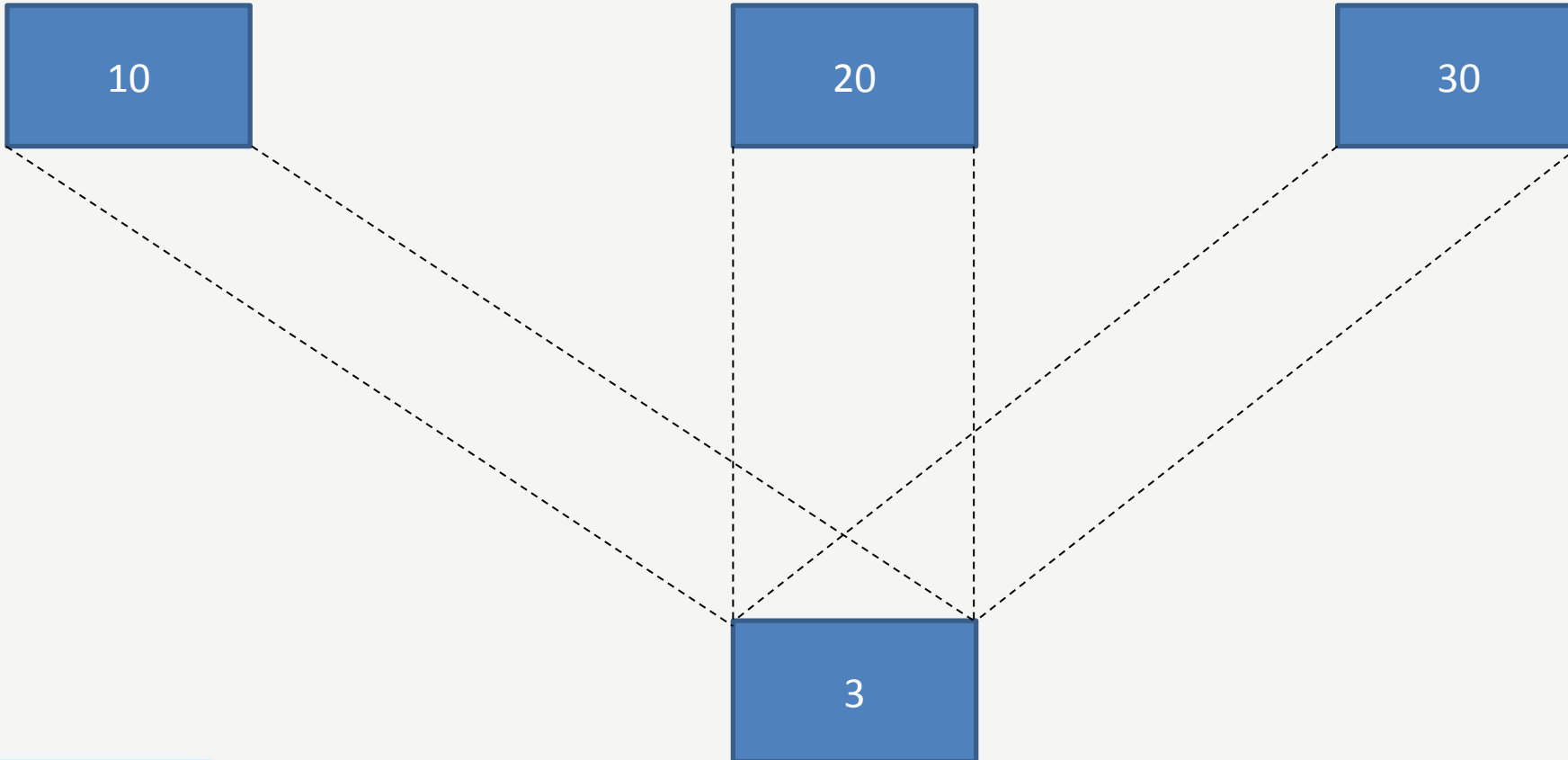
Object 2  
num



Object 3  
num



count





# Static Member Function

---



## ***Static Member Functions:***

Like static member variable, we can also have static member functions.

A member function declared static has the following properties:

- ❖ A static function can have access to only other static members(functions or variables) declared in the same class.
- ❖ A static member function can be called using the class name (instead of its objects) as follows:

***class-name :: function-name;***



```
#include<iostream>
using namespace std;
class LearnVern
{
    int num;
    static int count;
public:
    void val_counter(void)
    {
        num = ++count;
    }

    void obj_counter(void)
    {
        cout << "Object number:" << num << "\n";
    }

    static void display_count(void)
    {
        cout << "count:" << count << "\n";
    }
};
int LearnVern :: count;
```

```
int main()
{
    LearnVern x, y;
    x.val_counter();
    y.val_counter();

    LearnVern :: display_count();

    LearnVern z;
    z.val_counter();

    LearnVern :: display_count();

    x.obj_counter();
    y.obj_counter();
    z.obj_counter();

    return 0;
}
```

## OUTPUT

```
count: 2
count: 3
Object number: 1
Object number: 2
Object number: 3
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Objects as Function Arguments

---

**Objects as Function Arguments** : Similar to other data type, an object may be used in a function argument.

There exist two ways of doing same(Call By Value & Call By Reference):

- ❖ A copy of the entire object is passed to the function.
- ❖ Only the address of the object is transferred to the function.

The first method is called ***call-by-value***. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.

The second method is called ***call-by-reference***. When an address of the object is passed, the called function works directly on the actual object used in the call.



```
#include <iostream>
using namespace std;

class clock
{
    int hrs;
    int mints;
public:
    void insert_time(int h, int m)
    {
        hrs = h;
        mints = m;
    }

    void display_time (void)
    {
        cout << hrs << " hours & ";
        cout << mints << "minutes"<< "\n";
    }

    void sum(clock, clock);
};
```

```
void clock :: sum(clock t1, clock t2)
{
    mints = t1.mints + t2.mints;
    hrs = mints / 60;
    mints = mints % 60;
    hrs = hrs + t1.hrs + t2.hrs;
}

int main()
{
    clock C1, C2, C3;
    C1.insert_time (3,55);
    C2.insert_time (3,45);

    C3.sum(C1, C2);

    cout << "Time1 = ";    C1.display_time();
    cout << "Time2 = ";    C2.display_time();
    cout << "Time3 = ";    C3.display_time();

    return 0;
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Friend Function in C++

---

**A friend function possesses certain special characteristics:**

- ❖ It is not in the scope of the class to which it has been declared as friend. Since it is not in the scope of the class, it cannot be called using the object of that class.
- ❖ It can be invoked like a normal function without the help of any object.
- ❖ Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- ❖ It can be declared either in the public or the private part of a class without affecting its meaning.
- ❖ Usually, it has the objects as arguments.



```
#include<iostream>
using namespace std;
class LearnVern {
    int x;
    int y;
public:
    void getdata() {
        x = 30;
        y = 40;
    }
    friend int avg(LearnVern );
};

int avg(LearnVern LV)
{
    return int(LV.x + LV.y)/2;
}

int main()
{
    LearnVern O;
    O.getdata();
    cout<<"Average Value = "<< avg(O)<< "\n";
    return 0;
}
```





```
#include<iostream>
using namespace std;
class LV2;

class LV1
{
    int x;
public:
    void getdata(int a)
    {
        x = a;
    }
    friend void maximum(LV1, LV2 );
};

class LV2
{
    int y;
public:
    void getdata(int a)
    {
        y = a;
    }
    friend void maximum(LV1, LV2 );
};
```

```
void maximum(LV1 o, LV2 p)
{
    if(o.x >= p.y)
        cout<<"maximum among two classes = "<< o.x;
    else
        cout<<"maximum among two classes = "<< p.y;
}

int main()
{
    LV1 O;
    O.getdata(30);
    LV2 P;
    P.getdata(20);
    maximum(O, P);

    return 0;
}
```

# Constructor in C++

**Constructors** : A constructor is a '*special*' member function whose task is to initialize the objects of its class when it is created.

It is special because its *name is the same as the class name*.

The constructor is invoked whenever an object of its associated class is created.

It is called constructor because it constructs the values of data members of the class.

# Default Constructor in C++

- ❖ Default Constructor: The constructor without argument are known as default constructor.

```
#include<iostream>
using namespace std;
class LearnVern
{
    int m;
    int n;
public:
    LearnVern(void);
    void display();
};

LearnVern :: LearnVern(void)
{
    m = 10;
    n = 20;
}
```

```
void display()
{
    cout<<"m = "<< a <<"n = "<<b;
}
int main()
{
    LearnVern 0;
    0.display()
    return 0;
}
```

# Parameterized Constructor in C++

```
class LearnVern
{
    int x;
public:
    LearnVern()
    {
        x = 10;
    }
    LearnVern(LearnVern & O)
    {
        x = O.x;
    }
    void showdata()
    {
        cout<<"x =" << x;
    }
};
```

```
int main()
{
    LearnVern A;
    LearnVern B(A);
    LearnVern C = A;
    cout << "\n x of object A: ";
    A.display();
    cout << "\n x of object B: ";
    B.display();
    cout << "\n x of object C: ";
    C.display();

    return 0;
}
```

# Copy Constructor in C++

Copy Constructor: The constructor that accepts a reference of its own class is known as copy constructor.

A copy constructor is used to declare and initialize an object from another object. For example:

The statement ***LearnVern O2 (O1);*** would define the object O2 and at the same time initialize it to the values of O1.

*Another form of this statement is **LearnVern O2 = O1;***  
The process of initializing through a copy constructor is known as copy initialization.

```
class LearnVern
{
    int m;
    int n;
public:
    LearnVern(LearnVern &);
};
```

# **Multiple Constructors in a Class OR Overloaded Constructors**

## Multiple Constructors in a Class:

So far we have used **default constructor** where the constructor itself supplies the data values by the calling program, **parametrized constructors** where the function call passes the appropriate values from main() and **copy constructors** where The constructor that accepts a reference of its own class is known as copy constructor. They are:

```
LearnVern();           // default constructor
LearnVern(int, int);    //parametrized
constructors
LearnVern(LearnVern &); //copy constructors
```

C++ permits us to use all these constructors in the same class. For example, we could define a class as shown in box:

```
class LearnVern
{
    int m, n;
public:
    LearnVern() // default constructor
    {
        m = 0;
        n = 0;
    }

    LearnVern(int a, int b) //parametrized
                                constructors

    {
        m = a;
        n = b;
    }

    LearnVern (LearnVern & i) // copy
                                constructors
    {
        m = i.m;
        n = i.n;
    }
};
```

The declaration ***LearnVern A;*** would automatically invoke the first constructor(default) and set both *m* and *n* of object *A* to zero.

The statement ***LearnVern B(20,40);*** would call the second constructor(parametrized) which will initialize the data members *m* and *n* of object *B* to 20 and 40 respectively.

Finally, the statement ***LearnVern C(B);*** would invoke the third constructor which copies the values of object *B* into object *C*. In other words, it sets the value of every data element of object *C* to the value of the corresponding data element of object *B*. As mentioned earlier, such a constructor is called the **copy constructor**.

Process of sharing the same name by two or more functions is referred to as ***function overloading***. Similarly, when more than one constructor function is defined in a class, we say that the ***constructor is overloaded***.

```
class LearnVern
{
    int m, n;
public:
    LearnVern() // default constructor
    {
        m = 0;
        n = 0;
    }

    LearnVern(int a, int b) //parametrized
                                constructors

    {
        m = a;
        n = b;
    }

    LearnVern (LearnVern & i) // copy
                                constructors

    {
        m = i.m;
        n = i.n;
    }
};
```





```
#include<iostream>
using namespace std;
class LearnVern
{
    int m, n;
public:
    LearnVern()
    {
        m = 0;
        n = 0;
    }

    LearnVern(int a, int b)
    {
        m = a;
        n = b;
    }

    LearnVern (LearnVern & i)
    {
        m = i.m;
        n = i.n;
    }
}
```

```
void display()
{
    cout<<"m = "<<m<<"n = "<<n;
}

int main()
{
    LearnVern A;
    LearnVern B(5, 10);
    LearnVern C(A);
    cout << "\n m & n of object A: ";
    A.display();
    cout << "\n m & n of object B: ";
    B.display();
    cout << "\n m & n of object C: ";
    C.display();

    return 0;
}
```

Operator overloading is one of the many exciting features of C++ language.

It is an important technique that has enhanced the power of extensibility of C++.

As already discussed many times that C++ tries to make the user-defined data types behave in much the same way as the built-in types.

For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.

This means that C++ has the ability to provide the operators with a special meaning for a data type. *The mechanism of giving such special meanings to an operator is known **as operator overloading**.*

We can overload (give additional meaning to) all the C++ operators by the creative use of the operator except the following:

- ☐ Class member access operators (`., *`).
- ☐ Scope resolution operator (`::`)
- ☐ Size operator (`sizeof`)
- ☐ Conditional operator (`?:`).

**Defining Operator Overloading :** To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is:

```
return type classname :: operator op (arglist)
{
    Function body
}
```

**Note:** return type is the type of value returned by the specified operation and **op** the operator being overloaded. The op is preceded by the **keyword operator**.  
*operator op is the function name.*



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Unary Operator Overloading

---

Operator functions must be either *member functions* (non-static) or *friend functions*.

A basic difference between them is that a *friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators.*

This is because the object used to invoke the member function is passed implicitly and therefore available for the member function. This is not the case with friend functions.

Overloaded operator function can be called / invoked by expression such as :

1.  $op\ x$  or  $x\ op$  // for unary operators
2.  $x\ op\ y$  // for binary operators



## OVERLOADING UNARY MINUS

```
#include <iostream>
using namespace std;
class LearnVern
{
    int m;
    int n;
    int o;
public:
    void getdata (int a, int b, int c);
    void showdata (void);
    void operator-(); // overload unary minus
};

void LearnVern: getdata(int a, int b, int c)
{
    m = a;
    n = b;
    z = c;
}

void Learnvern :: showdata(void)
{
    cout <<"m = " << m <<"\n";
    cout <<"n = " << n <<"\n";
    cout <<"o = " << o;
}
```

```
void Learnvern :: operator-()
{
    m = -m;
    n = -n;
    o = -o;
}

int main()
{
    LearnVern 0;
    0.getdata(15, -25, 35);
    cout<<"Value of object before
overloading\n";
    0.showdata();
    -0;

    cout<<"Value of object after
overloading\n";
    0.showdata();
    return 0;
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Binary Operator Overloading

---



Operator functions must be either *member functions* (non-static) or *friend functions*.

A basic difference between them is that a *friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators.*

This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions.

Overloaded operator function can be called / invoked by expression such as :

1. `op x` or `x op` // for unary operators
2. `x op y` // for binary operators

Let class LearnVern has 3 objects namely A, B, C. Then the statement `C = A + B` will invoke the binary operator function, This can be understood as `C = Sum(A, B)` where Sum function is used for adding two objects.

**We can also rewrite `C = A + B` as `C = A.operator+(B);`**



## OVERLOADING BINARY PLUS

```
#include <iostream>
using namespace std;
class LearnVern
{
    int m;
    int n;
public:
    LearnVern() { }
    LearnVern(int a, int b)
    {
        m = a;
        n = b;
    }
    LearnVern operator+(LearnVern);
//overload binary plus
    void showdata();
};

LearnVern Learnvern ::
operator+(LearnVern o)
{
    LearnVern p;
    p.m = m + o.m;
    p.n = n + o.n;
    return(p);
}
```

```
void Learnvern :: showdata(void)
{
    cout <<"m = "<<m<<" & " <<cout<<"n = "
    <<n <<"\n";
}

int main()
{
    LearnVern A(5, 10);
    LearnVern B(10, 20);
    LearnVern C;
    C = A + B;
    cout<<"Object A: "; A.showdata();
    cout<<"Object B: "; B.showdata();
    cout<<"Object C: "; C.showdata();

    return 0;
}
```



# Inheritance in C++

---

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again.

It would not only save time and money but also reduce frustration and increase reliability.

Fortunately, C++ strongly supports the concept of reusability.

*The mechanism of deriving a new class from an old one is called **inheritance (or derivation)**.*

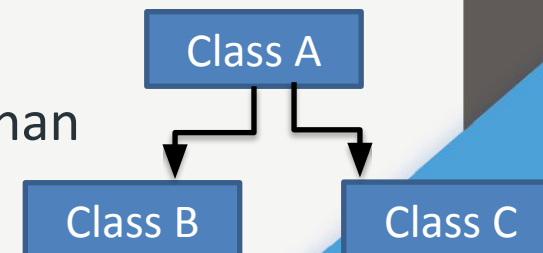
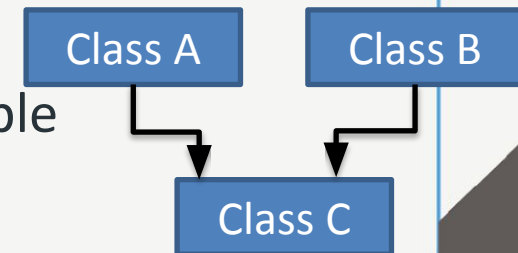
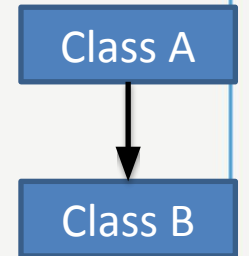
The old class is referred to as the **base class** and the new one is called the **derived class or subclass**.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level.

**Single Inheritance:** A derived class with only one base class, is called single inheritance.

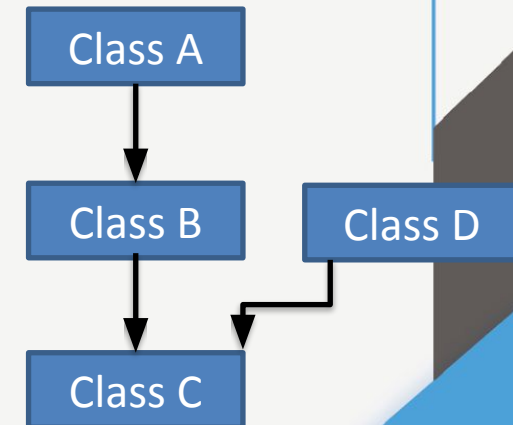
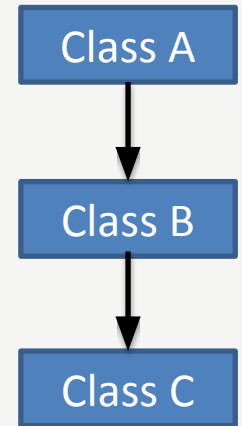
**Multiple Inheritance:** A derived class with several base classes is called multiple inheritance.

**Hierarchical Inheritance:** Properties of one class may be inherited by more than one class. This process is known as hierarchical inheritance.



**Multilevel Inheritance:** The mechanism of deriving a class from another 'derived class is known as multilevel inheritance.

**Hybrid Inheritance:** When two or more than two inheritance are mixed together to form a unique inheritance then this is called Hybrid Inheritance.





# Single Inheritance in C++

---

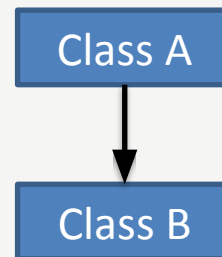
The mechanism of deriving a new class from an old one is called **inheritance**.

The old class is referred to as the **base class** and the new one is called the **derived class or subclass**.

The derived class inherits some or all of the traits from the base class.

A class can also inherit properties from more than one class or from more than one level.

**Single Inheritance:** A derived class with only one base class, is called single inheritance.







```
#include<iostream>
using namespace std;
class LV1
{
    int m;
public:
    int n;
    void getdata();
    int return_m();
    void show_m();
};

class LV2: public LV1
{
    int o;
public:
    void add();
    void display();
};

void LV1::getdata()
{
    m = 10;  n = 20;
}

int LV1:: return_m()
{
    return m;
}
```

```
void LV1::show_m()
{
    cout<< "m = "<< m<<"\n";
}

void LV2::add()
{
    getdata();
    o = n + return_m();
}

void LV2::display()
{
    show_m();
    cout<<"n = "<< n<<"\n";
    cout<<"o = "<< o<<"\n";
}

int main()
{
    LV2 A;
    A.getdata();
    A.add();
    A.show_m();
    A.display();
    return 0;
}
```

## OUTPUT

```
m = 10
m = 10
n = 20
o = 30
```



```
#include<iostream>
using namespace std;
class LV1
{
    int m;
public:
    int n;
    void getdata();
    int return_m();
    void show_m();
};

class LV2: private LV1
{
    int o;
public:
    void add();
    void display();
};

void LV1::getdata()
{
    m = 10;  n = 20;
}

int LV1 :: return_m()
{
    return m;
}
```

```
void LV1::show_m()
{
    cout<< "m = "<< m<<"\n";
}

void LV2::add()
{
    getdata();
    o = n + return_m();
}

void LV2::display()
{
    show_m();
    cout<<"n = "<< n<<"\n";
    cout<<"o = "<< o<<"\n";
}

int main()
{
    LV2 A;
    A.getdata(); // Not work
    A.add();
    A.show_m(); // Not work
    A.display();
    return 0;
}
```

## OUTPUT

```
m = 10
m = 10
n = 20
o = 30
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Multilevel Inheritance in C++

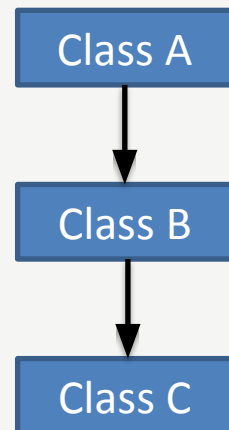
---

The mechanism of deriving a new class from an old one is called **inheritance**.

The old class is referred to as the **base class** and the new one is called the **derived class or subclass**. The derived class inherits some or all of the traits from the base class.

A class can also inherit properties from more than one class or from more than one level.

**Multilevel Inheritance:** The mechanism of deriving a class from another 'derived class is known as multilevel inheritance.





```
#include<iostream>
using namespace std;
class employee
{
    protected:
        int eid;
    public:
        void getid(int);
        void showid(void);
};

void employee :: getid(int id)
{
    eid = id;
}

void employee :: showid()
{
    cout<<"Employee Id: " << eid<<"\n";
}

class task : public employee
{
    protected:
        int task1;
        int task2;
    public:
        void gettask(int, int);
        void putscore(void);
};
```

```
void task :: gettask(int x, int y)
{
    task1 = x; task2 = y;
}
void task :: putscore()
{
    cout << "Score of task1: "<<task1<<"\n";
    cout << "Score of task2: "<<task2<<"\n";
}
class appraisal: public task
{
    int grade;
    public:
        void display(void);
};
void appraisal :: display(void)
{
    grade = task1 + task2;
    showid();
    putscore();
    cout<<"Final grade: "<<grade<<"\n";
}

int main()
{
    appraisal A;
    A.getid(705);
    A.gettask(100, 150);
    A.display();
    return 0;
}
```

## OUTPUT

**Employee Id: 705**  
**Score of task1: 100**  
**Score of task2: 150**  
**Final grade: 250**



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

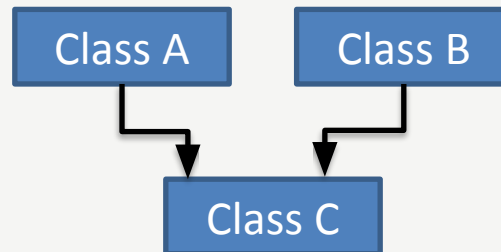
# Multiple Inheritance in C++

---

The mechanism of deriving a new class from an old one is called **inheritance**.

The old class is referred to as the **base class** and the new one is called the **derived class or subclass**. The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level.

**Multiple Inheritance:** A derived class with several base classes is called multiple inheritance.





```
#include<iostream>
using namespace std;
class LV1
{
    protected:
        int a;
    public:
        void get_a(int);
}

class LV2
{
    protected:
        int b;
    public:
        void get_b(int);
}

class LV3 : public LV1, public LV2
{
    public:
        void display();
};
```

```
void LV1 :: get_a(int m)
{
    a = m;
}

void LV2 :: get_b(int n)
{
    b = n;
}

void LV3 :: display(void)
{
    cout<<"a = "<<a<<"\n";
    cout<<"b = "<<b<<"\n";
    cout<<"a + b = "<<a + b<<"\n";
}

int main()
{
    LV3 A;
    A.get_a(10);
    A.get_b(100);
    A.display();
    return 0;
}
```

## OUTPUT

```
a = 10
b = 100
a + b = 110
```





**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Hierarchical Inheritance in C++

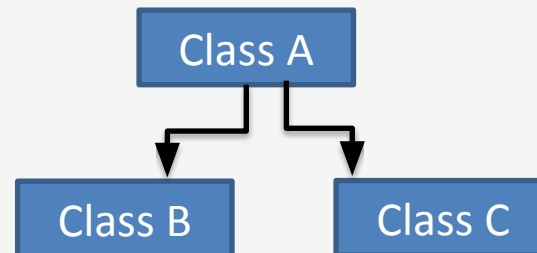
---

The mechanism of deriving a new class from an old one is called **inheritance**.

The old class is referred to as the **base class** and the new one is called the **derived class or subclass**. The derived class inherits some or all of the traits from the base class.

A class can also inherit properties from more than one class or from more than one level.

**Hierarchical Inheritance:** Properties of one class may be inherited by more than one class. This process is known as hierarchical inheritance.



```
#include <iostream>
using namespace std;
class LV1
{
public:
    int a;
    int b;
    void getdata()
    {
        cout<< "Enter value of a and b:\n";
        cin>> a >> b;
    }
};

class LV2 : public LV1
{
public:
    void mul()
    {
        cout<<"\nMultiplication of a & b= " <<  a * b
        <<"\n";
    }
};
```

```
class LV3 : public LV1
{
public:
    void sum()
    {
        cout<<"\nAddition of a and b = " <<a + b;
    }
};

int main()
{
    LV2 O;
    LV3 P;
    O.getdata();
    O.mul();
    P.getdata();
    P.sum();

    return 0;
}
```



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

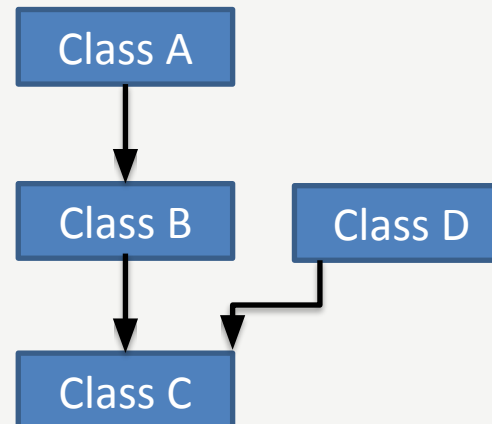
# Hybrid Inheritance in C++

---

The mechanism of deriving a new class from an old one is called **inheritance**.

The old class is referred to as the **base class** and the new one is called the **derived class or subclass**. The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level.

**Hybrid Inheritance:** There could be a situation where we need to apply two or more type of inheritance to design a program, such type of inheritance is known as Hybrid Inheritance.





```
#include<iostream>
using namespace std;
class employee
{
    protected:
        int eid;
    public:
        void getid(int);
        void showid(void);
};

void employee :: getid(int id)
{
    eid = id;
}

void employee :: showid()
{
    cout<<"Employee Id: " << eid<<"\n";
}

class task : public employee
{
    protected:
        int task1;
        int task2;
    public:
        void gettask(int, int);
        void putscore(void);
};
```

```
void task :: gettask(int x, int y)
{
    task1 = x; task2 = y;
}
void task :: putscore()
{
    cout << "Score of task1: "<<tsak1<<"\n";
    cout << "Score of task2: "<<tsak2<<"\n";
}

class add_task
{
    protected:
        int add_task;
    public:
        void getaddtask(int o)
        {
            add_task = o;
        }
        void putaddscore(void);
        {
            cout << "Score of additional task: "<<tsak1<<"\n";
        }
};

class appraisal: public task, public add_task
{
    int grade;
    public:
        void display(void);
};
```

```
void appraisal :: display(void)
{
    grade = task1 + task2 + add_task;
    showid();
    putscore();
    putaddscore();
    cout<<"\nFinal grade:
"<<grade<<"\n";
}

int main()
{
    appraisal A;
    A.getid(705);
    A.gettask(100, 150);
    A.getaddtask(200)
    A.display();
    return 0;
}
```

## OUTPUT

**Employee Id: 705**  
**Score of task1: 100**  
**Score of task2: 150**  
**Score of additional task:**  
**200**



**TOPSTECHNOLOGIES**

Training | Outsourcing | Placement | Study Abroad

# Module – 7

## [C,C++ – Templates]

# Templates in C++

- Templates are powerful features of C++ which allows you to write generic programs.
- In simple terms, you can create a single function or a class to work with different data types using templates.
- Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

1. Function Templates
2. Class Templates



Function Template can work with different data types at once.

A function template defines a family of functions.

Syntax:

```
template <class type> ret-type func-name(parameter list) {  
    // body of function  
}
```