Jump2Learn™
PUBLICATION

www.jump2learn.com

# CONCEPTS of OBJECT ORIENTED PROGRAMMING with DATA STRUCTURE

Jump2Learn - The Online Learning Place

Dr. Yatin K. Solanki | Ms. Nikisha R. Chalakwala | Mr. Chetan N. Rathod

# Beginning With C++

## What is C++?

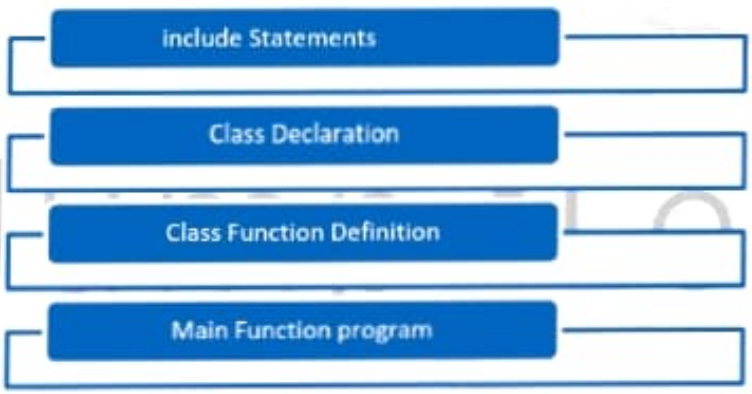
C++ is a general-purpose, case-sensitive, free-form programming language that was developed as an enhancement of the C language to include object-oriented paradigm. C++ was developed by Bjarne Stroustrup. C++ gives programmers a high level of control over system resources and memory.

One of the biggest advantage of C++ is the feature of object-oriented programming which includes concepts like classes, inheritance, polymorphism, data abstraction, and encapsulation that allow code reusability and makes a program even more reliable.

## Application of C++

- C++ is a versatile language for handling very large program.
- We can develop editors, compilers, database, communication systems and any complex real-life application system.
- It allowed creating hierarchy–related objects; we can build special object-oriented libraries which can be used later by many programmers.
- C++ program is easily maintainable and expandable. When new feature needs to be implemented, it is very easy to add to existing structure of object.

## Structure of C++ Program




### Include section:

In C++ program, first section is the include section. It contains pre-processor statements to include various header (library) files.
E.g. to use the input and output object, we use #include<iostream.h>.

**Class Declaration:**

After the include section there is class declaration section. In class declaration we are defining class, its member variable and declaration for the member function.

**Class Function Definition:**

Class member function definition section contains the definition of various class member functions. It defines all member function of class that invoked by object of class in main program.



**Main Function Program:**

At last there is a main function, which start the execution of program and create the object of class.

## Program Features

**Creating source File:**

C++ Programs can be created using any text editor. For example in windows OS, notepad editor use for creating and editing the source code. Also turbo C++ provide an integrated environment for developing and editing programs. The file name should have a proper with file extension .cpp

**Comments:**

C++ Introduce new comment double slash (//). Comment start with a double slash symbol and terminate at the end of the line. Comment may start at anywhere in the line and terminate by end of line. Double Slash (//) comment is single line comment. The C language comments (/*....*/) are still valid and use for Multiline comments in C++.

**Namespace:**

C++ standard introduce new concept Namespace to define a scope for the identifiers that are used in a program. In C++, All Classes, functions and templates are declared within the namespace named std. To use all data member of std namespace in our program, use keyword '**using**'.



**Example:**

```
using namespace std;
```

This statement define that members defined in std namespace will be used frequently in our program and this statement must be written below the include section only.

**Scope Resolution Operator:**

C++ is block structure programming language. We know that same variable name can be used to have different meanings in different blocks. The scope means –where we can access variable and its life span. Usually scope of variable extends from the point of its declaration till the end of block. A variable declared inside a block is said to be local.

**Example:**

```
int x = 10;
int main()
{
    {
     int x=20;
     cout << "Value of global x is " << x;
     cout << "\nValue of local x is " << x;
    }
 cout << "\nValue of global x is " << x;
 return 0;
}
```

**Output:**

```
Value of global x is 20
Value of local x is 20
Value of global x is 10
```

Two delectation of x refer to two different memory location containing different values. The statement in outer block cannot refer to the variable x declare in inner block. Inner block hides a declaration of the same variable in an outer block. If we want to access global version of variable in inner block, we have to use scope resolution (::) operator.

**Example:**

```
int x = 10;
int main()
{
  {
     int x=20;
     cout << "Value of global x is " << ::x;
     cout << "\nValue of local x is " << x;
  }
 cout << "\nValue of global x is " << x;
 return 0;
}
```

**Output:**

```
Value of global x is 10
Value of local x is 20
Value of global x is 10
```

**What is Tokens?**

The smallest individual units in a program are known as tokens. Keywords, Identifiers, Constants, Strings, Operators are examples of C++ tokens.

**What is Keywords?**

Keywords are reserved words which have fixed meanings in C++ and these meanings cannot be changed. They cannot be used for naming identifiers.

**List of Keywords:-**

namespace, using, int, float, char, double, void, bool, short, long, if, else, switch, case, default, for, while, do, break, continue, struct, union, enum, typedef, goto, sizeof, try, catch, throw, new, delete, friend, inline, public, private, protected, operator, virtual, this, class, etc..

**What is Identifiers?**

C++ programs contains elements like variables, arrays, functions, class, etc. Each of them are to be named. Identifiers are names given to these elements.

**What is Constants?**

Constants refer to fixed values that do not change during the execution of a program.

**What is Variable?**

A variable is a data name that may be used to store a data value. The values of variables may change during the execution of a program.

**What is Initialization?**

The process of giving initial values to variable is called initialization.

## Operators and Expressions:

- The symbols which are used to perform logical and mathematical operations in a program are called operators.
- These operators join individual constants and variables to form expressions.
- Operators, constants and variables are combined together to form expressions.
- Consider the expression A + B * 5. Where +, * are operators, A, B are variables, 5 is constant and A + B * 5 is an expression.

### Arithmetic Operators:

These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus.

| | |
|---|---|
| + | Addition or Unary plus |
| - | Subtraction or unary minus |
| * | Multiplication |
| / | Division |
| % | Modulo division |

**Relational Operators:**

These operators are used to compare the value of two variables.

| | |
|---|---|
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| = = | Equal to |
| != | Not equal to |

**Logical Operators:**

| | |
|---|---|
| && | logical AND |
| \| \| | logical OR |
| ! | logical NOT |

- The logical operators && and || are used when we want to test more than one condition at a time and make decisions. Ex:- a>b && c= =10
- An expression which combines two or more relational expressions is termed as a **Logical expression** or **Compound relational expression.**
- Logical expression produces a Boolean result. ( ie:- either true or false)

**Assignment Operators:**

- They are used to assign the result of an expression to a variable.
  = assignment operator
- We have 5 additional assignment operators ( "shorthand" assignment operators)

  +=, -=, *=, /= and %=

**Syntax:** variable operator= expression;

**Ex:** a += 5; It is equivalent to a=a+5

**Increment and Decrement Operators:**

- The increment operator ++ adds 1 to the operand.
- The decrement operator -- subtracts 1 from the operand.

**Syntax:**

++variable or variable++

-- variable or variable --

- The prefix operator (++variable) first adds 1 to the operand and then the result is assigned to the variable on left.

  m = 5;

  y = ++m; here, the value of y and m would be 6.
- The postfix operator (variable++) first assigns the value to the variable on left and then increments the operand.

  m = 5;

  y = m++; here, the value of y would be 5 and m would be 6.

Most of all C operators are valid in C++. In addition, C++ introduces some new operators mentioned in below table:

| | |
|---|---|
| << | insertion operator |
| >> | extraction operator |
| New | memory allocation operator |
| Delete | memory release(deallocation) operator |
| :: | scope resolution operator |
| ::* | pointer-to-member declarator |
| ->* | pointer-to-member operator |



### What is Unary Operator?

A Unary Operator is an operator that takes a single operand in an expression or a statement. ++, --, -, sizeof(), ! are the examples of unary operators.

### What is Binary Operator?

Operators that operate on two operands are known as binary operators.

### What is Operand?

The variables are used in the arithmetic operations are called operand. For example, in the statement A + B, variable A and B are known as operands.

## Flow Control Statements

- Branching (conditional statement)
- Looping
- Jumping statement (unconditional statement)

Branching is deciding what actions to take and looping is deciding how many times to take a certain action. The jump statement cause an unconditional jump to another statement elsewhere in the code.



## Branching

- Branching is so called because the program chooses to follow one branch or another.
- ' If ' statement is known as branching statement.
- There are four kind of 'if' statements
    1. Simple if..... statement
    2. if.....else .. statement
    3. The else if ladder
    4. Nested if statement

### 1. Simple if... statement:

**Syntax:**

```
if (test_expression)
{
        True_block statement(s);
}
```

First **test_expression** will be evaluated, if the **test_expression** is true then the true_block statements are executed.

if the **test_expression** is false then the **nothing will** happened , it simply skip the all statements presents within the **opening** and closing braces.

### 2. The if....else statement:

**Syntax:**

```
if (test_expression)
{
        True_block statement(s);
}
else
{
        False_block statement(s);
}
```

First of all **test_expression** will be evaluated, if the **test_expression** is true then the true_block statements are executed; otherwise the false_block statements are executed

In other words, If condition returns true then the statements inside the body of "if" are executed and the statements inside body of "else" are skipped.

If condition returns false then the statements inside the body of "if" are skipped and the statements inside body of "else" are executed.

### 3. The else if ladder:

It is used when multipath decisions are involved. It is a chain of ifs in which the statement associated with each else is an if.

**Syntax:**

```
if (test1)
{
        Stmt-1;
}
else if (test2)
{
        Stmt-2;
}
```

```
            .....
            .....
            .....
    else if (test n)
    {
            Stmt-n;
    }
    else
    {
            Default stmts;
    }
```

This construct is known as the else if ladder. The test are evaluated from top to downwards as soon as a true test is found, the statement associated with it is executed. (skipping the rest of the ladder). When all the test becomes false then default statements will be executed.

The else..if statement is useful when you need to check multiple conditions within the program. There can be any number of else..if statement in a program. If none of the conditions are met then the statements in else block gets executed.

### 4. Nesting of if...else statements ( Nested If ):

Nested If means If statement within another If statement.

When a series of decisions are involved, we may have to use more than one if...else statements in nested form as follows:

**Syntax:**

```
if (test1)
{
    if (test2)
    {
            Stmt-1;
    }
    else
    {
            Stmt-2;
    }
}
else
{
    Stmt-3;
}
```

First of all test1 will be checked, if the test1 is false then the stmt-3 will be executed; otherwise it continues to check the test2.

if the test2 is true then the stmt-1 will be executed; otherwise the stmt-2 will be executed.

## The Switch Statement

- It is a built-in multiway decision statement.
- It tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.
- Switch case statements are a substitute for long if statements
- A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.
- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Switch is a control statement that allows a value to change control of execution.

**Syntax:**

```
switch (n)
{
  case 1:              // code to be executed if n = 1;
     break;
  case 2:              // code to be executed if n = 2;
     break;
  default:             // code to be executed if 'n' doesn't match any cases
     break;
}
```

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon (:).
- Duplicate case values are not allowed.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the **switch** statement.



## Looping

In looping, a program executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false.

You may encounter situations, when a block of code needs to be executed several number of times.

A loop statement allows us to execute a statement or group of statements multiple times.

## Types of Loops

Depending upon the position of a control statement in a program, a loop is classified into two types:

1. Entry controlled loop

2. Exit controlled loop

- In an **Entry controlled loop,** a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.
- In an **Exit controlled loop,** a condition is checked after executing the body of a loop. It is also called as a post-checking loop.

**We have three types of loop constructs:**

1. The while loop

2. The do-while loop

3. The for loop

## While Loop:

- A while loop is the most straight forward looping structure.

**Syntax:**

```
while (condition)
{
        statements;
}
```

- It is an entry-controlled loop.
- In while loop, a condition is evaluated first. If a condition is true then and only then the body of a loop is executed.
- After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false.
- Once the condition becomes false, the control goes out of the loop.
- After exiting the loop, the control goes to the statements which are immediately after the loop.
- The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory.
- In while loop, if the condition is not true, then the body of a loop will not be executed, not even once.

**Do-While loop:**

- A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop.

**Syntax:**

```
do
{
        Statements;

} while (expression);
```



- In the do-while loop, the body of a loop is always executed at least once. After the body is executed, then it checks the condition.
- If the condition is true, then it will again execute the body of a loop otherwise control is transferred out of the loop.
- Similar to the while loop, once the control goes out of the loop the statements which are immediately after the loop is executed.
- The critical difference between the while and do-while loop is that in while loop the while is written at the beginning. In do-while loop, the while condition is written at the end and terminates with a semi-colon (;)

**For loop:**

- A for loop is a more efficient loop structure.

**Syntax:**

```
for ( initialization; condition; incrementation or decrementation )
{
        statements;
}
```



- Initialization of the control variable is done first, using assignment statement. This Statement is executed only once.
- The value of the control variable is tested using the given condition, it determines when the loop will exit.
- If the condition is true, the body of the loop is executed otherwise the loop is terminated.
- When the body of the loop is executed, the control is transferred back to the for Statement after evaluating the last statement in the loop (increment or decrement).

## Jumping Statement

### goto Statement:

A **goto** statement provides an unconditional jump from the goto to a labeled statement in the same function. **goto statement** is used for altering the normal sequence of program execution by transferring control to some other part of the program. Use of **goto** statement is highly discouraged because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify.

**Syntax:**

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

Where label is an identifier that identifies a labeled statement. A labeled statement is any statement that is preceded by an identifier followed by a colon (:).

## break statement

- An early exit from a loop can be accomplished by using the **break statement.**
- When the **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.
- When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** will exit only a single loop.

**Syntax:**

```
break;
```

## continue Statement

- It is used to skipping a part of a loop.
- During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions.
- The **continue** statement causes the loop to be continued with the next iteration after skipping any statements in between.

**Syntax:**

```
continue;
```

## Manipulators

Manipulators are operators used in C++ for **formatting output**. The data is manipulated by the programmer's choice of display.

endl:

This manipulator has the same functionality as the '\n' (newline character). But this also flushes the output stream.



**Example:**

```
int main()
{
  cout << "Hello" <<endl << "World!";
}
```

Output:

Hello
World!

**Setw:**

- This manipulator sets the minimum field width on output.
- The header file that must be included while using setw manipulator is <iomanip.h>
- It specifies a field width for printing the value of variable.

**Example:**

```
#include <iostream>
#include <iomanip>

void main( )
{
   int x1=123,x2= 124, x3=89;
   cout << setw(20) << "Subject" << setw(10) << "Marks" << endl
        << setw(20) << "SE" << setw(10)<< x1 << endl
        << setw(20) << "OOP with DS" << setw(10)<< x2 << endl
        << setw(20) << "Python" << setw(10)<< x3 << endl;
}
```



Output:

```
          Subject      Marks
               SE        123
      OOP with DS        124
           Python         89
```

**setprecision:**

- The setprecision Manipulator is used with floating point numbers.
- It is used to set the number of digits printed to the right of the decimal point.
- the header file that must be included while using *setprecision* manipulator is <iomanip.h>

**Example:**

```
#include <iostream>
#include <iomanip>
void main( )
{
   float x = 0.1;
   cout << "x="<<setprecision(3) << x << endl;

   cout <<  "x="<<x << endl;              //default 6 precision after decimal point
}
```

**Output:**

X=0.100

X=0.100000

**setfill:**

This is used after setw manipulator.

If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

- the header file that must be included while using setfill manipulator is <iomanip.h>

**Example:**

```
#include <iostream>
#include <iomanip>
void main()
{
   cout << setw(15) << setfill('*') << 1297 << endl;
}
```

**Output:**

**********1297

# Unit - 1

# Concepts of OOPS



- Program, Programming
- Evaluation of Software
- Procedure Oriented Programming
- Object Oriented Programming
- Differences between POP and OOP
- Differences between C and C++
- Basic Concepts of Object Oriented Programming
- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing
- Structure vs. Class
- Benefits of Object Oriented Programming (OOP)
- Various Library (Header) Files of C++
- Data Types of C++
- Output Operator
- Input Operator
- Cascading of I/O Operators
- Concepts of String
- Pointer to Character Array
- The Header File "String.h" and String Handling Functions
- Concepts of Class and Objects
- Inline Function

## What is Program?

- A Program is a collection of instructions that can be executed step by step to tell the computer precisely what action to perform.
- A Program is usually written by a computer programmer in a programming language. From the program in its human-readable form of source code,- a program written in high level language by human.
- A compiler or assembler can derive machine code — a form consisting of instructions that the computer can directly execute.

## What is Programming?

- Programming is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task
- Programming involves tasks such as: analysis, generating algorithms, profiling algorithms' accuracy and resource consumption, and the implementation of algorithms in a chosen programming language (commonly referred to as coding).
- The source code of a program is written in one or more languages that are intelligible to programmers, rather than machine code, which is directly executed by the central processing unit (CPU).
- The purpose of programming is to find a sequence of instructions that will automate the performance of a task (which can be as complex as an operating system) on a computer, often for solving a given problem.

## Evaluation of Software

- Software: - A collection of program that tells a computer how to perform such activity.
- Ernest R. Tello, a well - known writer compared the evolution of software technology to the growth of a tree. Like a tree, the software evolution has distinct phases or "layer" of growth. These Layers were built up one by one over the last five decades with each layer representing an improvement over the previous one.
- Alan Kay, one of the promoters of the object-oriented paradigm and the principal designer of Smalltalk, has said: "As complexity increases, architecture dominates the basic materials".
- To build today's complex software it is just not enough to put together a sequence of programming statement and sets of procedures and modules; we need to incorporate such technique and program structure that are easy to comprehend, implement and modify.
- Since the invention of the computer; many programming approaches have been tried. These include techniques such as modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each has been to concern to handle the increasing complexity of program that is reliable and maintainable. These techniques have become popular among programmers over the last three decades.

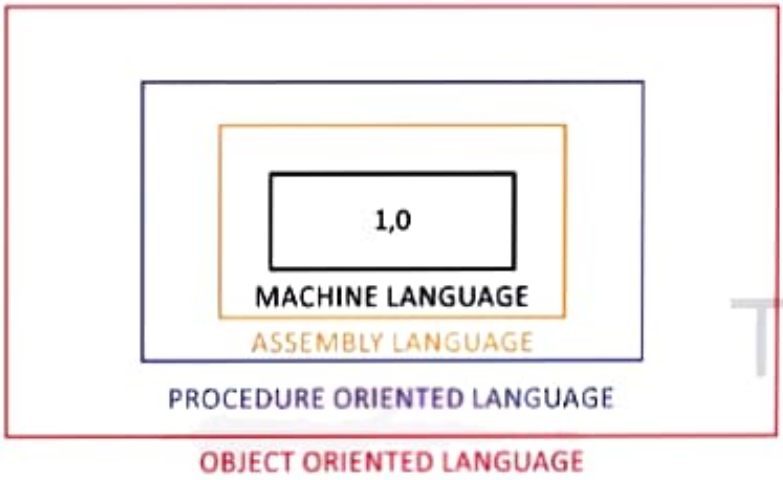



**FIGURE 1.1 SOFTWARE EVOLUTIONS**

- With the mother language such as C, Structure programming became very popular and was the main technique of the 1950s. Structure programming was a powerful tool that enabled programmers to write moderately complex program easily. As programmer grew larger, structure approach failed to show the desired result in terms of bug-free, easy to maintain and reusable programs.
- Object Oriented Programming (OOP) is an approach to organize and develop the program that attempt to eliminate some of loopholes of conventional programming and adding some more powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

## Procedure Oriented Programming

- Procedural programming is a programming paradigm, derived from structured programming, based on the concept of the procedure call. Procedures, also known as routines, subroutines, or functions,
- Conventional programming, using high level language such as COBOL, FORTRAN and C is commonly known as procedure oriented programming (POP).
- In procedure oriented approach, the problem is carried out sequence of steps such as reading, calculating and printing to accomplish for specific task. These operations are converted into number of functions.
- The primary focus is on "Function".
- Simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or it.
- Computer processors provide hardware support for procedural programming through a stack register and instructions for calling procedures and returning from them.

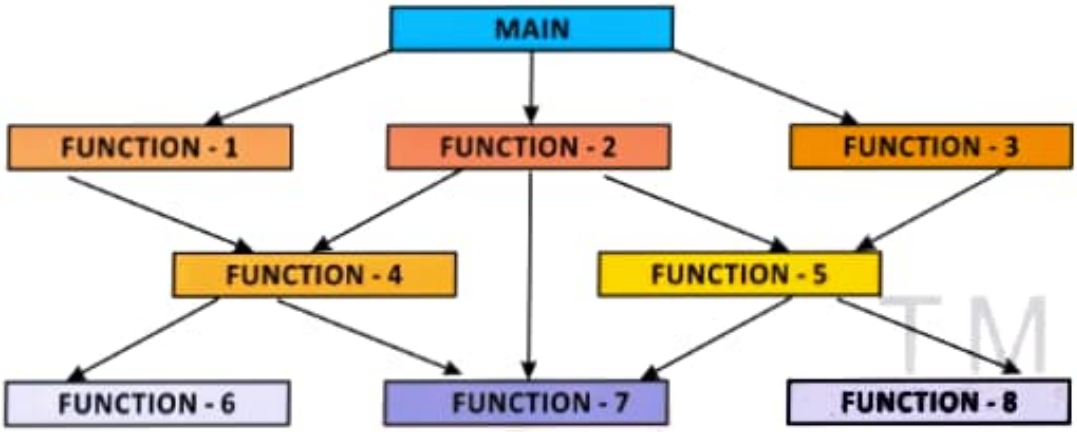



**FIGURE 1.2 STRUCTURE OF PROCEDURE ORIENTED PROGRAMMING**

- POP basically consists of writing a list of instruction (or action) for the computer to follow and organizing these instruction into groups known as functions. We normally use a flowchart to organize these actions and represent the flow of control from one action to another. While we concepting on the development of function very little attention is given to the data that are being used by various function. What happens to the data? How are they affected?
- In multi-function program, many important data items are places as global therefore global data may be accessed by all the function. In a large program it is very difficult to identify what data is used by which function that also lead to bugs creep in.

## Characteristics of Procedure Oriented Programming

- Emphasis is on Function.
- Large program are divided into smaller program known as function.
- Most of the function share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Top-down approach in program design.



## Object Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).
- The invention of object oriented approach is to remove some loopholes reside in procedure approach.

- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. OOP allows decomposition of the problem into a number of entities called object and then builds data and functions around these objects.
- The data of an object can be accessed by the function associated with that object. However function of one object can access the function of other objects.

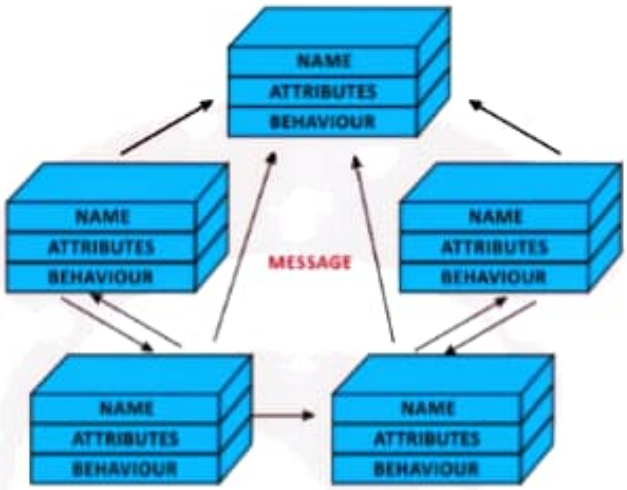



**FIGURE 1.3 OBJECT ORIENTED PROGRAMMING**

- In this era, Object Oriented paradigm is most popular than other due to its working feature. The way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand. Hence object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since each and every object having their independent memory so that they cannot affect and modified each other data.

## Characteristics of Object Oriented Programming

- Focus on data rather than Functions
- Programs are divided into small parts which are known as Objects.
- Data structures are designed such that they characterize the objects.
- A function that operates on the data of an object is tied together in the data structure.
- Data is hidden and cannot be accesses by external functions
- Object may communicate with each other through functions.
- New data and function can be added easily whenever necessary.
- Bottom-up approach in Program design.

## Differences between POP and OOP

- POP creates a step by step Program that guides the application through a sequence of instructions. Each instruction is executed in order.
- POP also focuses on the idea that all algorithms are executed with functions and data that the programmer has access to and is able to change.
- OOP is much more similar to the way the real world works; it is analogous to the human brain. Each program is made up of many entities called objects.
- A message must be sent requesting the data, just like people must ask one another for information; we cannot see inside each other's heads.

| PROCEDURE ORIENTED PROGRAMMING | OBJECT ORIENTED PROGRAMMING |
|---|---|
| POP, Program is divided into small Parts, called Functions. | In OOP, program is divided into Parts, called Object. |
| POP follows Top-Down approach | OOP follows Bottom Up approach |
| POP does not have any proper way for hiding data so it is less secure. | OOP provides data hiding so provides more security |
| In POP, Overloading is not possible | In OOP, Overloading is possible in the form of function overloading and Operator Overloading. |
| In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system | In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data. |
| POP does not have any access Specifier | OOP has access specifiers named Public, Private, Protected, etc. |

**TABLE 1.1 DIFFERENCES BETWEEN POP AND OOP**



## Differences between C and C++

- C++ is an extension of C language. This means that you can not only use the new features introduced with C++ but can also use the power and efficiency of C language.
- The main difference between C and C++ is that C++ is object oriented while C is procedure oriented. Object oriented programming paradigm is focused on writing programs that are more readable and maintainable. It also helps the reuse of code by packaging a group of similar objects or using the concept of component programming model. It helps thinking in a logical way by using the concept of real world concepts of objects, inheritance and polymorphism.

| C | C++ |
|---|---|
| C was developed in 1969 at AT&T Bell Labs by Dennis Ritchie. | C++ was developed in 1979 by BjarneStroustrup. |
| C is a subset of C++ language. | C++ is a superset of C. You can run most of the C code in C++ but reverse is not possible. |
| Procedural programming is supported in C. | Procedural and Object oriented programming both are supported in C++. |
| C is a function driven language. | C++ is an object driven language. |
| sData and functions are separate and free entities. | Data and functions are encapsulated in the form of an object. Class is a blueprint of the object. |
| Does not support information hiding. | Encapsulation hides the data which can be used for information hiding. |
| Multiple Declarations of global variables are allowed. | Multiple Declarations of global variables are not allowed. |
| Function overloading and Operator overloading is not supported. | Function overloading and Operator overloading is supported. |
| Functions cannot be defined inside structures. | Functions can be defined inside a structure. |
| Namespace feature is not provided. | Namespace is allowed to avoid name collisions. |
| scanf is used for input where printf is used for output. | cin is used for input and cout is used for output. |
| Reference variables are not supported. | Reference variables are supported. |
| Virtual and Friend functions are not supported. | Virtual and Friend functions are supported. |
| malloc() and calloc() functions are used for dynamic memory allocation where as free() function is used for memory de-allocation. | "new" operator is used for memory allocation and "delete" operator is used for memory de-allocation. |
| No support for object oriented programming. So there is no support for polymorphism, encapsulation, and inheritance. | C++ language supports polymorphism, encapsulation, and inheritance which are part of object oriented programming. |
| main() function can be called from other Functions. | We cannot call main() function from other functions. |
| All variables must be defined at the starting of a scope. | You can declare variables anywhere in C++. |
| Inheritance is not possible. | Inheritance is allowed in C++. |
| Exception handling is not supported. | Exception handling is supported in C++. |

**TABLE 1.2 DIFFERENCES BETWEEN C AND C++**

## Basic Concepts of Object-Oriented Programming

- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing



### 1) Objects:

- Objects are the basic run-time entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.
- Programming problem is analyzed in terms of objects and the nature of communication between them.
- Objects take space in the memory and they have an associated address like a structure in C.
- When a program is executed, the objects interact by sending messages to one another.
- For example, if "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance.
- Each object contains data and code to manipulate the data. Objects can communicate without having to know details of each other's data or code. It is sufficient to know the type of message accepted and returned by the objects.

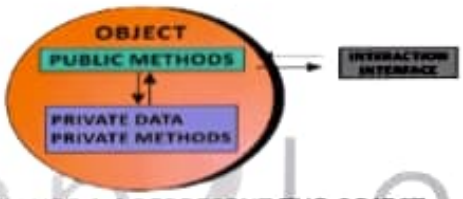



**FIGURE 1.4 REPRESENT THE OBJECT**



### 2) Classes

- Classes are user-defined data types and behave like the built-in data types.
- We know that objects contain data and code to manipulate that data. The entire set of data and Code of object can be made a user-defined data type with the help of a class.
- Once a class has been defined, we can create any number of objects belonging to that class.
- Each Object is associated with the data of class type with which they are created.
- Thus, a class is a collection of objects of similar type. For example:- mango, apple and orange are members of the class fruit.
- If fruit has been defined as a class then the statement **fruit apple;** will create an object **apple** belonging to the class **fruit.**


**FIGURES 1.5 REPRESENT THE CLASS**

### 3) Data Abstraction and Encapsulation:

- The wrapping up of data and functions into a single unit (called class) is known as encapsulation.
- Data encapsulation is the most striking feature of a class.
- The data can be access by only those functions which are wrapped in the class.These functions provide the interface between the object's data and the program.
- This prevention of the data from direct access by the program is called data hiding or information hiding.
- Abstraction refers to the act of representing essential features without including the background details.
- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size,total, rate etc... and functions to operate on these attributes.
- The attributes are sometimes called data member because they hold information and the functions are called methods or member functions.

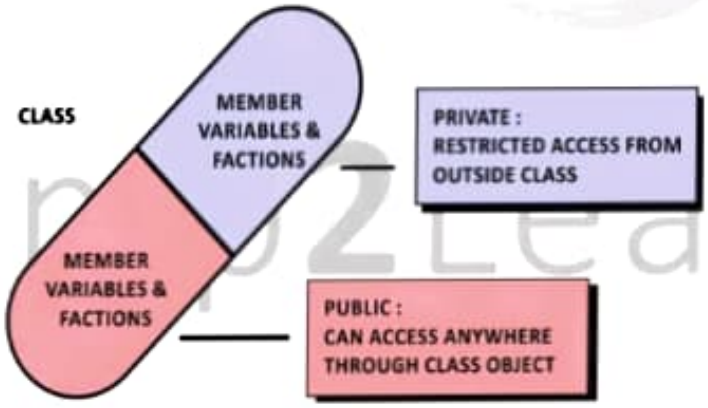
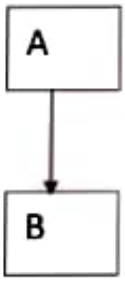


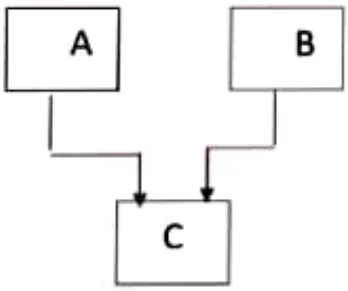

**FIGURE 1.6 REPRESENT DATA ENCAPSULATION**

**4) Inheritance:**

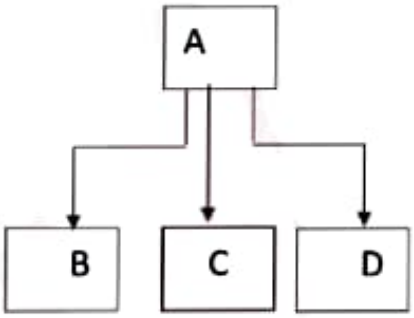
- The mechanism of deriving a new class from an existing ( old ) class is called inheritance.
- The old class is known as base class, super class or parent class and new class is known as derived class, subclass or child class.
- The concept of inheritance provides the idea of reusability.
- This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing class.
- The new class will have the combined features of both the classes.
- Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification.



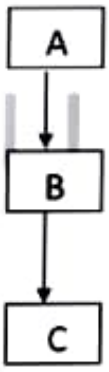

**SINGLE INHERITANCE**



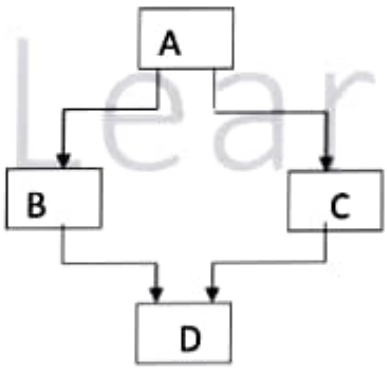

**MULTIPLE INHERITANCE**




**HIERARCHICAL INHERITANCE**




**MULTILEVEL INHERITANCE**




**HYBRID INHERITANCE OR MULTIPATH INHERITANCE**

- A derived class with only one base class is called single inheritance.
- A derived class with several base classes is called multiple inheritance.
- The features of one class may be inherited by more than one class. This process is known as hierarchical inheritance.
- The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance.
- There could be situations where we need to apply two or more types of inheritance to design a program. This type of inheritance is known as hybrid inheritance.

**5) Polymorphism :**

- Polymorphism is one of the most important concepts of OOP.
- Polymorphism means the ability to take more than one form.
- An operation may be exhibit different behaviors indifferent instances. The behavior depends upon the types of data used in the operation.
- Consider the example of addition: -for two numbers the operation will generate a sum of that numbers but if the operands are string then the operation will produce a third string by concatenation.
- The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.
- Using a single function name to perform different types of tasks is known as function overloading.
- Polymorphism is extensively used in implementing inheritance.

**6) Dynamic Binding:**

- Dynamic binding is also known as late binding. It means that the code associated with a given Procedure call is not known until the time of the call at run-time.
- It is associated with polymorphism and inheritance.

**7) Message Passing:**

- An object-oriented program consists of a set of objects that communicate with each other.
- Objects communicate with one another by sending and receiving information, same way as people pass messages to one another.
- A message for an object is request for execution of a procedure, therefore will invoke a function (procedure) in the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Objects have a life cycle. They can be created and destroyed. Communication with an object is possible as long as it is alive.


**FIGURE 1.7 MESSAGE PASSING**

## STRUCTURE VS. CLASS

| STRUCTURE | CLASS |
|---|---|
| Structure is a collection of different types of data type. | Class is group of common objects that share common properties and relationships. |
| Structure is a value type that is why its object is created on the stack memory. | Class is reference type and its object is created on the heap memory. |
| Structure members are only public specifier. | Class members have public, private and Protected specifier but default is private. |
| Structure contains only data member. | Class contains data member and member function bind into single unit for data hiding. |
| Structure cannot be inherited. Because all data member public which can use any where | Class can be inherited due to achieve privacy and data hiding features. |
| Structure does not have constructor because it have only data member and constructor is a function which call automatically. | Class has Constructorfor automatic initialization when object create. |
| Structure should be used when you want to use a small data structure | Class is better choice for complex data structure. |

**TABLE 1.3 DIFFERENCES BETWEEN STRUCTURE AND CLASS**

## Benefits of Object Oriented Programming (OOP)

- Through inheritance, we can eliminate redundant code and extend the use of existing classes which is not possible in procedure oriented approach.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch which happen in procedure oriented approach. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of object to co-exist without any interference.

- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable from.
- Object oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

## Various Library (header) files of C++

Header files are used in the programs which contains definition or implementation of the predefine functions. Many functions stored in the C library. Library functions are grouped category-wise and stored in the different files known as **header files.** If we want to access the functions stored in the library, it is necessary to tell theCompiler about the files to be accessed.This is achieved by using the pre-processor directive **#include** as follows :

#include< filename >

Filename is the name of the library file that contains the required function definition.

**Syntax**

#include<iostream.h>

Or

#include"iostream.h"

**Types of header files in C++**

1. **System header files** – These are predefined header files presents in the compilers. They are also known as pre-existing header files.
2. **User header files** – These are user defined header file includes in the programs by #define directive.



**Standard Header Files and their Uses:**

**#include<iostream.h>** : It is used as a stream of Input and Output using **cin** and **cout**.

**#include<string.h>** : It is used to perform various functionalities related to string manipulation like strlen(), strcmp(), strcpy(), strcat(), etc.

**#include<math.h>** : It is used to perform mathematical operations like sqrt(), log2(), pow(), etc.

**#include<iomanip.h>** : It is used to access set() and setprecision() function to limit the decimal places in variables.

**#include<signal.h>** : It is used to perform signal handling functions like signal() and raise().

**#include<stdarg.h>** :It is used to perform standard argument functions like va_start() and va_arg(). It is also used to indicate start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.

**#include<exception.h>** : This defines library for utilities of Exception handling.

**#include<system_error.h>:** This defines library for platform-dependent error code,std::error_code.

**#include<errno.h>** : It is used to perform error handling operations like errno(), strerror(), perror(), etc.

**#include<istream.h>** : It is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as get, getline, read, etc.

**#include<ostream.h>** : It is responsible for handling output stream. It provides number of function for handling chars, strings and objects such as write, put, etc.

**#include<fstream.h>** : It is used to control the data to read from a file as an input and data to write into the file as an output.

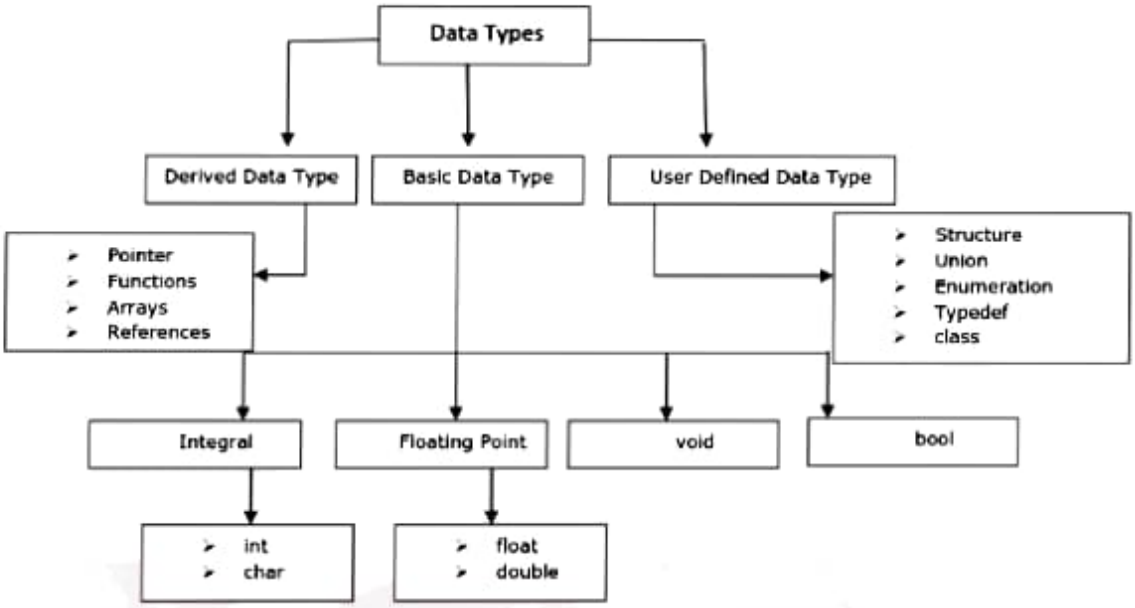
**#include<time.h>** : It is used to perform functions related to date() and time() like setdate() and getdate(). It is also used to modify the system date and get the CPU time respectively.



## Introduction to C++ Data Types

Data types define the type of data that a variable can hold. It can be a built-in data type like integer, float, double, char, or derived data types like arrays, pointers and functionsor can be a user-defined data type like structure, union, enum. Data types should be defined before the execution as it informs the compiler the type of data specific variables holds. Integer data type can hold only integer values; it cannot hold the float values or character values.

A Data type is to let know the variable, what type of element it is and definitely going to determine the memory allocation of that variable. We know that each data type has a different memory allocation.




There are three different C++ data types namely; Primitive, Derived and User Defined.

## 1 Primitive Data Types (Basic Data Type)

These are pre-defined in c++. It is also called the built-in data types. We can directly use them to declare the variables.

**Integer:**

- All c compilers offer different integer data types. They are short and long in both signed and unsigned forms. Short integer requires half the space in memory than the long and integer.
- Integer with sign is called signed integer.
- The signed integer has signs positive or negative.
- The signs are represented in the computer in the binary format as 1 for – (minus) and 0 for + (plus).
- We declare long and unsigned integers to increase the range of values.
- The use of qualifier signed on integers is optional because the default declaration assumes a signed number.

| Type | Size (byte) | Range |
|---|---|---|
| int or signed int | 2 | -32,768 to 32,767 |
| unsigned int | 2 | 0 to 65,535 |
| short int or signed short int | 1 | -128 TO 127 |
| unsigned short int | 1 | 0 to 255 |
| long int or unsigned long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |

Usually defined by the keyword "Int". We can declare integer variable as below:

int age;

Here, "**age**" is an integer type of variable.

**Character:**

- A single character can be defined as a character (char) type data.
- Characters are usually stored in 8 bits (one byte) of internal storage.
- Usually defined by the keyword "**char**". We can declare character variable as below:

  char ch;

Here, "**ch**" is acharacter type of variable.

| Type | Size (byte) | Range |
|---|---|---|
| char or signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |

**Floating Point:**

- Floating point numbers are stored in 32 bits (4 byte), with 6 digits of precision.
- Floating point numbers are defined in C by float keyword.
- When accuracy provided by a float number is not sufficient, the type double can be used to define the number.
- Usually defined by the keyword "float". We can declare float variable as below:
- float per;
  - Here, "per" is a float type of variable.

Double Floating Point:

- Usually defined by the keyword "double". We can declare double variable as below:

  double unit;

  Here, "**unit**" is a double type of variable.



- **double** is more precise than **float** and can store 64 bits, double of the number of bits **float** can store. Double is used for storing large numbers, we prefer double over float.

**void**: void means without any value. "**void**" datatype represents a valueless entity. void data type is used for those function which does not returns a value.

The ISO/ANSI C++ Standard has added certain new data types to the original C++ specifications.They are presented to provide better control in certain situations as well as for providing conveniences to C++ programmers.One of the new data type is: **bool**

```
bool result = true;
```

**Boolean:** Usually defined by the keyword "**bool**". Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. The values *true* or *false* have been added as keywords in the C++ language.We can declare boolean variable as below:

Here, "result" is a boolean variable initialized with true value



**Important Points about Boolean type:**

- The default numeric value of true is 1 and false is 0.
- We can use bool type variables or values true and false in mathematical expressions also.For instance,

```
int x = false + true + 6;
```

Above statement is valid and the expression on right will evaluate to 7, because false hasvalue 0 and true will have value 1.

It is also possible to convert implicitly the data type integers or floating point values to bool type. For Example:

```
bool  p = 0;        // false
bool  q = 10;       // true
bool  r = 12.35;    // true
```

Note: Boolean data type may not work in old C++ editor. It is newly introduced by the ISO / ANSI C++ standard.

### 1. Derived Data Types

The data types that are derived from the primitive or built-in data types are known as derived data types.

**Array:** An array is a collection of items stored at continuous memory locations. The idea of array is to represent many instances in one variable.

**Syntax:**

```
DataTypeArrayName [size_of_array];
```



**Example:**

```
int  a[5];
```

Here, '**a**' is integer array and we can store maximum 5 integer elements in array '**a**'.

**Pointer**: Pointer refers to a variable that holds the address of another variable. Likeregular variables, pointers have a data type. For example, a pointer of type integer can hold the address of a variable of type integer. In C++, a pointer holds the address of an object stored in memory. The pointer then simply "points" to the object. The type of the object must correspond with the type of the pointer. A variable can be declared as pointer by putting '*'in the declaration.

**Syntax:**

```
DataType  *var_name;
```

**Example:**

```
int *ptr;
```

Here, "**ptr**" is a pointer variable and it points to an address which holds integer data.

**Function:** A function is a block of code or program-segment that is defined to perform a specific task. A function is generally defined to save the user from writing the same lines of code again and again for the same input. All the lines of code are put together inside a single function and this can be called anywhere required. main() is a default function that is defined in every program of C++.

**Syntax:**

```
returnType functionName (argument_ list);
```

**Example:**

```
int max (int a , int b);
```

Here, '**max**' is a name of the function. **a** and **b** are integer type of arguments. The above function '**max**' will return integer value.

**Reference**: When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

**Syntax:**

```
DataType& reference_var_name = Existing_var_name;
```

**Example:**

```
int& ref = a;
```

Here, '**a**' is integer type of variable and it is already declared. After executing above statement whenever we change value of anyone among '**a**' and '**ref**' then values of both variables will be change accordingly.

## 1. User-Defined Data Types

The data types that are defined by the user are called the user-defined data type.

**Structure**: Storing the combination of either similar or different data types under continuous memory locations. As we already saw, in arrays we can store only items with similar data types. But structures can store different data types.

**Example:**

```
struct student
{
   int rollno;
   char name[20];
   float per;
```

```
};
```

**Union:** It is similar as structure but all members of Union share the same memory location whereas members of the structure stored at different memory location.

**Example:**

```
union  student
{
  int rollno;
  char  name[20];
  float  per;
};
```



**Class:** It is defined in the object-oriented programming.It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance (objects) of that class. A class is like a blueprint for an object.

**Syntax:**

```
classclass_name
{
    DataMember_1;
    DataMember_2;
    ----------
    ----------
    DataMember_N;
    Public:
        MemberFunction_1;
        MemberFunction_2;
        ----------
        ----------
        MemberFunction_N;
};
```



**Enumeration:** Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.The keyword "enum" is used to declare an enumeration.These are generally used when we already know a set of values for a particular variable and choose a single value from them.

**Syntax:**

```
enum enum_name  {const1, const2,....., constN};
```

Here, enum_name is any name given by the user and const1, const2.... are the values of type flag.

**Typedef:** This data type is for just giving a new or a different name to the data types. C++ allows you to define explicitly new data type names by using the keyword typedef. Using typedef does not actually create a new data type, rather it defines a name for an existing type.

**Syntax:**

typedef type name;

Here, **type** is any C++ valid data type and **name** is the new name for this data type.

**Difference between Structure and Union**

| Structure | Union |
|---|---|
| Each member of the structure occupies and uses its own memory space. | All members of a union use the same memory space. |
| Its required more memory space due to occupy space for each member individually | Its required less memory space due to share same memory space among all members of union |
| All the member of structure can be initialized | Only one member of union can be initialized |
| Total memory size of structure is the summation of memory sizes of all its members. | Total memory size depends on the memory size of its largest elements. |
| Keyword: **struct** | Keyword: **union** |

**OUTPUT OPERATOR**

- The statement **cout<< "c++ is better than C";** causes the string in quotation marks to be displayed on the screen.
- This statement introduces two new C++ features, **cout** and **<<**.
- The identifier **cout** (pronounced as 'C out') is a predefined object that represents the standard output stream in C++.
- Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices.
- The operator **<<** is called the **insertion or put to operator**.

**Syntax:**

cout<< variable1[<<variable2<<variable(N)];

- The object cout has a simple interface. If **str** represents a string variable, then the following statement will display its Contents:

**Example:**

cout<<str;

## INPUT OPERATOR

- The statement **cin>> no1;** is an input statement and causes the program to wait for the user to type in a number.
- The number keyed in is placed in the variable no1.
- The identifier **cin** (pronounced as 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.
- The operator >> is known as **extraction or get from operator.**
- It extracts (or takes) the value from the keyboard and assigns it to the variable on its right.

**Syntax:**

cin>> variable1[>>variable2>>variable(N)];

**Example:**

cin>>str

## CASCADING of I/O Operators

- We have used the insertion operator **<<** repeatedly for printing results.
- The statement **cout<< "Sum = " << sum << "\n";**first sends the string "Sum=" to cout and then sends the value of sum.
- Finally, it sends the newline character so that the next output will be in the new line.
- The multiple use of <<one statement is called cascading.
- When cascading an output operator, we should ensure necessary blank spaces between different items.
- We can also cascade input operator >> as shown below:

  **cin>> number1 >> number2;**
- The values are assigned from left to right. That is, if we enter two values, say, 10 and 20, then 10 will be assigned to number1 and 20 to number2.



## Concepts of String

Strings are actually one-dimensional array of characters terminated by a null character ('\0'). Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "India". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "India."

```
char country[6] = {'I', 'n', 'd', 'i', 'a', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows:

```
char  country[ ] = "India";
```

**Character Array:**

A character array is a sequence of characters, just as a numeric array is a sequence of numbers. A character array is an array whose elements are of the type "char". Normally, strings are implemented with character arrays.

**Declaration of strings:**

Declaring a string is as simple as declaring a one-dimensional array. Basic syntax for declaring a string is as below:

```
char str_name[size];
```

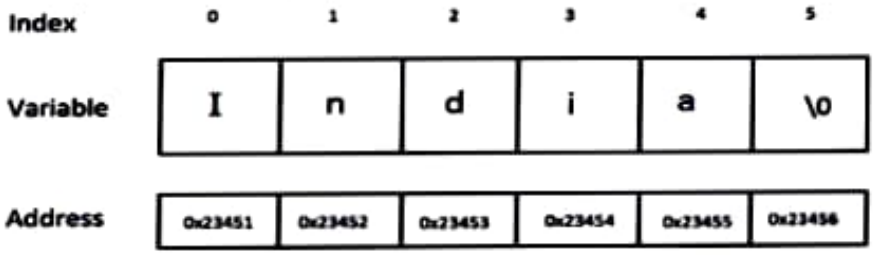
In the above syntax str_name is any name given to the string variable and size is used to define the length of the string, i.e the number of characters strings will store. Remember that there is an extra terminating character which is the Null character ('\0') used to indicate the termination of string which differs strings from normal character arrays.

**Initializing a String:**

A string can be initialized in different ways. Below is an example to declare a string with name as "**country**" and initialize it with "**India**".

1. char country[] = "India";
2. char country[50] = "India";
3. char country[] = {'I','n','d','i','a','\0'};

Following is the memory representation of the above defined and initialized string:




Actually, we do not place the 'null' character at the end of a string constant. The compiler automatically places the '\0' (null character) at the end of the string when it initializes the array.

**Example:**

```
int main()
{
char country[6] = {'I', 'n', 'd', 'i', 'a', '\0' };
cout<<"Country Name is :"<<country;
return 0;
}
```

**Output:**

Country Name is : India

**Pointer to character array**

A Pointer is a variable which holds the address of another variable of same data type. Pointers are used to access memory and manipulate the address. Pointers are one of the most distinct and exciting features. It provides power and flexibility to the language.

When an array is declared, compiler allocates sufficient memory to contain all its elements. Its base address is also allocated by the compiler.
Consider following declaration:
char country[6] = {'I', 'n', 'd', 'i', 'a', '\0' };

Suppose the base address of array "country" is 1000 and each character requires one byte, all the characters will be stored as follows:

| | I | n | d | i | a | \0 |
|---|---|---|---|---|---|---|
| | country[0] | country[1] | country[2] | country[3] | country[4] | country[5] |
| **Address** | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 |

Array Variable "country" will give the base address, which is a constant pointer pointing to country[0]. Hence "country" contains the address of country[0] i.e 1000.

"country" has two purpose -

- It is the name of the array
- It acts as a pointer pointing towards the first character in the array.

country is equal to &country[0] by default.

**Example:**

```
int main()
{
        char country[6] = {'I', 'n', 'd', 'i', 'a', '\0' };
        char *ptr = country;
        for(int i = 0; i< 6; i++)
        {
                cout<<*ptr;
                ptr++;
        }
        Return 0;
}
```

**Pointer and Character strings:**

Pointer is used to create strings. Pointer variables of char type are treated as string.

```
char *str = "Computer";
```

The above code creates a string and stores its address in the pointer variable **str**. The pointer **str** now points to the first character of the string "Computer".The string created using char pointer can be assigned a value at runtime.

```
char *str;
str = "Computer";
```

The content of the string can be printed as below:

```
cout<<str;
```

**str** is a pointer to the string and also name of the string. Therefore we do not need to use indirection operator *.

## The header file "string.h" and Its Important Functions

The **string.h** header defines various functions for manipulating arrays of characters.

**Some of the most commonly used String functions are:**

| Sr.No. | Function & Purpose |
|---|---|
| 1 | **strcpy(str1, str2);**<br><br>Copies string str2 into string str1. |
| 2 | **strcat(str1, str2);**<br><br>Concatenates string str2 onto the end of string str1. |
| 3 | **strlen(str1);**<br><br>Returns the length of string str1. |
| 4 | **strcmp(str1, str2);**<br><br>Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2. |
| 5 | **strrev(str1 );**<br><br>Returns the reverse string of a given string str1. |

**strcpy():**This functionis used to copy one string to another.

**Syntax:** strcpy(dest, src);

This function accepts two parameter dest and src. 'dest' is pointer to the destination array where the content is to be copied and 'src' is string which will be copied.After copying the source string to the destination string, the strcpy() function returns a pointer to the destination string.

**Example:**

```
int main()
{
        char str1[20] = "Hello";
        char str2[20] = "Jump2Learn";

        strcpy(str1, str2);

        cout<<"String 1 = "<<str1<<"\n";
        cout<<"String 2 = "<<str2;

        return 0;
}
```

**Output**

```
String 1 = Jump2Learn
String 2 = Jump2Learn
```

**strcat():** This function will append a source string to the end of destination string.

**Syntax:** strcat ( dest , src );

The strcat() function takes two arguments **dest** and **src**. It will append copy of the source string in the destination string. The strcat() function returns dest, the pointer to the destination string.

**Example:**

```
int main()
{
        char str1[20] = "Hello ";
        char str2[20] = "Students";

        strcat(str1, str2);

        cout<<"String 1 = "<<str1<<"\n";
        cout<<"String 2 = "<<str2;
        return 0;
}
```

**Output:**

```
String 1 = Hello Students
```

String 2 = Students

**strlen():** This function willcalculates the length of the given string. It doesn't count null character '\0'. But it counts blank spaces.

**Syntax:** strlen ( str );

The strlen() function takes one argument **str**, which represents the string variable whose length we have to find. This function returns the length of passed string.

**Example:**

```
int main()
{
        char str1[30] = "Every Student is Special";
        int l;

        l = strlen(str1);

        cout<<"Length of the given String is : "<<l;
        return 0;
}
```

**Output:**

Length of the given String is : 24

**strcmp():**This function takes two strings as arguments and compare these two strings lexicographically.

**Syntax:** strcmp ( str1, str2 );

The function srtcmp takes two strings as parameters and returns an integer value based on the comparison of strings.

- strcmp() compares the two strings lexicographically means it starts comparison character by character starting from the first character until the characters in both strings are equal or a NULL character is encountered.
- If first character in both strings are equal, then this function will check the second character, if this is also equal then it will check the third and so on.
- This process will be continued until a character in either string is NULL or the characters are unequal.

**Example:**

```
int main()
{
        char str1[20] = "KRIYATI";
        char str2[20] = "KRIYATI";
        int res;
        res = strcmp(str1, str2);
        if(res == 0)
```

```
            cout<<"Both Strings are Equal";
        else
            cout<<"Both Strings are Different";
        return 0;
    }
```

**Output:**

Both Strings are Equal

**strrev():** The strrev() function is used to reverse the given string.

**Syntax:** strrev ( str );

The strrev() function takes one argumentstr, that represents the given string which is needed to be reversed.This function returns the string after reversing the given string.

**Example:**

```
int main()
{
    char str1[20] = "VALSAD";

    cout<<"The given String is : "<<str1;
    cout<<"\nTheReversed String is : "<<strrev(str1);

    return 0;
}
```

**Output:**

The given String is : VALSAD
The Reversed String is :DASLAV

**Note:** Remember that we need to include **string.h** header file in our program when you want to use any of string handling functions discussed above.

## CONCEPTS OF CLASS AND OBJECTS



### CLASSES AND OBJECTS

- The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.
- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one package.
- The data and functions within a class are called members of the class.

**Class Definitions:**

- Class is a user define data types but it behave likes a built in data types.
- When you define a class, you define a blueprint for a data type.
- This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.
- A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces.
- A class definition must be followed either by a semicolon or a list of declarations.

**Syntax:**

```
classclass_name
{
        Data member1;
        .....
        Data memberN;
    public:
        Member function1;
        .....
        Member functionN;
};
```

**For example,** we defined the Box data type using the keyword **class** as follows –

```
class Box
{
    public:
        double length;  // Length of a box
        double breadth;  // Breadth of a box
        double height;  // Height of a box
};
```

- The keyword **public** determines the access attributes of the members of the class that follows it.
- A public member can be accessed from outside the class anywhere within the scope of the class object.
- You can also specify the members of a class as **private** or **protected.**

**Define Objects:**

- A class provides the blueprints for objects, so basically an object is created from a class.
- We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types.

- Following statements declare two objects of class Box –

```
Box Box1;       // Declare Box1 of type Box
Box Box2;       // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

**Accessing the Data Members:**

The public data members of objects of a class can be accessed using the direct member access operator (.)

```
#include<iostream.h>
#include<conio.h>
class Box
{
   public:
          double length;                    // Length of a box
          double breadth;                   // Breadth of a box
          double height;                    // Height of a box
};

int main()
{
          Box  Box1;                        // Declare Box1 of type Box
          Box  Box2;                        // Declare Box2 of type Box
          double volume =0.0;               // Store the volume of a box here

          // box 1 specification
          Box1.height =5.0;
          Box1.length =6.0;
          Box1.breadth =7.0;

          // box 2 specification
          Box2.height =10.0;
          Box2.length =12.0;
          Box2.breadth =13.0;

          // volume of box 1
          volume=Box1.height *Box1.length *Box1.breadth;
          cout<<"Volume of Box1 : "<< volume <<endl;

          // volume of box 2
          volume=Box2.height *Box2.length *Box2.breadth;
          cout<<"Volume of Box2 : "<< volume <<endl;
```

```
    return0;
}
```

When the above code is compiled and executed, it produces the following result:
Volume of Box1 : 210
Volume of Box2 : 1560

It is important to note that private and protected members cannot be accessed directly using direct member access operator (.)

## Inline function

- The inline functions are a C++ enhancement feature to increase the execution time of a program.
- Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.
- C++ provides an inline functions to reduce the function call overhead.
- Inline function is a function that is expanded in line when it is called.
- When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.
- This substitution is performed by the C++ compiler at compile time.
- Inline function may increase efficiency if it is small.

**Syntax:**

```
inline return-type function-name(arguments)
{
        // function code
}
```

- Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining.
- C++ **inline** function is powerful concept that is commonly used with classes.
- If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
- Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.
- To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function.
- A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.
- Following is an example, which makes use of inline function to return max of two numbers :-

```
inline int Max(int x,int y)
{
        return (x > y) ? x : y;
```

```
}

int main()
{
        cout<<"Max (20,10): "<<Max(20,10)<<endl;
        cout<<"Max (0,200): "<<Max(0,200)<<endl;
        cout<<"Max (100,1010): "<<Max(100,1010)<<endl;
        return0;
}
```

When the above code is compiled and executed, it produces the following result –

Max (20, 10): 20
Max (0,200): 200
Max (100, 1010): 1010

**Compiler may not perform inlining in such circumstances like:**

- If a function contains a loop. (for, while, do-while)
- If a function contains static variables.
- If a function is recursive.
- If a function contains switch or goto statement.

**Inline functions provide following advantages:**

- Function call overhead doesn't occur.
- It also saves the overhead of push/pop variables on the stack when function is called.
- It also saves overhead of a return call from a function.

When you create inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal functioncalls.



## Default Arguments

- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.

Following is a simple C++ example to demonstrate the use of default arguments. We don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

**Example:**

```
#include<iostream.h>
// A function with default arguments, it can be called with 2 or 3  or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
        return (x + y + z + w);
}
int main()
{
        cout<< sum(10, 15) <<endl;
        cout<< sum(10, 15, 25) <<endl;
        cout<< sum(10, 15, 25, 30) <<endl;
        return 0;
}
```

**Output:**

```
25
50
80
```

**Important Points about Default Arguments:**

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
- During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.
- Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable z is not default.

```
int sum(int x=10, int y, int z=30,int w);
```

**Some examples of function declaration with default values are:**

```
int Add(int x, int y, int z=30);          //Valid
int Add(int x, int y=20, int z=30);       //Valid
int Add(int x=10, int y=20, int z=30);    //Valid

int Add(int x=10, int y, int z);          //Invalid
int Add(int x=10, int y, int z=30);       //Invalid
```

## Static Members of a Class

- It is a variable which is declared with the static keyword, it is also known as class member, and thus only single copy of the variable creates for all objects.
- Any changes in the static data member through one member function will reflect in all other object's member functions.
- We can define class member's static using static keyword.
- When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- A static member is shared by all objects of the class.
- All static data is initialized to zero when the first object is created, if no other initialization is present.
- We can't put it in the class definition but it can be initialized outside the class as done in the following example by re-declaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

**Example:**

```
#include<iostream>
class Box
{
   public:
        static int objectCount;
        // Constructor definition
        Box(double l =2.0,double b =2.0,double h =2.0)
        {
             cout<<"Constructor called."<<endl;
             length= l;
             breadth= b;
             height= h;

             // Increase every time object is created
             objectCount++;
        }
        double Volume()
        {
             return(length * breadth * height);
        }

        private:
             double length;          // Length of a box
             double breadth;         // Breadth of a box
             double height;          // Height of a box
```

```
};

// Initialize static member of class Box
int Box :: objectCount=0;

int main(void)
{
        Box  Box1(3.3,1.2,1.5);          // Declare box1
        Box  Box2(8.5,6.0,2.0);          // Declare box2

        // Print total number of objects.
        cout<<"Total objects: "<<Box::objectCount<<endl;

        return0;
}
```

**Output:**
Constructor called.
Constructor called.
Total objects: 2

**Note:** we will discuss concept of constructor in Unit-2.

## Static Member Functions

- By declaring a member function as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator ::.
- A static member function can only access static data member, other static member functions and any other functions from outside the class.
- You could use a static member function to determine whether some objects of the class have been created or not.



## Nested Class

- A **nested class** is a **class** which is declared in another enclosing **class**.
- The nested class is also a member variable of the enclosing class and has the same access rights as the other members.
- The members of an enclosing **class** have no special access to members of a **nested class**; the usual access rules shall be followed.

**Example:**

```
class A
{
        public:
        class B
        {
                private:
                    int num;
                public:
                    voidgetdata(int n)
                    {
                            num= n;
                    }
                    voidputdata()
                    {
                            cout<<"The number is "<<num;
                    }
        };
};
int main()
{
        cout<<"Nested classes in C++"<<endl;
        A :: B obj;
        obj.getdata(9);
        obj.putdata();
        return0;
}
```

**Output:**

```
Nested classes in C++
The number is 9
```

In the above program, class B is defined inside the class A so it is a nested class. The class B contains a private variable num and two public functions getdata() and putdata(). The function getdata() takes the data and the function putdata() displays the data.

## Local Class

A class declared inside a function becomes local to that function and is called Local Class.

**Ex:-**

```
void fun()
```

```
    {
        class Test                    // local to function 'fun'.
        {
            /* members of Test class */
        };
    }
int main()
{
    return 0;
}
```



## Scope resolution operator (::)

- The **scope resolution operator**( :: ) is used for several reasons.
- For **example**: If the global variable name is same as local variable name, the **scope resolution operator** will be used to call the global variable.
- It is also used to define a function outside the class.
- It is used to access the static variables of class.
- It will be used in the case of multiple inheritance.

**Ex:- To access a global variable when there is a local variable with same name:**

```
int x;                    // Global x

int main()
{
    int x = 10; // Local x
    cout<< "Value of global x is "<< ::x;
    cout<< "\nValue of local x is "<< x;
    return 0;
}
```



**Output:**
Value of global x is 0
Value of local x is 10

**Ex:-To define a function outside a class.**

```
classA
{
    public:
    // Only declaration
    void fun();
};
```

```
// Definition outside class using '::'
void A :: fun()
{
  cout<< "fun() called";
}
 int main()
 {
    A a;
    a.fun();
    return 0;
}
```

**Output:**

fun() called

**Ex:-To access a class's static variables.**

```
class Test
{
   static int x;
      public:
          static int y;

   void func(int x)
   {
        // We can access class's static variable
        // even if there is a local variable

        cout<< "Value of static x is "<< Test::x;
        cout<< "\nValue of local x is "<< x;
   }
};
// In C++, static members must be explicitly defined
// like this

int Test::x = 1;
int Test::y = 2;


int main()
{
   Test obj;
   int x = 3 ;
   obj.func(x);

   cout<< "\nTest::y = "<< Test::y;
```

```
    return0;
}
```

**Output:**

Value of static x is 1
Value of local x is 3
Test::y = 2;

**Ex:-In case of multiple Inheritance:**

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

```
class A
{
        protected:
          int x;
          public:
            A()
            {
                x = 10;
            }
};
class B
{
        protected:
          int x;
        public:
          B()
          {
              x = 20;
          }
};
class C: public A, public B
{
  public:
    void fun()
    {
        cout<< "A's x is "<< A::x;
        cout<< "\nB's x is "<< B::x;
    }
};

int main()
{
  C c;
```

```
    c.fun();
    return0;
}
```

**Output:**
A's x is 10
B's x is 20

**Note:** we will discuss concept of Inheritance in Unit-2.



## Call by value and Call by reference

### Call by Value:

- The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
- In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter(modify) the arguments used to call the function.

**Ex:-**

```
#include<iostream.h>
// function declaration
void swap(int x, int y);

int main ()
{
        // local variable declaration:
        int a =10;
        int b =20;

        cout<<"Before swap, value of a :"<< a <<endl;
        cout<<"Before swap, value of b :"<< b <<endl;

        // calling a function to swap the values.
        swap(a, b);

        cout<<"After swap, value of a :"<< a <<endl;
        cout<<"After swap, value of b :"<< b <<endl;

        return0;
}
```

```
// function definition to swap the values.
void swap(int  x, int  y)
{
        int temp;
        temp= x;                /* save the value of x */
        x = y;                  /* put y into x */
        y = temp;               /* put x into y */
        return;
}
```



**Output:**

Before swap, value of a :10
Before swap, value of b :20
After swap, value of a :10
After swap, value of b :20

**Call by reference:**

- The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter.
- Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.
- To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

**Ex:-**

```
#include<iostream>

// function declaration
void swap(int &x,int &y);
int main ()
{
        // local variable declaration:
        int a =10;
        int b =20;

        cout<<"Before swap, value of a :"<< a <<endl;
        cout<<"Before swap, value of b :"<< b <<endl;

        /* calling a function to swap the values using variable reference.*/
        swap(a, b);

        cout<<"After swap, value of a :"<< a <<endl;
```

```
        cout<<"After swap, value of b :"<< b <<endl;

        return0;
}
// function definition to swap the values.
void swap(int &x, int &y)
{
        int temp;
        temp= x;            /* save the value at address x */
        x = y;              /* put y into x */
        y = temp;           /* put x into y */
}
```

**Output:**

Before swap, value of a :10
Before swap, value of b :20
After swap, value of a :20
After swap, value of b :10

### Passing an Object as function argument

- In C++ we can pass object of the class as arguments and also return them from a function the same way we pass and return other variables.
- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

**Syntax:**

```
function_name(object_name);
```

**Ex:-**

```
#include<iostream>
class A
{
    public:
        int n=100;
        char ch='A';

        void disp(A a)
        {
           cout<<a.n<<endl;
           cout<<a.ch<<endl;
        }
};
int main()
```

```
{
    A obj;
    obj.disp(obj);
    return 0;
}
```

**Output:**
100
A



Here in class A we have a function disp() in which we are passing the object of class A. Similarly we can pass the object of another class to a function of different class.

## Returning Object as argument

**Syntax:**

```
object = return object_name;
```

In the following example we have two functions, the function **input()** returns the Student object and function **disp()** takes Student object as an argument.

```
#include<iostream>
class Student
{
    public:
      int stuId;
      int stuAge;

      /* In this function we are returning the Student object.*/

      Student input(int n, int a)
      {
            Student obj;
            obj.stuId= n;
            obj.stuAge= a;
            return obj;
      }
      /* In this function we are passing object as an argument.*/

      void disp(Student obj)
      {
            cout<<"Student Id =  "<<obj.stuId<<endl;
            cout<<"Student Age =  "<<obj.stuAge<<endl;
      }
```

```
};

int main()
{
        Student s;
        s = s.Input(101,18);
        s.disp(s);
        return 0;
}
```

**Output:**

```
Student Id = 101
Student Age = 18
```

## Arrays of Objects

- An **object** of class represents a single record in memory, if we want more than one record of class type, we have to create an **array** of class or **object**.
- As we know, an **array** is a collection of similar type, therefore an **array** can be a collection of class type.

**Syntax:**

```
class_name  array_name [size] ;
```

**Ex:-**

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

class Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;

    public:
      void GetData()
      {
            cout<<"\n\tEnter Employee Id : ";
            cin>>Id;
            cout<<"\n\tEnter Employee Name : ";
            cin>>Name;
```

```
                cout<<"\n\tEnter Employee Age : ";
                cin>>Age;
                cout<<"\n\tEnter Employee Salary : ";
                cin>>Salary;
        }
        void PutData()
        {
          cout<<"\n"<<Id<<"\t"<<Name<<"\t"<<Age<<"\t"<<Salary;
        }
    };
void main()
{
        int i;
        Employee E[3];
        for(i=0;i<3;i++)
        {
                cout<<"\nEnter details of "<<i+1<<" Employee";
                E[i].GetData();
        }
      cout<<"\nDetails of Employees";
      for(i=0;i<3;i++)
        {
              E[i].PutData();
        }
}
```

**Output :**

```
Enter details of 1 Employee
Enter Employee Id : 101
Enter Employee Name : Harsh
Enter Employee Age : 29
Enter Employee Salary : 45000

Enter details of 2 Employee
Enter Employee Id : 102
Enter Employee Name : Manoj
Enter Employee Age : 31
Enter Employee Salary : 51000

Enter details of 3 Employee
Enter Employee Id : 103
Enter Employee Name : Viral
Enter Employee Age : 28
Enter Employee Salary : 47000

Details of Employees
```

| | | | |
|---|---|---|---|
| 101 | Harsh | 29 | 45000 |
| 102 | Manoj | 31 | 51000 |
| 103 | Viral | 28 | 47000 |

In the above example, we are getting and displaying the data of 3 employee using array of object.

## 'this' pointer in C++

- To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

- Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
- Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
- The compiler supplies an implicit pointer along with the names of the functions as 'this'.
- The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.
- 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

**Ex:-**

```
#include<iostream>
 /* local variable is same as a member's name */
classTest
{
   private:
      int x;
   public:
      void setX (int x)
      {
            // The 'this' pointer is used to retrieve the object's x
            // hidden by the local variable 'x'
         this->x = x;
      }
      void print()
      {
           cout<< "The value of x = "<< x <<endl;
      }
};
int main()
{
```

```
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}
```

**Output:**

The value of x = 20



## References

- A reference variable is an alias, that is, another name for an already existing variable.
- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- You cannot have NULL references. A reference must be initialized when it is created.

### Creating References in C++

- You can access the contents of the variable through either the original variable name or the reference.
- For **Example:**

```
int i = 17;
```

We can declare reference variables for i as follows.

```
Int &r = i;
```

- Read the **&** in these declarations as **reference**.
- Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration in following example as "s is a double reference initialized to d"

**Ex:-**

```
#include <iostream>
int main ()
{
    // declare simple variables
    int i;
    double d;

   // declare reference variables
   int &r = i;
   double &s = d;

   i = 5;
```

```
    cout<< "Value of i : " <<i<<endl;
    cout<< "Value of i reference : " << r  <<endl;

    d = 11.7;
    cout<< "Value of d : " << d <<endl;
    cout<< "Value of d reference : " << s  <<endl;
    return 0;
}
```

**Output:**

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

- C++ supports passing references as function parameter more safely than parameters.
- You can return reference from a C++ function like any other data type.

## Dynamic Memory Allocation Operators

There are two types of memory allocation.

1) Static memory allocation -- allocated by the compiler. Exact size and type of memory must be known at compile time.
2) Dynamic memory allocation -- memory allocated during run time.

To dynamically allocate memory in C++, we use the **new** operator and To de-allocate dynamic memory, we use the **delete** operator.

**new:**

- Dynamic memory allocation means creating memory at runtime. For example, when we declare an array, we must provide size of array in our source code to allocate memory at compile time.
- But if we need to allocate memory at runtime we must use new operator followed by data type.
- If we need to allocate memory for more than one element, we must provide total number of elements required in square bracket[ ]. It will return the address of first byte of memory.

**Syntax of new operator:-**

```
ptr = new data-type;
                    //allocte memory for one element
```

```
ptr = new data-type [ size ];
                                    //allocte memory for fixed number of element
```

**delete:**

- Delete operator is used to deallocate the memory created by new operator at run-time.
- Once the memory is no longer needed it should by freed so that the memory becomes available again for other request of dynamic memory.

**Syntax of delete operator:-**

```
delete ptr;
                                    //deallocte memory for one element

delete[] ptr;
                                    //deallocte memory for array
```

**Example:**

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

void main()
{
   int size,i;
   int *ptr;

   cout<<"\n\tEnter size of Array : ";
   cin>>size;

   ptr = new int[size];
            //Creating memory at run-time and return first byte of address to ptr.

   for(i=0;i<5;i++)                 //Input arrray from user.
   {
         cout<<"\nEnter any number : ";
         cin>>ptr[i];
   }

   for(i=0;i<5;i++)           //Output arrray to console.
   {
      cout<<ptr[i]<<", ";
```

```
    }

    delete[] ptr;
                    //deallocating all the memory created by new operator
}
```

**Output :**

```
Enter size of Array : 5
Enter any number : 78
Enter any number : 45
Enter any number : 12
Enter any number : 89
Enter any number : 56

78, 45, 12, 89, 56,
```

## EXERCISE

**Short Questions:**

1. Define class and object.
2. When and how inline function can be used?
3. How scope resolution operator used in different ways in object oriented programming?
4. Which method cannot call through an object?
5. What are the advantages of cin and cout compared to printf and scanf?
6. Explain the limitation of inline function.
7. What is the application of scope resolution operator (::) in C++?
8. List out the advantages of inline function.
9. Discuss input and output operators.
10. Explain insertion and extraction operators. Give example.
11. What do you mean by cascading of operators? Explain with example.
12. What is the use of scope resolution operator?
13. When to make function as inline?

14. Define local class. Give example.
15. What do you mean by nested class?
16. What is the use of 'this' pointer?
17. What is containership?
18. What is the use of 'new' operator?
19. What is reference variable?
20. What is the advantage of new over malloc()?
21. Differentiate between int *p and int **p.

**Long Questions:**

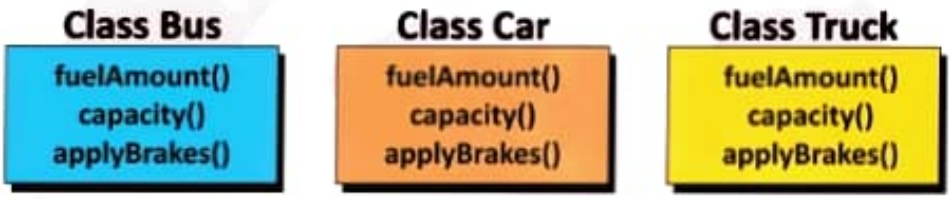
1. What is header file? Explain various header file used in C++.
2. Explain different string handling functions with appropriate example.
3. Write a note on C++ data types.
4. State the differences between C and C++.
5. What is truly Object Oriented Programming? Can we say C++ is truly Object Oriented Programming? Justify your answer.
6. What is object oriented programming? How it is differ from procedure oriented programming?
7. What do you mean by default argument? When it is useful?
8. What is Object Oriented Programming? Write the difference between OOP and POP.
9. Compare structure and class.
10. Compare array and class.
11. Explain any three concepts of object oriented programming in detail.
12. Explain default argument and function prototyping with an example.
13. Compare call by value and call by reference with appropriate example.
14. Explain object as function argument and returning object.
15. Explain array of objects with example.
16. Explain the benefits of object oriented programming.
17. Explain static data member and static function.
18. Explain memory management operator. Discuss new over malloc.

19. List out memory management operators. Point out reasons why Using new is better idea than using malloc() ?
20. Write a C++ program to consider Rollno, Name and Marks in C++ as data members. Take suitable member functions which provide functionality of input and display data. Class should capable to keep information for 5 students.

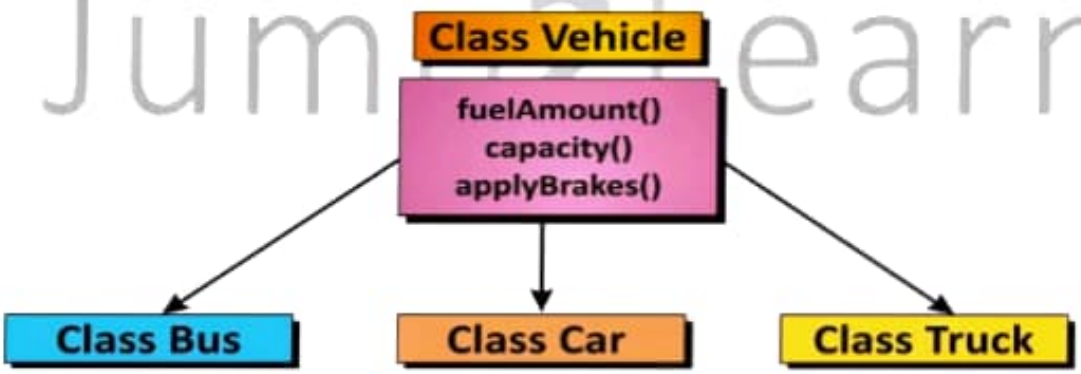
# Unit - 2
# Data Encapsulation and Inheritance

## INHERITANCE

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**.
- Inheritance is one of the most important features of Object Oriented Programming.
- It enable the reusability concept for real life object.
- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class. It is also known as child class.
- **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class. It is also known as parent class.
- Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes.
- If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:




- You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy and also new updatation
- To avoid this type of situation, inheritance is used.
- If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.
- Look at the below diagram in which the three classes are inherited from vehicle class:




- Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

## IMPLEMENTING INHERITANCE

For creating a sub-class which is inherited from the base class we have to follow the below syntax.

**Syntax**

```
class  subclass_name : access_mode  base_class_name
{
        //body of subclass
};
```

- Here, **subclass_name**is the name of the sub class, **access_mode**is the mode in which you want to inherit this sub class (for example: public, private or protected) and **base_class_name**is the name of the base class from which you want to inherit the sub class.
- If the access_mode is not used, then it is private by default.

**Example:**

```
class Parent
{
  public:
        int id_p;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
  public:
        int id_c;
};
//main function
int main()
 {
        Child obj1;
        // An object of class child has all data members
        // and member functions of class parent
        obj1.id_c = 7;
        obj1.id_p = 91;
        cout<< "Child id is "<<  obj1.id_c <<endl;
        cout<< "Parent id is "<<  obj1.id_p <<endl;
       return 0;
 }
```

**Output:**

```
Child id is 7
Parent id is 91
```

- In the above program the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

## Modes of Inheritance (Visibility Modifier)

**Public mode**:

If we derive a sub class from a public base class then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

**Protected mode**:

If we derive a sub class from a protected base class then both public member and protected members of the base class will become protected in derived class.

**Private mode**:

If we derive a sub class from a Private base class then both public member and protected members of the base class will become Private in derived class.

**Note:-**

The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example.

**Example:**

```
class A
{
        public:
           int x;
        protected:
           int y;
        private:
           int z;
};
class B : public A
{
           // x is public
           // y is protected
           // z is not accessible from B
};
class C : protected A
{
           // x is protected
           // y is protected
           // z is not accessible from C
};
```

```
class D : private A    // 'private' is default for classes
{
        // x is private
        // y is private
        // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

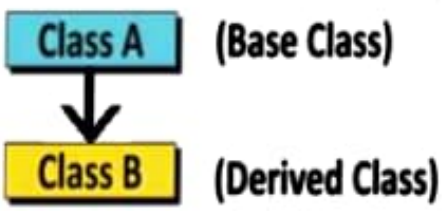
| BASE CLASS MEMBER ACCESS SPECIFIER | TYPE OF INHERITENCE | | |
|---|---|---|---|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Not Accessible (Hidden) | Not Accessible (Hidden) | Not Accessible (Hidden) |

**TABLE 2.1 ACCESS SPECIFIER IN INHERITANCE**

## TYPES OF INHERITANCE

### 1. SINGLE INHERITANCE

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.




**Syntax**

```
class subclass_name : access_mode base_class
{
    //body of subclass
};
```

**Example**

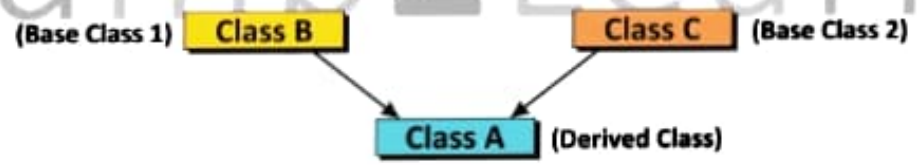
```
class Vehicle
{
    public:
        Vehicle()
        {
            cout<< "This is a Vehicle"<<endl;
        }
};
// sub class derived from base class
class Car: public Vehicle
{

};
// main function
int main()
{
                                    // creating object of sub class will
                                    // invoke the constructor of base classes
        Car obj;
        return 0;
}
```

**Output:**

This is a vehicle

## 2. MULTIPLE INHERITANCES:

Multiple Inheritances is a feature of C++ where a class can inherit from more than one classes. I.e one **sub class** is inherited from more than one **base classes.**




**Syntax:**

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
    //body of subclass
};
```

Here, the number of base classes will be separated by a comma (', ') and access mode for every base class must be specified.

**Example**

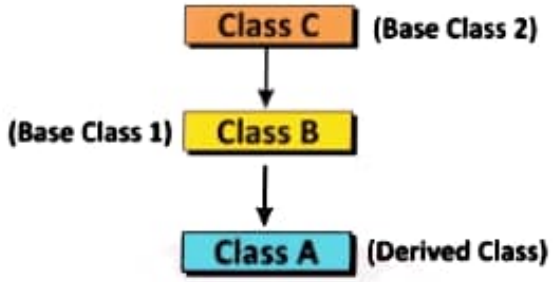
```
// first base class
class Vehicle
{
    public:
    Vehicle()
    {
       cout<< "This is a Vehicle"<<endl;
    }
};

// second base class
class FourWheeler
{
    public:
    FourWheeler()
    {
       cout<< "This is a 4 wheeler Vehicle"<<endl;
    }
};

    // sub class derived from two base classes
class Car: public Vehicle, public FourWheeler
{
};

    // main function
int main()
{
  // creating object of sub class will invoke the constructor of base classes
  Car obj;

  return 0;
}
```

**Output:**

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

### 3. Multilevel Inheritance:

In this type of inheritance, a derived class is inherited from another derived class.




**Example:**

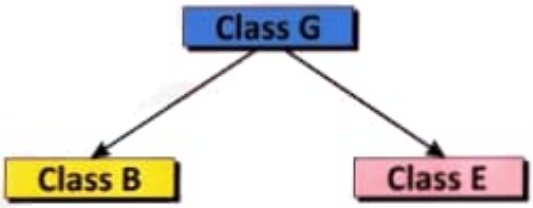
```
#include <iostream>
 // base class
class Vehicle
{
   public:
          Vehicle()
          {
              cout<< "This is a Vehicle"<<endl;
          }
};
class fourWheeler : public Vehicle
{
   public:
          fourWheeler()
          {
               cout<<"Objects with 4 wheels are vehicles"<<endl;
          }
};
class Car : public fourWheeler
{
   public:
          car()
          {
              cout<<"Car has 4 Wheels"<<endl;
          }
};
int main()
{                          //creating object of sub class will
                           //invoke the constructor of base classes
        Car obj;
        return 0;
}
```

**Output:**

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

### 4. Hierarchical Inheritance

In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.




**Example:**

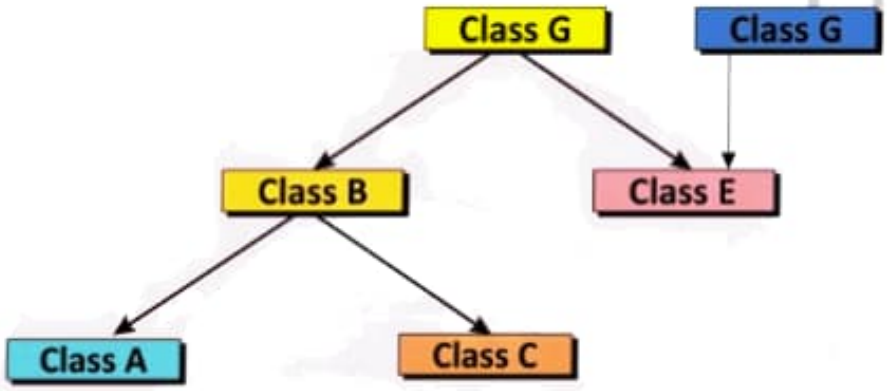
```
#include <iostream>
// base class
class Vehicle
{
   public:
          Vehicle()
          {
               cout<< "This is a Vehicle"<<endl;
          }
};
 // first sub class
class Car : public Vehicle
{
};
// second sub class
class Bus : public Vehicle
{
};
int main()
{
          // creating object of sub class will invoke the constructor of base class
          Car obj1;
          Bus obj2;
          return0;
}
```

**Output:**

This is a Vehicle
This is a Vehicle

## 5. Hybrid Inheritance

- Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical Inheritance and Multiple Inheritance.
- Below image shows the combination of hierarchical and multiple inheritance:




**Example:**

```
#include <iostream>

// base class
class Vehicle
{
    public:
    Vehicle()
    {
        cout<< "This is a Vehicle"<<endl;
    }
};

//base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};
```

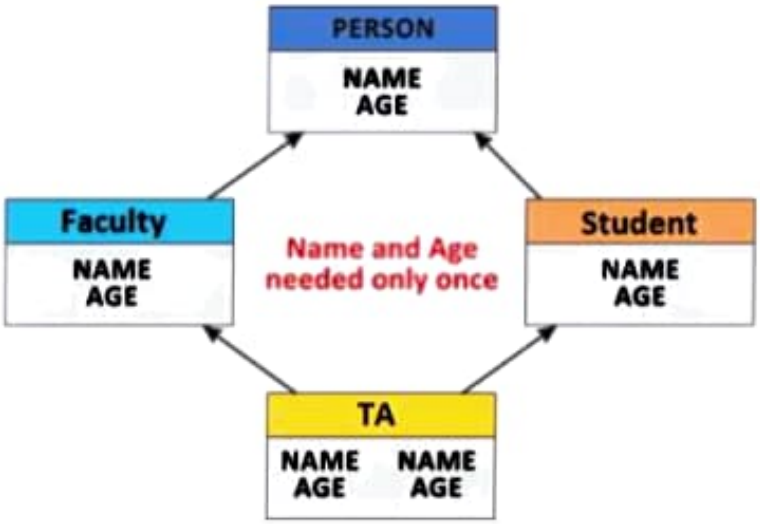
```
// first sub class
class Car : public Vehicle
{
};

// second sub class
class Bus : public Vehicle, public Fare
{
 };

// main function
int main()
{
                    // creating object of sub class will
                        // invoke the constructor of base class
   Bus obj2;
   return 0;
}
```

**Output:**

```
This is a Vehicle
Fare of Vehicle
```

## THE DIAMOND PROBLEM

The diamond problem occurs when two super classes of a class have a common base class. For example, in the following diagram, the **TA** class gets **two copies** of all attributes (NAME, AGE) of Person class, this causes ambiguities.



**Example:**

```
#include<iostream.h>
class Person
{
                    // Data members of person
        public:
                Person(int x)
                {
                    cout<< "Person::Person(int) called"<<endl;
                }
};

class Faculty : public Person
{
                    // data members of Faculty
        public:
                Faculty(int x) : Person(x)
                {
                    cout<<"Faculty::Faculty(int ) called"<<endl;
                }
};

class Student : public Person
{
                    // data members of Student
        public:
                Student(int x) : Person(x)
                {
                    cout<<"Student::Student(int) called"<<endl;
                }
};

class TA : public Faculty, public Student
{
        public:
                TA(int x) : Student(x), Faculty(x)
                {
                    cout<<"TA::TA(int ) called"<<endl;
                }
};

int main()
{
        TA ta1(30);
}
```

**Output:**

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

- In the above program, constructor of 'Person' is called two times.
- Destructor of 'Person' will also be called two times when object 'ta1' is destructed.
- So object 'ta1' has two copies of all members of 'Person', this causes ambiguities.
- The solution to this problem is 'virtual' keyword. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class.

**SOLUTION USING VIRTUAL CLASS:**

```
#include<iostream.h>
class Person
{
        public:
          Person(int x)
          {
                cout<< "Person::Person(int ) called"<<endl;
          }
          Person()
          {
                cout<< "Person::Person() called"<<endl;
          }
};
class Faculty : virtual public Person
{
        public:
                Faculty(int x) : Person(x)
                {
                        cout<<"Faculty::Faculty(int ) called"<<endl;
                }
};
class Student : virtual public Person
{
        public:
                Student(int x) : Person(x)
                        {
                            cout<<"Student::Student(int ) called"<<endl;
                        }
};
```

```
Class TA : public Faculty, public Student
{
        public:
                TA(int x) : Student(x), Faculty(x)
                {
                        cout<<"TA::TA(int ) called"<<endl;
                }
};

int main()
{
          TA  ta1(30);
}
```

**Output:**

```
Person::Person() called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

- In the above program, constructor of 'Person' is called once. One important thing to note in the above output is, the default constructor of 'Person' is called.
- When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

## Constructor

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.
- A constructor will have exact same name as the class and it does not have any return type at all, not even void.
- Constructors can be very useful for setting initial values for certain data member (member variables).
- Each time an object (instance) of a class is created the constructor method is called.
- Constructor is a special member function of class and it is used to initialize the objects of its class.
- It is treated as a special member function because its name is the same as the class name.
- These constructors get invoked whenever an object of its associated class is created.
- It's named as "constructor" because it constructs the value of data member of a class.
- Initial values can be passed as arguments to the constructor function when the object is declared.

**Syntax:**

```
class class_name
{
        int g, h;
        public:
        class_name();              // Constructor Declared
        ...
};
class_name :: class_name()
{
        g=1; h=2;                  // Constructor defined
}
```



**Special characteristics of Constructors:**

- They should be declared in the public section
- They do not have any return type, not even void
- They get automatically invoked when the objects are created
- They cannot be inherited though derived class can call the base class constructor
- Like other functions, they can have default arguments
- You cannot refer to their address
- Constructors cannot be virtual

**Types of Constructors:**

C++ offers four types of constructors:

1. Do nothing constructor
2. Default constructor
3. Parameterized constructor
4. Copy constructor

**Do nothing Constructor:**



- Do nothing constructors are that type of constructor which does not contain any statements. Do nothing constructor is the one which has no argument in it and no return type.

**Default Constructor:**

- The default constructor is the constructor which doesn't take any argument. It has no parameter but a programmer can write some initialization statement there.

**Example:**

```
#include<iostream.h>
class Calc
{
        int val;
        public:
        Calc()
        {
            val = 20;
        }
};

int main()
{
        Calc c1;
        cout<< c1.val;
}
```

- A default constructor is very important for initializing object members, that even if we do not define a constructor explicitly, the compiler automatically provides a default constructor implicitly.

**Parameterized Constructor:**

- A default constructor does not have any parameter, but programmers can add and use parameters within a constructor if required.
- This helps programmers to assign initial values to an object at the time of creation.

**Example:**

```
#include<iostream.h>
class Calc
{
    int val2;
    public:
    Calc(int x)
    {
        val2 = x;
    }
};

int main()
{
   Calc c1(10);
   Calc c2(20);
   Calc c3(30);
```

```
        cout<< c1.val2;
        cout<< c2.val2;
        cout<< c3.val2;
    }
```

**Copy Constructor:**

- C++ provides a special type of constructor which takes an object as an argument and is used to copy values of data members of one object into another object.
- In this case, copy constructors are used to declaring and initializing an object from another object.

**Example:**

```
Calc C2(C1);
    Or
Calc C2 = C1;
```

- The process of initializing through a copy constructor is called the *copy initialization*.

**Syntax:**

```
class-name (class-name &)
{
        . . .
}
```

**Example:**

```
#include <iostream>
class CopyCon
{
   int a, b;
   public:
   CopyCon(int x, int y)
   {
      a = x;
      b = y;
     cout<< "\nHere is the initialization of Constructor";
   }
   void Display()
   {
     cout<< "\nValues : \t" << a << "\t" << b;
   }
};
void main()
{
   CopyCon Object(30, 40);
         //Copy Constructor
   CopyCon Object2 = Object;
  Object.Display();
  Object2.Display();
}
```

## Destructors

- As the name implies, destructors are used to destroy the objects that have been created by the constructor within the C++ program.
- Destructor names are same as the class name but they are preceded by a tilde (~).
- It is a good practice to declare the destructor after the end of using constructor.
- Here's the basic declaration procedure of a destructor:

```
~Cube()
{

}
```



- The destructor neither takes an argument nor returns any value and the compiler implicitly invokes upon the exit from the program for cleaning up storage that is no longer accessible.
- Destructors in C++ are member functions in a class that delete an object.
- They are called when the class object goes out of scope such as when the function ends, the program ends, a delete variable is called etc.
- Destructors are different from normal member functions as they don't take any argument and don't return anything. Also, destructors have the same name as their class and their name is preceded by a tilde(~).

**Example:**

```
#include<iostream.h>
class Demo
{
        private:
        int num1, num2;
        public:
        Demo(int n1,int n2)
        {
           cout<<"Inside Constructor"<<endl;
           num1 = n1;
           num2 = n2;
        }
        void display()
        {
           cout<<"num1 = "<< num1 <<endl;
           cout<<"num2 = "<< num2 <<endl;
        }
        ~Demo()
        {
           cout<<"Inside Destructor";
        }
};
```

```
int main()
{
    Demo obj1(10,20);
    obj1.display();
    return 0;
}
```

**Output:**

```
Inside Constructor
num1 = 10
num2 = 20
Inside Destructor
```

- In the above program, the class Demo contains a parameterized constructor that initializes num1 and num2 with the values provided by n1 and n2.
- It also contains a function display() that prints the value of num1 and num2.
- There is also a destructor in Demo that is called when the scope of the class object is ended.

## CONSTRUCTOR/ DESTRUCTOR CALL IN INHERITANCE

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.
- If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invokation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.

## WHY THE BASE CLASS'S CONSTRUCTOR IS CALLED ON CREATING AN OBJECT OF DERIVED CLASS?

- When a class is inherited from other class, The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only.
- So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only.
- This is why the constructor of **base class is called first to initialize all the inherited members.**

**Example**

```
#include <iostream.h>
 // base class
class Parent
{
        public:
             // base class constructor
            Parent()
            {
              cout<< "Inside base class"<<endl;
            }
};

 // sub class
class Child : public Parent
{
        public:
            //sub class constructor
            Child()
            {
              cout<< "Inside sub class"<<endl;
            }
};

int main()
{
         // creating object of sub class
        Child obj;
        return 0;
}
```



**Output:**

```
Inside base class
Inside sub class
```



**ORDER OF CONSTRUCTOR CALL FOR MULTIPLE INHERITANCES:**

For multiple inheritances order of constructor call is, the base class's constructors are called in the order of inheritance and then the derived class's constructor.

**Example:**

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

```
// first base class
class Parent1
{
        public:
            // first base class's Constructor
             Parent1()
             {
                cout<< "Inside first base class"<<endl;
             }
};
 // second base class
class Parent2
{
        public:
            // second base class's Constructor
             Parent2()
             {
                cout<< "Inside second base class"<<endl;
             }
};
 // child class inherits Parent1 and Parent2
class Child : public Parent1, public Parent2
{
        public:
             // child class's Constructor
             Child()
             {
                cout<< "Inside child class"<<endl;
             }
};
 // main function
int main()
{
         // creating object of class Child
        Child obj1;
        return 0;
}
```

**Output:**

```
Inside first base class
Inside second base class
Inside child class
```

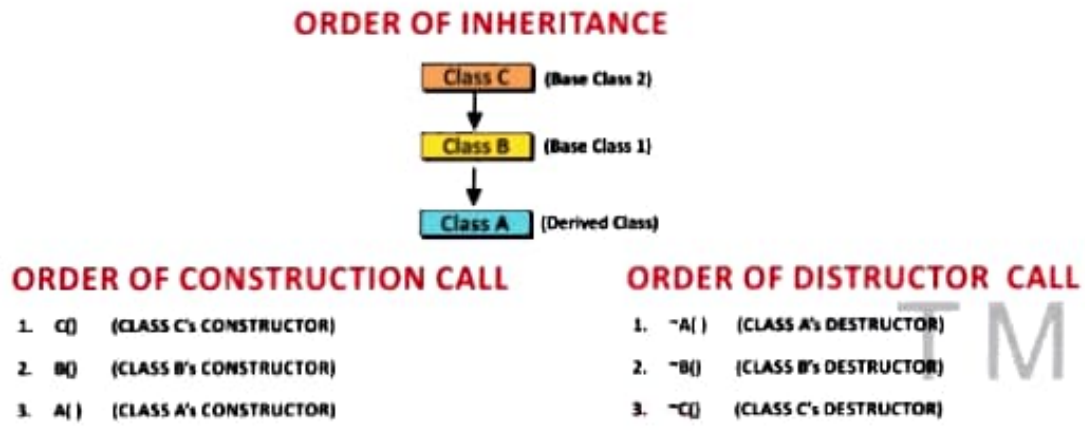



**FIGURE 2.1 ORDER OF CONSTRUCTOR AND DESTRUCTOR CALL FOR A GIVEN ORDER OF INHERITANCE**

## Containership

Whenever an object of a class is declared as a member of another class it is known as a container class. In the containership the object of one class is declared in another class.

We can create an object of one class into another and that object will be a member of the class. This type of relationship between classes is known as containership or has-a relationship as one class contain the object of another class. And the class which contains the object and members of another class in this kind of relationship is called a container class.

The object that is part of another object is called contained object, whereas object that contains another object as its part or attribute is called container object.

## Difference between containership and inheritance:

### Containership

When features of existing class are wanted inside your new class, but, not its interface for eg.

1) Computer system has a hard disk
2) Car has an Engine, chassis, steering wheels.

### Inheritance

When you want to force the new type to be the same type as the base class.
for eg.

1) Computer system is an electronic device
2) Car is a vehicle

**Example:**

```
#include<iostream.h>
#include<conio.h>
class A
    {
        int a;
        public:
        void get();
    };
class B
    {
        int b;
        A t;                    // object 't' of class 'A' is declare in class 'B'
        public:
        void getdata();
    };
void A :: get()
{
        cin>>a;
        cout<<a;
}
void B :: getdata()
{
        cin>>b;
        cout<<b;
        t.get();                //calling of get() of class A in getdata() of class B
}
void main()
{
        clrscr();
        B ab;
        ab.getdata();
        getch();
}
```

# EXERCISE

**Short Questions:**

1. What is the purpose of abstract class?
2. What is the use of 'protected' over 'private' modifier?
3. What is the use of destructor? When it is called?
4. Explain data hiding concept in classes.
5. Constructor can't have a return type. True/False.
6. What is dynamic constructor?
7. Differentiate between copy and parameterized constructor.
8. What is visibility modifier?
9. What is the use of destructor? How is it created?
10. What is Constructor with default argument?
11. How the constructor is called in multilevel inheritance?
12. What is the use of 'protected' modifier?
13. What is object? Give one example of it.
14. What is abstract class? What is use of it?

**Long Questions:**

1. What is destructor? How it is written in C++? When it is called? Differentiate between constructor and destructor with proper example.
2. Explain constructor with example.
3. Explain the execution of base class constructors for multilevel and multiple Inheritance.
4. What is constructor? How do we call a constructor?
5. What is visibility modifier? List them. Differentiate with proper example.
6. Differentiate between multilevel and multiple Inheritance.
7. How to access the member if inheritance is private?
8. What is inheritance? Explain multiple and multilevel inheritance.
9. What is the purpose of inheritance? Explain multilevel inheritance with example.
10. What is constructor? Explain all types of constructor.

11. Explain visibility modifier with example.
12. What is inheritance? Explain different types of inheritance in brief.
13. Explain copy constructor with proper example.
14. How to remove ambiguity occurred in the case of hybrid inheritance?
15. What is destructor? How it is written in C++? When it is called? Differentiate between constructor and destructor with proper example.
16. What is constructor? Explain parameterized constructor with an example.
17. What do you mean by constructors in derived classes? If constructor function in derived and base class, then which constructor function get executed explain with an example?
18. Explain data abstraction and encapsulation.
19. Explain Hybrid Inheritance and how we can remove the ambiguity occurred while implementing hybrid inheritance.
20. Write a program to create a class student stores the roll_no, class test stores the mark obtained in two subjects and class result contains the total marks obtained the test. The class result can inherit the details of the marks obtained in the test and the roll_no of student class.
21. Write a program for creating a class figure & calculate the area of the circle (pi*r*r) with the help of constructor. Get the details from the user & display it
22. Create an abstract class "shape" which stores data members like length, breadth and radius. Create two classes Circle" and "Rectangle" which stores data members like area respectively. Write a function to calculate area and display it.

Jump2Learn™
PUBLICATION

www.jump2learn.com

# CONCEPTS of OBJECT ORIENTED PROGRAMMING with DATA STRUCTURE

Jump2Learn - The Online Learning Place

Dr. Yatin K. Solanki | Ms. Nikisha R. Chalakwala | Mr. Chetan N. Rathod

# Unit - 3

# Polymorphism

TM

## WHAT IS POLYMORPHISM?

- The word poly means many and morphism means form i.e. many forms.
- In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.
- Polymorphism is considered as one of the important features of Object Oriented Programming.

## IN C++ POLYMORPHISM IS MAINLY DIVIDED INTO TWO TYPES:

- Compile time Polymorphism
- Runtime Polymorphism

### 1. COMPILE TIME POLYMORPHISM:

This type of polymorphism is achieved by function overloading or operator overloading.

### FUNCTION OVERLOADING:

- Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.
- C++ allows you to specify more than one definition for a **function** name in the same scope, which is called **function overloading.**
- An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
- When you call an overloaded **function**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function with the parameter types specified in the definitions.
- The process of selecting the most appropriate overloaded function is called **overload resolution**.
- When there are multiple functions with same name but different parameters then these functions are said to be **overloaded.**
- Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments.**
- You cannot overload function declarations that differ only by return type.

**Example:**

```
#include<iostream.h>
class printData
{
    public:
        void print(int i)
        {
            cout<<"Printing int: "<< i <<endl;
        }
        void print(float  f)
        {
            cout<<"Printing float: "<< f <<endl;
        }
        void print(char c)
        {
            cout<<"Printing character: "<< c <<endl;
        }
};
int main()
{
        printData pd;
                                // Call print to print integer
        pd.print(5);
                                // Call print to print float
        pd.print(12.35);
                                // Call print to print character
        pd.print('Y');

        return0;
}
```

**Output:**

```
Printing int: 5
Printing float: 12.35
Printing character: Y
```



## CONSTRUCTOR OVERLOADING

- In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.
- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.

- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

**Example:**

```
#include <iostream>
class construct
{
        public:
          float area;
                                // Constructor with no parameters
          construct()
          {
              area = 0;
          }
                                // Constructor with two parameters
          construct(int a, int b)
          {
              area = a * b;
          }

          void disp()
          {
              cout<< area<<endl;
          }
};
int main()
{
          // Constructor Overloading with two different constructors
          // of class name
      construct o;
      construct o2( 10, 20);

      o.disp();
      o2.disp();

      return 1;
}
```

**Output:**

```
0
200
```

## OPERATORS OVERLOADING

- C++ tries to make the user defined data types behave in much the same way as the built-in types.
- For example, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. (we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +)
- 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.
- You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.
- Almost any operator can be overloaded in C++. However there are few operators which cannot be overloaded.
- **Operator that are not overloaded** are follows:
    - Scope resolution operator (::)
    - Class member access operator (. (dot) and .* (dot-asterisk))
    - Conditional Operator (?:)
    - Size Operator (sizeof)

### FOLLOWING ARE SOME RESTRICTIONS TO BE KEPT IN MIND WHILE IMPLEMENTING OPERATOR OVERLOADING:

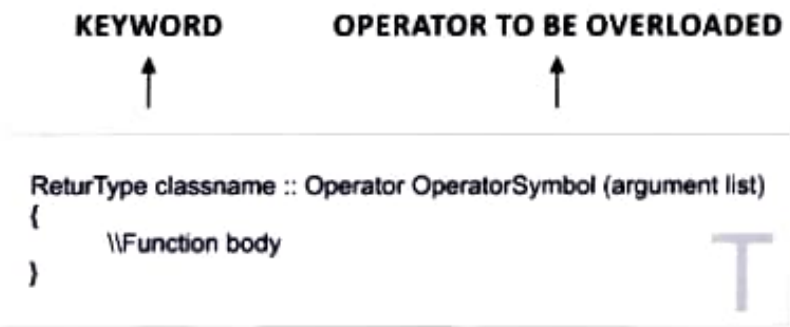
1. Precedence and Associativity of an operator cannot be changed.
2. Numbers of Operands cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of operator. When an operator is overloaded, it's original meaning is not lost.



### 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 special function, called **operator function**, which describes the task.

**Syntax**




- Where **ReturnType** is the type of value returned by the specified operation and **OperatorSymbol** is the operator being overloaded. **Operator OperatorSymbol** is the function name, where **Operator** is keyword.
- Operator function must be either member functions or friend functions.
- A basic difference between them is that a friend function will have only one argument for unary operators and two arguments for binary operators, while a member function has no arguments for unary operators and only one for binary operators.
- This is because the object use to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions.
- Arguments may be passed either by value or by reference.

**PROCESS OF OVERLOADING INVOLVES THE FOLLOWING STEPS:**

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operation function **operator operatorsymbol ()** in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.

**Example:**

**Overload minus ' - ' operator (overload unary minus operator)**

```
#include<iostream.h>
class space
{
    int a;
    int b;
    int c;
    public:
      void getdata(int x, int y, int z);
      void display();
      void operator-();
};
```

```
void space :: getdata(int x, int y, int z)
{
    a = x;
    b = y;
    c = z;
}
void space :: display()
{
    cout<<"A: "<< a <<" B:"<< b <<"C: "<<c<<endl;
}
void space  ::  operator-()
{
        a = -a;
        b = -b;
        c = -c;
}
int main()
{
        space S;
        S.getdata(10, -20, 30);
        cout<< " S  :  ";
        S.display();

        -S;
        cout<< " -S  :  ";
        S.display();
        return0;
}
```

**Overload pre/post increment (++) operator**

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

class point
{
  int x;
  int y;

  public:
    void getdata(int x1, int y1)
     {
        x = x1;
        y = y1;
     }
```

```
    void display()
    {
        cout<< "Point (" << x << "," <<y<<")\n";
    }
    point operator++()
    {
       x++;
       y++;
       return *this;

    }
    point operator++(int x)
    {
        point p=*this;
        ++(*this);
        return p;

    }
};
int main()
{
  point p1,p2;
  clrscr();
  p1.getdata(10,12);
  p1.display();
  ++p1;
  p1.display();
  p2=p1++;
  cout<<"P2=";
  p2.display();
  cout<<"p1=";
  p1.display();
  getch();
  return 0;
}
```



**Overload Binary + Operator to Concate two strings**

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

class string
{
  char s[100];
```

```
    public:
      string()
      {
            strcpy(s,"");
      }
      string(char s1[])
      {
            strcpy(s,s1);
      }
      void display()
      {
            cout<<s<<endl;
      }
      string operator +(string str)
      {
          string temp;
          strcpy(temp.s,s);
          strcat(temp.s,str.s);
          return temp;
      }
};
void main()
{
  string fstr="BCA";
  string sstr="college";
  string tstr;
  fstr.display();
  sstr.display();
  tstr = fstr + sstr;
  cout<<"Concatenation :";
  tstr.display();
}
```



## Type Conversion

- Constants and variables of different types are mixed in an expression; C++ applies automatic type conversion to the operands as per certain rules.
- Assignment operator do the automatic type conversion, the type of data to the right of an assignment operator is automatically converted to the type of the variable on the left.

**There are two types of casting**

## 1. Implicit Type Conversion

- **Needs:** C++ permits mixing of constant and variables of different types in an expression.
- C++ automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significant.
- This automatic conversion is known as implicit type conversion.
- If the operands are of different types, the lower type is automatically converted to the higher type before the operation proceeds.
- **Rules:**
  1. If one of the operands is long double, the other will be converted to long double and the result will be long double.
  2. Else If one of the operand is double, the other will be converted into double and the result will be double.
  3. Else, if one of the operand is float, the other will be converted into float and the result will be float.
  4. Else, if one of the operand is unsigned long int, the other will be converted into unsigned long int and the result will be unsigned long int.
  5. Else, If one operand is long int and the other is unsigned int then unsigned int can be converted into long int and result will be long int.
- float to int causes truncation of the fractional part.
- double to float causes rounding of digit
- long int to int causes dropping of excess higher order bits.

## 2. Explicit Type Conversion:

- It is used when we want to force a type conversion in a way that is different from the automatic conversion.
- E.g.
```
int a=5,c=9;
float b;
b= a/c;
```
- Here a and c both are integers in the line, the decimal part of the result of the division would be lost and the b would represent wrong value.
- E.g. `b= (float)a/c;`
- The operator (float) converts the 'a' to floating point for the purpose of evaluation of the expression. Then using rules of automatic conversion, the division is performed in floating point mode.

**What happens when they are user-defined data types?**

- Consider the following statement that adds two objects and then assigns the result to a third object.

```
v3 = v1 + v2;          // v1, v2 and v3 are class type objects
```

- When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable?

**Three types of situations might arise in the data conversion between incompatible types:**

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

**Basic to Class Type:**

- It is achieved by creating parameterized constructor in class.

**For example:**
Convert char type array into String Object Type.

```
class String
{
   char *s;
   Public :
          String (char *a)
          {
              strcpy(s,a);
          }
};
void main()
{
       String s1,s2;
       char *n1="TMT"
       char *n2="BCA"
       S1=String(n1);
       S2=n2;
}
```

Here, S1= string(n1), first converts n1 from char* type to string type and then Assigns the string type values to the object s1.
S2 = n2; also does the same job by invoking the constructor implicitly.

**Class to Basic Type:**

- It is achieved by overload casting operator.

  General Syntax for overload casting (Type)

**Synatx:**

```
Operator  Typename()
{
    .....
    ..... (Function Statement)
    .....
}
```

- This function converts a class type data to typename. For example, the operator double() converts a class object to type double, the operator int converts a class type object to type int, and so on.

**Example:**

```
Vector :: operator double()
{
    double sum =0;
    for (int i=0; i<size;i++)
       sum = sum +  v[i] * u[i]
    return sum;
}

double s = double(v1);
    Or
double s1 = v1;
```

- Where v1 is an object of type vector. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type.

**Casting Operator function should satisfy the following Conditions:**

1. It must be a class member.
2. It must not specify a return type.
3. It must not have any arguments.

- Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

**Example:**
**string to char* as follows:**

```
string :: operator char*()
{
    return (p)
}
```

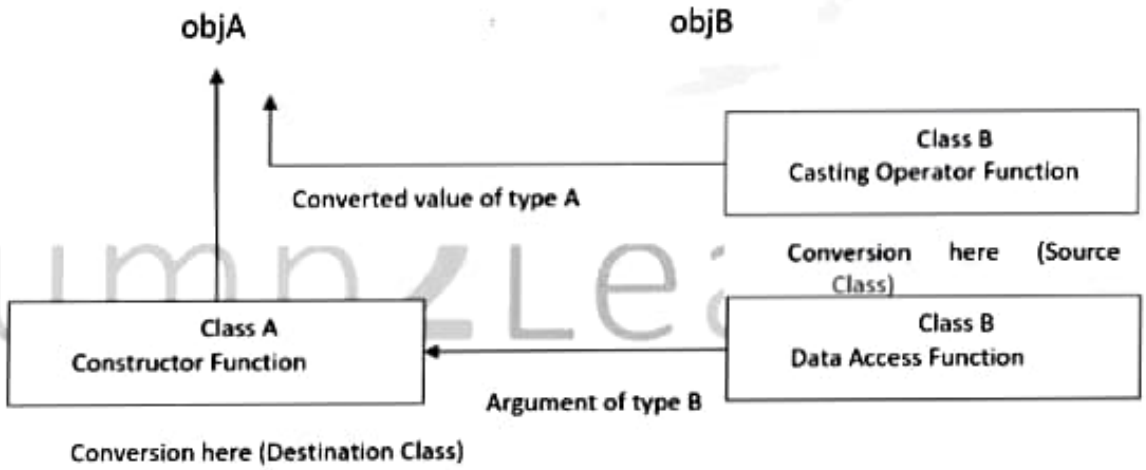


**One class to another class type**

- We would like to convert one class type data to another class type.

Example:
**Ob1 = ob2;**

- Ob1 is an object of class A and ob2 is an object of class B. The class B type data is converted to the class A type data and the converted value is assigned to the ob1. Since the conversion takes place from Class B to class A, B is known as the source class and A is known as the destination class.






- Such conversions between objects of different classes can be carried out by either a constructor or a conversion function.

| Conversion required | Conversion takes place in | |
|---|---|---|
| | Source class | Destination class |
| Basic → Class | Not applicable | Constructor |
| Class → Basic | Casting Operator | Not Applicable |
| Class → Class | Casting Operator | Constructor |

- When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

## Friend function

- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.
- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

### Declaration of friend function:

```
class class_name
{
   friend data_type function_name(argument/s);      // syntax of friend function.
};
```

- In the above declaration, the friend function is preceded by the keyword friend.
- The function can be defined anywhere in the program like a normal C++ function.
- The function definition does not use either the keyword friend or scope resolution operator.

### Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

**Example when the function is friendly to two classes:**

```
#include <iostream.h>
class B;                                  // forward declarartion.
class A
{
   int x;
   public:
   void setdata(int i)
   {
      x = i;
   }
   friend void min(A,B);                  // friend function.
};
class B
{
   int y;
   public:
   void setdata(int i)
   {
      y = i;
   }
   friend void min(A,B);                  // friend function
};
void min(A a,B b)
{
     if(a.x<=b.y)
          cout << a.x ;
     else
          cout << b.y ;
}
int main()
{
  A a;
  B b;

  a.setdata(10);
  b.setdata(20);
  min(a,b);
  return 0;
}
```

**Output:**

10

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

**Example of Operator Overloading using Friend function (input/output oprator)**

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#define SIZE 10

class vector
{
  int a[SIZE];
  public:
  vector()
  {
    for (int i=0;i<SIZE;i++)
              a[i]=0;

  }
  vector(int *x)
  {
    for (int i=0;i<SIZE;i++)
              a[i]=x[i];
     }

 friend istream & operator >> (istream &vin, vector &x)
 {
    for(int i=0;i<SIZE;i++)
              vin>>x.a[i];
    return (vin);
 }
 friend ostream & operator <<(ostream &vout, vector &x)
 {
    for(int i=0;i<SIZE;i++)
              vout<<x.a[i]<<endl;
    return (vout);
 }
};
void main()
{
  vector v;
  cout<<"\n Enter elements of vector ";
  cin>>v;
  cout<<"\n Display vector elements";
  cout<<v;
  getch();
}
```

## Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

**Example of a friend class:**

```
#include <iostream>
class A
{
   int x =5;
   friend class B;                    // friend class.
};
class B
{
 public:
   void display(A &a)
   {
      cout<<"value of x is : "<<a.x;
   }
};
int main()
{
   A a;
   B b;
   b.display(a);
   return 0;
}
```

**Output:**

value of x is : 5

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.



**OVERLOADING OPERATOR USING A FRIEND FUNCTION:**

- In this approach, the operator overloading function must proceed with friend keyword, and declare a function in class scope.
- Keeping in mind, friend operator function takes two parameters in a binary operator and one parameter in a unary operator.
- Friend function using operator overloading offers better flexibility to the class.
- These functions are not a members of the class and they do not have 'this' pointer.
- When you overload a unary operator you have to pass one argument.
- Friend function can access private members of a class directly.

**Syntax:**

```
friend return-type operator operator_symbol (argument_list)
{
    //Statements;
}
```

**Example:**

**Program to overload Unary minus (-) operator using Friend function.**

```
#include<iostream>
class space
{

    int a;
    int b;
    int c;
    public:
        voidgetdata(int x, int y, int z);
        void display();
        void friend operator-(space &p);          //Pass by reference
};
void space :: getdata(int x, int y, int z)
{
   a = x;
   b = y;
   c = z;
}
voidspace :: display()
{
    cout<<"A: "<< a <<" B:"<< b <<"C: "<<c<<endl;
}

void operator-(space &p)
{
   p.a = -p.a;                //Object name must be used as it is a friend function
   p.b = -p.b;
   p.c = -p.c;
}
int main()
{
        space S;
        S.getdata(10, -20, 30);

        cout<< " S : ";
```

```
        S.display();

        -S;
        cout<< " -S : ";
        S.display();

        return0;
}
```



## 2. RUNTIME POLYMORPHISM:

This type of polymorphism is achieved by Function Overriding.

**FUNCTION OVERRIDING:**

- If derived class defines same function as defined in its base class, it is known as function overriding in C++.
- It enables you to provide specific implementation of the function which is already provided by its base class.
- That base function is said to be overridden.
- Behavior of functions: Overriding is needed when derived class function has to do some added or different job than the base class function.
- It is like creating a new version of an old function, in the child class.
- Both functions must have the same parameters in both classes. Ie both function have same signature.

**Example:**

```
class base
{
    public:
      virtual void print ()
      {
          cout<< "print base class"<<endl;
      }
      void show ()
      {
          cout<< "show base class"<<endl;
      }
};
class derived : public base
{
    public:
      void print ()
      {
          cout<< "print derived class"<<endl;
      }
```

```
        void show ()
        {
            cout<< "show derived class"<<endl;
        }
};

//main function
int main()
{
        base *bptr;
        derived d;
        bptr = &d;
            //virtual function, binded at runtime (Runtime polymorphism)
        bptr->print();
            // Non-virtual function, binded at compile time
        bptr->show();
        return 0;
}
```

**Output:**

print derived class
show base class

**Difference between Overloading vs. Overriding:**

| FUNCTION OVERLOADING | FUNCTION OVERRIDING |
|---|---|
| The scope is the same | The scope is different |
| Signatures must differ (ex: parameter) | Signatures must be same |
| Number of overloading functions possible | Only one overriding function possible |
| May occur without inheritance | It mainly occurs due to inheritance |



## ABSTRACTION

- Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But,

we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.

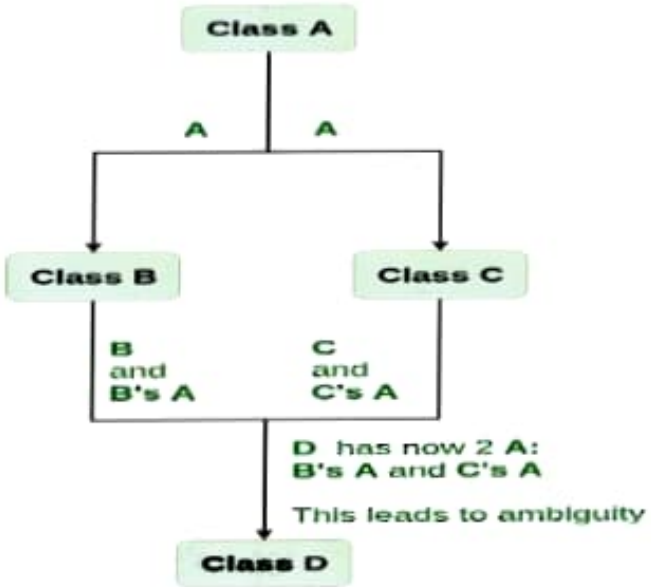
- C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

**ADVANTAGES OF ABSTRACTION:**

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.
- The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

## Virtual base class

- Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.
- Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these classes are inherited into another new class **D** as shown in figure below.



- As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**.
- When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**.
- This confuses compiler and it displays error.

**Example:**

```
#include <iostream>
classA
{
public:
   voidshow()
   {
      cout << "Hello form A \n";
   }
};
classB : publicA
{
};
classC : publicA
{
};
classD : publicB, publicC
{
};

int main()
{
   D object;
   object.show();
}
```





**Compile Errors:-**

```
prog.cpp: In function 'int main()':
error: request for member 'show' is ambiguous
 object.show();
      ^
note: candidates are: void A::show()
 void show()
      ^
note: void A::show()
```

- To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as below:

**Syntax for Virtual Base Classes:**

**Syntax 1:**

```
class B : virtual public A
{
};
```

**Syntax 2:**

```
class C : public virtual A
{
};
```

- **virtual** can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class.
- Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances.
- When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

**Example:**

```
#include <iostream.h>
class A
{
public:
  void show()
  {
    cout << "Hello from A \n";
  }
};
class B : public virtual A
{
};
class C : public virtual A
{
};
classD : public B, public C
{
};
```

```
int main()
{
   D object;
   object.show();
}
```

**Output:**

Hello from A



**Example:**

```
#include <iostream.h>
class A
{
 public:
   int a;
   A()                    // constructor
   {
      a = 10;
   }
};
classB : public virtual A
{
};
class C : public virtual A
{
};
class D : public B, public C
{
};
int main()
{
   D object;                              // object creation of class d
   cout << "a = "<< object.a << endl;
   return 0;
}
```



**Output:**

a = 10

**Here,** The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

## VIRTUAL FUNCTION

- A virtual function is a function in a base class that is declared using the keyword virtual.
- A virtual function is a member function which is declared within a base class and is re-defined (Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called.
- This sort of operation is referred to as dynamic linkage, or late binding.

### RULES FOR VIRTUAL FUNCTIONS:

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

**Example:**

```
#include <iostream.h>

class A
{
  int x=5;
  public:
          void display()
          {
            std::cout << "Value of x is : " << x<<std::endl;
          }
};

class B: public A
{
  int y = 10;
```

```
    public:
        void display()
        {
          std::cout << "Value of y is : " <<y<< std::endl;
        }
};
int main()
{
        A *a;
        B b;
        a = &b;
        a -> display();
        return 0;
}
```

**Output:**

Value of x is : 5

**Example:**

```
#include <iostream.h>
class A
{
    public:
        virtual void display()
        {
            cout << "Base class is invoked"<<endl;
        }
};
class B : public A
{
    public:
        void display()
        {
            cout << "Derived Class is invoked"<<endl;
        }
};
int main()
{
    A *a;                   //pointer of base class
    B b;                    //object of derived class
    a = &b;
    a -> display();         //Late Binding occurs
}
```

**Output:**

Derived Class is invoked

## PURE VIRTUAL FUNCTIONS

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "do-nothing" function.
- The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

**Example:**

```
class Shape
{
   protected:
        int width, height;
   public:
        Shape(int a = 0, int b = 0)
        {
           width = a;
           height = b;
        }
   virtual int area() = 0;                    // pure virtual function
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

**Compile-time (early binding) VS. run-time (late binding) behavior of Virtual Functions:**

Consider the following simple program showing run-time behavior of virtual functions.

**Example:**

```
#include<iostream.h>
class base
{
 public:
  virtual void print ()
  {
       cout<< "print base class"<<endl;
  }
```

```
    void show ()
    {
          cout<< "show base class"<<endl;
    }
};
class derived : public base
{
  public:
    void print ()
    {
        cout<< "print derived class"<<endl;
    }
    void show ()
    {
        cout<< "show derived class"<<endl;
    }
};
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
                                //virtual function, binded at runtime
    bptr->print();

                                // Non-virtual function, binded at compile time
    bptr->show();
}
```

**Output:**

```
print derived class
show base class
```

- Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class.
- In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.
- Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is *print derived class* as pointer is pointing to object of derived class )

and show() is non-virtual so it will be bound during compile time(output is *show base class* as pointer is of base type ).

- If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.



## EXERCISE

**Short Questions:**

1. What is polymorphism?
2. Explain use of late binding.
3. What do you mean by function overloading?
4. What is function overriding?
5. What are the rules for the Binary operator overloading using friend?
6. What is static binding?
7. Explain pure virtual function.
8. Which function in C++ is defined to do nothing? Explain it.
9. List the operators which cannot be overloaded.
10. List out the operators which cannot be overloaded using friend function.
11. What are the rules for Unary Operator Overloading Using Friend?

**Long Questions:**

1. Explain compile time and runtime polymorphism
2. Explain operator overloading with example.
3. What is Polymorphism? Differentiate between Compile time and Runtime Polymorphism.
4. What do you mean by polymorphism? Explain run time polymorphism with an example.
5. What do you mean by overloading of an operator? Why it is necessary to overload an operator?
6. Differentiate between overloading and overriding. Explain the concept of overriding with example.
7. Explain pure virtual function? When it is necessary? Explain rules of pure virtual function.

8. Differentiate between unary and binary operator overloading.
9. How runtime polymorphism is achieved in C++ ? Explain with example.
10. In which circumstances function can be made friend? Write the advantage of friend function.
11. What do you mean by overloading of an operator? How many arguments are needed for overloading unary operator and why?
12. Explain virtual base class.
13. What is operator overloading? Write rules for operator overloading. Also provide proper example.
14. What is polymorphism? Explain different types of polymorphism in brief.
15. What is the use of friend function? Write all rules with example.
16. What is dynamic binding? Explain it with proper example.
17. What is friend function? Why we need to write friend function? Explain with example. Discuss its advantages.
18. Explain virtual base class. How it is differ from virtual function. Explain with proper example.
19. What is operator overloading? Write any seven rules for operator overloading. Explain unary operator overloading with an example.
20. Explain compile time polymorphism with an example.
21. Write a program to overload + and = = operator for string manipulation.
22. Write a program to overload >> and << operators.
23. Write a program to overload the = = and + = operator (String object)
24. Use + operator to concate two string. Use = operator to copy one string to another string.

# Unit - 4

# Data Structure

## Introduction of Data Structure

A data structure is a particular way of organizing data in some fashion in a computer so that later on it can be accessed, queried, or even updated easily and quickly. The idea is to reduce the space and time complexities of different tasks. It is a collection of values. The values can have relationships among them and they can have functions applied to them.

Data structures are mainly of two types:

1. **Linear Data structure:** Here the Data elements are organised in a sequence of some manner.
2. **Non-linear Data structure:** Here the data is ordered in any random order and not in a sequence.

Some common data structures are arrays, Linked List, Stack, Queue, etc.

**Some areas in which data structures are applied extensively:**

Data structures are used in any program or software.

They are used in the areas of,

- Compiler Design
- Operating System
- Database Management System
- Graphics
- Simulation
- Numerical Analysis
- Artificial Intelligence
- In designing algorithms that are highly efficient.
- For managing large internet indexing services and databases with huge information.

## Recursion

Recursion is the concept of well-defined self-reference. In computer science, recursion is a programming technique using function or algorithm that calls itself one or more times until a specified condition is met.

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function 'A' either calls itself directly or calls a function 'B' that in turn calls the original function 'A'. The function 'A' is called recursive function.

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –

- **Base criteria** – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.

- **Progressive approach** – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

### Base condition in recursion:

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
  if (n < = 1)        // base case
    return 1;
  else
    return n*fact(n-1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

### Recursive functions:

A recursive function is one that calls itself.Many mathematical functions can be defined recursively such as factorial and Fibonacci.

**Example: Factorial**
One of the simplest examples of a recursive definition is that for the factorial function:

```
factorial( n ) = if ( n = 0 ) then 1
                 else
                   n * factorial( n-1 )
```

A natural way to calculate factorials is to write a recursive function which matches this definition:

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

Note how this function calls itself to evaluate the next term. Eventually it will reach the termination condition and exit.

The termination condition is obviously extremely important when dealing with recursive functions. If it is omitted, then the function will continue to call itself.

Failure to include a correct termination condition in a recursive function is a steps for disaster.

**Example: Fibonacci**

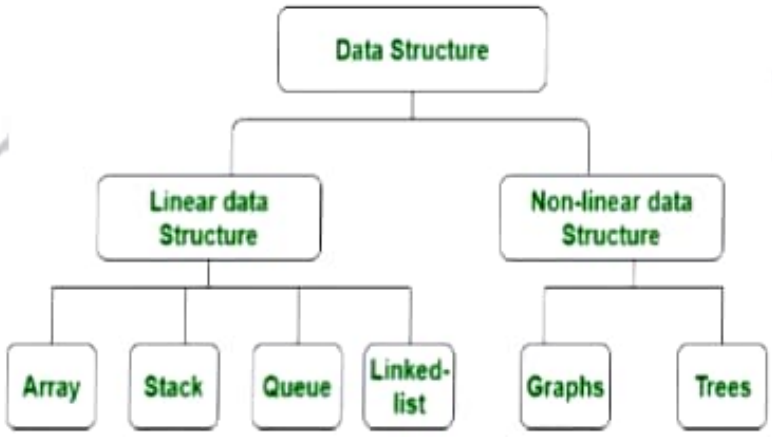
Another commonly used example of a recursive function is thecalculation of Fibonacci numbers.

```
fib( n ) = if ( n = 0 ) then 1
           if ( n = 1 ) then 1
           else
               fib( n-1 ) + fib( n-2 )
```

A usual way to calculate Fibonacci numbers is to write a recursive function which matches this definition:

```
function fib( int n )
{
        if ( (n == 0) || (n == 1) )
                return 1;
        else
                return fib(n-1) + fib(n-2);
}
```

**Linear and Non-Linear Data Structure:**



**Linear Data Structure:**

A linear data structure is a structure in which the elements are stored sequentially and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

The types of linear data structures are Array, Queue, Stack, Linked List.

**Non-Linear Data Structure:**

A non-linear data structure is another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is non-sequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and Graphs are the types of non-linear data structure.

**Following are the important differences between Linear Data Structures and Non-Linear Data Structures:**

| Linear Data Structures | Non-linear Data Structures |
|---|---|
| In a linear data structure, data elements are arranged in a linear order where each and every elementis attached to its previous and next adjacent. | In a non-linear data structure, data elements are attached in hierarchically manner. |
| In a linear data structure, memory is not utilized in an efficient way. | While in a non-linear data structure, memory is utilized in an efficient way. |
| In linear data structure, data elements can be traversed in a single run only. | While in non-linear data structure, data elements can't be traversed in a single run only and needs multiple runs to be traversed completely. |
| In linear data structure, all data elements are present at a single level. | In non-linear data structure, data elements are present at multiple levels. |
| Linear data structures are easier to implement. | Non-linear data structures are difficult to understand and implement as compared |

| Linear Data Structures | Non-linear Data Structures |
|---|---|
| | to linear data structures. |
| Time complexity of linear data structure often increases with increase in size. | Time complexity of non-linear data structure often remains with increase in size. |
| Applications of linear data structures are mainly in application software development. | Applications of non-linear data structures are in Artificial Intelligence and image processing. |
| Examples of Linear data structures are Array, List, Queue, Stack, etc.. | Examples of Non-Linear data structures are Graph, Tree, etc... |

## Stack

Stacks are dynamic data structures that follow the Last In First Out (LIFO) principle. The last item to be inserted into a stack is the first one to be deleted from it.

For example, you have a stack of books on a table. The book at the top of the stack is the first item to be moved if you require a book from that stack.

At any given time, we can only access the top element of a stack.

A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.

## Basic Operations:

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic tasks, a stack is used for the following two primary operations:

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.



Every times, we maintain a pointer to the last pushed element on the stack. As this pointer always represents the top of the stack, it is known as **top**. The **top** pointer provides top value of the stack without actually removing it.

Stacks have restrictions on the insertion and deletion of elements. Elements can be inserted or deleted only from one end of the stack i.e. from the **top**.

The element at the top is called the top element. The operations of inserting and deleting elements are called **PUSH** and **POP** respectively.

When the top element of a stack is deleted, if the stack remains non-empty, then the element just below the previous top element becomes the new top element of the stack.

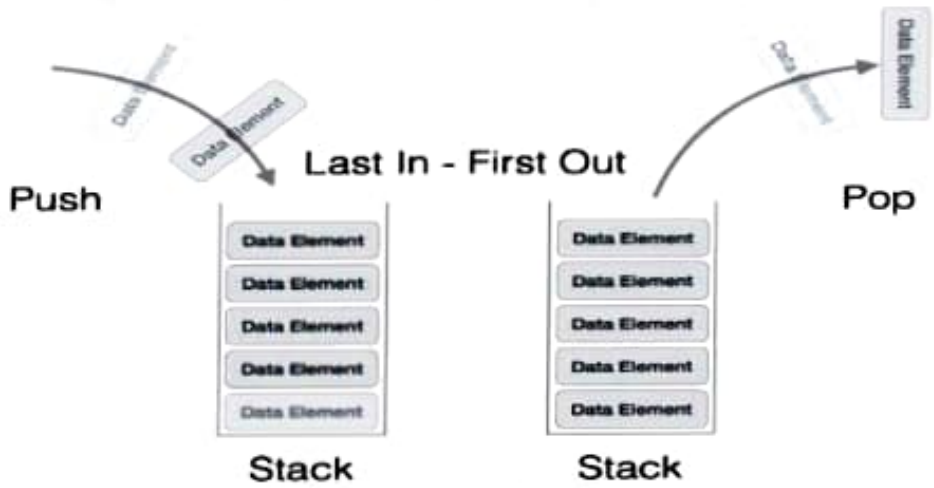
For example, in the stack of books, if you take a book from the top and do not replace it, then the second book automatically becomes the top element (top book) of that stack.

Other important operations on stack are **peep and change (update)** which are used to search element on specific location and replace element in a stack respectively.

- **peep()**– Getting an element of a specific position in stack (From the top)
- **change() or update()** – changing (updating) the value of the specific element from the top.

**Stack Representation:**

The following diagram represents a stack and its operations –




**Stack Implementation:**

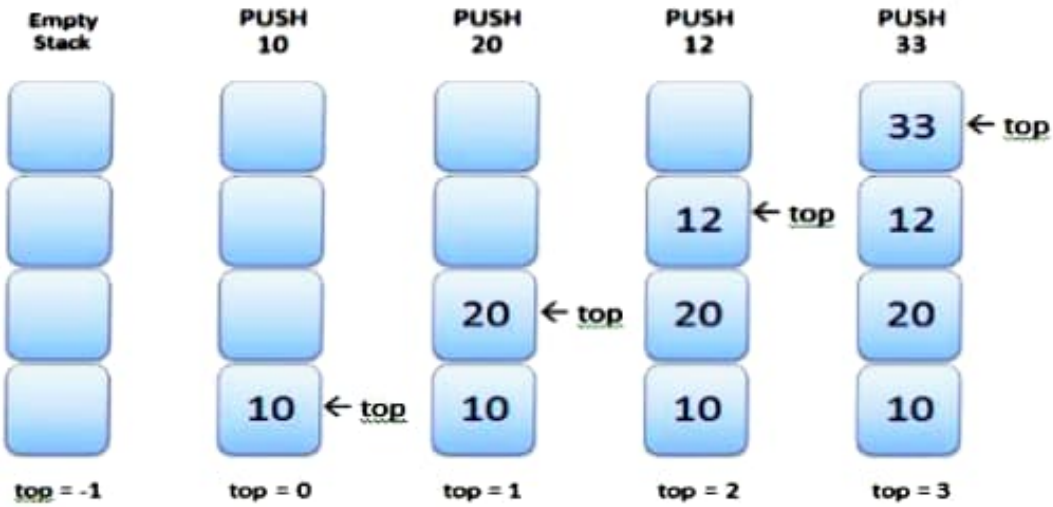
Before implementing actual operations, we have to perform following tasks to create an empty stack.

1. Create a one dimensional array S with fixed size.
   (int S[size])

2. Define an integer variable top and initialize with -1.
   (int top = -1)

**Push Operation:**

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at the top position.




**PUSH operation on Stack with the position of top pointer**
**'top' is initialized to -1. Each time an element is pushed, then top = top+1.**

Push function takes one integer value as parameter and inserts that value into the stack. The process of putting a new data element onto stack is known as a Push Operation. Push operation involves following steps –

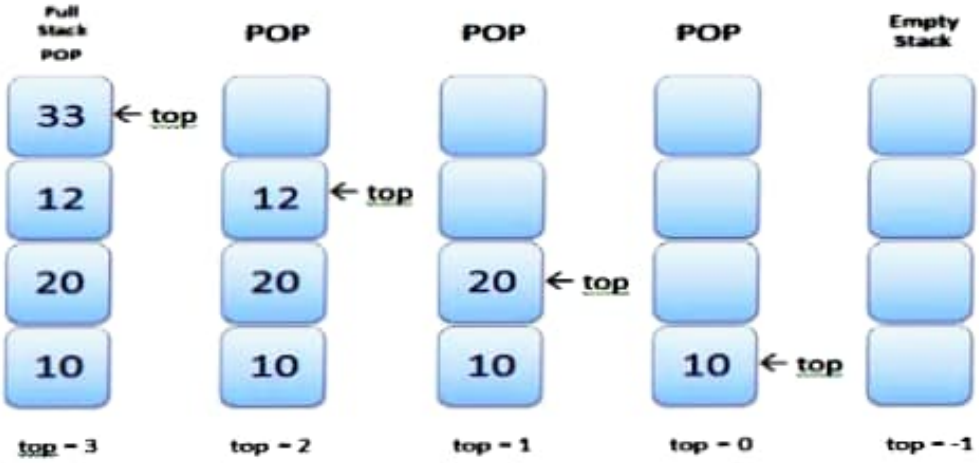
- **Step 1**–[Checks if the stack is overflow (full) or not.]
  [ If the stack is overflow, display an error message and return.]

```
If ( top == size – 1 )

    print "Stack is Overflow"
    return;
```

- **Step 2**–[ If the stack is not full, increments top by 1 to point next empty space. ]
  top = top + 1;
  - Step 3–[Adds data element to the stack location, where top is pointing.]
    S[top] = data;
- **Step 4** – Exit

**Pop Operation:**

Accessing the content while removing it from the stack, is known as a **Pop** Operation. In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position.




POP operation on Stack with the position of top pointer
Each time an element is popped, then top = top - 1.

Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack:

- **Step 1**–[Checks if the stack is underflow (empty) or not.]
  [If the stack is underflow, display an error message and return with 0.]

```
If ( top == – 1 )

    print "Stack is Underflow"
    return 0;
```

- **Step 2**–[If the stack is not underflow Remove the data element from which **top** is pointing.]

```
data = s[top];
```

- **Step 3**–[decrements the value of **top by 1**]

```
top = top - 1;
```

- **Step 4**–[Returns data element.]

```
return data;
```

- **Step 5**– Exit.

## Display Operation:

In a stack, display() function is used to display elements of the stack.We can use the following steps to display the elements of a stack.

- **Step 1** – [Checks if the stack is underflow (empty) or not.]
  [If the stack is underflow, display an error message]
  If ( top == – 1 )

  Print "Stack is Underflow"

- **Step 2** –[If the stack is not underflow then define variable 'i' and initialize it with top]
  int i = top;

- **Step 3** – [Display stack[i] value.]
  print value of Stack[i]

- **Step 4**–[decrements the value of i by 1]
  i= i - 1;

- **Step 5** – Repeat step-3 and step-4 until value of i becomes 0.

- **Step 6**– Exit.

**Peep Operation:**

This operation is used to knowing a data at particular position in stack. A pointer **top** denoting the top element of the stack.

The peep() function returns the value of the $i^{th}$ element from the top of the stack.

Rememberthat the element is not deleted by this function.

A **peep** operation may involve the following steps –

- **Step 1** – [Accepts the position of the element (from the top of the stack) which you want to find]
  Input i

- **Step 2** – [Check for stack underflow]
  if(( top – i + 1)<0)

  print "Stack Underflowon peep"
  return 0

- **Step 3** – [Return ith element from the top of the stack]
  return(s[top – i + 1]);

- **Step 4** –Exit.

## Change or update Operation:

This procedure changes the value of the ith element from the top of the stack to the Valuecontained in data variable. A pointer top denoting the top element of the stack.

A change/update operation may involve the following steps –

- **Step 1** – [Accepts new element for updating existing element]
  Input data
- **Step 2** – [Accepts the position of the element (from the top of the stack) which you want to change]
  Input i
- **Step 3** – [Check for stack underflow]
  if(( top – i + 1)<0)
  print "Stack Underflow on change"
- **Step 4** – [Change ith element from top of the stack]
  s[top – i + 1] = data;
- **Step 5** –Exit.

## Operations of Stack In C++ Using OOP Concepts

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

class stack
{
  int n,top;
  int s[20];
  public:
    stack()
    {
      n=5;
```

```
        top=-1;
    }
    stack(int no)
    {
      n=no;
      top=-1;
    }
   void push (int data)
   {
       if(top==n-1)
       {
            cout<<endl<<" stack overflow";
            return;
       }
      top++;
      s[top]=data;
  }

  int pop()
  {
      int data;
      if(top==-1)
      {
            cout<<endl<<"stack underflow";
            return 0;
      }
     data=s[top];
     top--;
     return data;
  }

 int peep()
 {
     int i,data;
     cout<<endl<<"which element u want :";
     cin>>i;

    if((top-i+1)<0)
    {
          cout<<endl<<"stack underflow on peep";
          return 0;
    }
    data=s[top-i+1];
    return data;
}
```

```
    void change(int data)
    {
        int i;
        cout<<endl<<"which element u want to change:";
        cin>>i;
        if((top-i+1)<0)
        {
                cout<<endl<<"stack underflow on change";
        }
        s[top-i+1]=data;
    }
    void display()
    {
        if(top==-1)
                cout<<"\n stack is empty";
        else
        {
                cout<<"\n the contents of stack is\n";
                for(int i=top;i>=0;i--)
                        cout<<"\n\t"<<s[i];
        }
    }
};
void main()
{
        int e,ch;
        stack s;
        do
        {
                clrscr();
                cout<<"\n\t\t -: STACK IMPLEMENTION :-";
                cout<<"\n\t 1. push";
                cout<<"\n\t 2. pop";
                cout<<"\n\t 3. display";
                cout<<"\n\t 4. peep";
                cout<<"\n\t 5. change";
                cout<<"\n\t 6. exit";
                cout<<"\n\t enter choice:";
                cin>>ch;

                switch (ch)
                {
                case 1:
                        cout<<"\n enter element:";
                        flushall();
```

```
                cin>>e;
                s.push(e);
                break;

        case 2:
                e=s.pop();
                if(e!=0)
                        cout<<"\n the deleted element is "<<e;
                break;

        case 3:
                s.display();
                break;

        case 4:
                e=s.peep();
                if(e!=0)
                        cout<<"\n the selected element is "<<e;
                break;

        case 5:
                cout<<"\n enter new element:";
                flushall();
                cin>>e;
                s.change(e);
            break;

        case 6:
                exit(0);

        default:
                cout<<"\n wrong choice";
    }
    getch();

  } while (ch!=6);
}
```

**STACK IMPLEMENTION WITH STRING**

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

class stack
```

```
{
  int n,top;
  char s[10][20];

  public:
    stack()
    {
      n=5;
      top=-1;
    }

    stack(int no)
    {
      n=no;
      top=-1;
    }

    void push (char data[])
    {
       if(top==n-1)
       {
            cout <<endl<<"stack overflow";
            return;
       }
      top++;
      strcpy(s[top],data);
    }


    char* pop()
    {
      char data [20];
      if(top==-1)
      {
            cout<<endl<<"stack underflow";
            return NULL;
      }

      strcpy(data,s[top]);
      top--;
      return data;
    }
    void display()
    {
      if(top==-1)
```

```
        {
            cout<<endl<<"stack is empty";
        }
        else
        {
            cout<<endl<<"the contents of stack is";
            for(int i=top;i>=0;i--)
                cout<<endl<<"\t"<<s[i];
        }
    }
};              //end of class

void main()
{
    int i,ch;
    char e [20];
    stack s;

    do
    {
      clrscr();
      cout<<endl<<"\t\t stack implemention";
      cout<<endl<<"\t 1. push";
      cout<<endl<<"\t 2. pop";
      cout<<endl<<"\t 3. display";
      cout<<endl<<"\t 4. exit";

      cout<<endl<<"\t enter choice:";
      cin>>ch;
      switch (ch)
      {
      case 1:
            cout<<endl<<" enter string:";
            flushall();
            cin>>e;
            s.push(e);
            break;
      case 2:
            strcpy(e,s.pop());
            if(strcmp(e,NULL)!=0)
            cout<<endl<<"the deleted string is "<<e;
            break;

      case 3:
            s.display();
```

```
                    break;

        case 4:
                    break;

        default:
                    cout<<endl<<"wrong choice";
    }
    getch();

    } while (ch!=4);
}
```



## Expression Parsing

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in an expression.

**Infix Notation:**

We write expression in infix notation, e.g.: a + b, where operators are used in-between operands. It is easy for us, humans to read, write, and speak in infix notation but the same does not appropriate with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

**Prefix Notation:**

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

**Postfix Notation:**

This notation style is known as Reverse Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

**The following table briefly tries to demonstrate the difference in all three notations –**

| Sr.No. | Infix Notation | Prefix Notation | Postfix Notation |
|---|---|---|---|
| 1 | a + b | + a b | a b + |
| 2 | | | |

| | | | |
|---|---|---|---|
| | (a + b) * c | * + a b c | a b + c * |
| 3 | a * (b + c) | * a + b c | a b c + * |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) * (c + d) | * + a b + c d | a b + c d + * |
| 6 | ((a + b) * c) – d | - * + a b c d | a b + c * d - |

To parse any arithmetic expression, we need to take care of operator Precedence and Associativity also.

**Precedence:**

When an operand is in-between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

a + b * c ⇨ a + ( b * c )

As multiplication operation has precedence over addition, b * c will be evaluated first.

**Associativity:**

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b – c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and – are left associative, so the expression will be evaluated as (a + b) – c.

Precedence and Associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –

| Sr. No. | Operator | Precedence | Associativity |
|---|---|---|---|
| 1 | Exponentiation ^ | Highest | Right Associative |

| | | | |
|---|---|---|---|
| 2 | Multiplication ( * )& Division ( / ) | Second Highest | Left Associative |
| 3 | Addition ( + ) & Subtraction ( - ) | Lowest | Left Associative |

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis'( )'.

For example –In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**. so that the expression part enclosing within parenthesis is evaluated first.

## Infix to Postfix Conversion:

**Following steps may be involved to convert infix expression into its equivalent postfix expression:–**

Suppose we have an arithmetic expression written in infix form. Following algorithm finds equivalent postfix expression in the string named "postfix".

**Step 1 –** Scan all the characters of given infix expression from left to right and repeat step 2 to 5 for each character of expression.

**Step 2 –** If an operand is encountered add it to postfix string.

**Step 3 –** If a left parenthesis '(' is encountered push it onto STACK

**Step 4 –** If an operator is encountered then,

1. Repeatedly pop from STACK and add to the postfix stringfor each operator (on the top of STACK) which has same or higher precedence than the arrived operator.
2. Push the new arrived operator to STACK.

**Step 5 –** If a right parenthesis ')' is encountered then,

1. Repeatedly POP from the STACK and add to postfix string. [For each operator on top of stack until a left parenthesis '(' is encountered]
2. Pop the left parenthesis '('.

**Step 6 –** After all characters are scanned, we have to pop all the characters that the stack may have and add it to the Postfix string one by one. Repeat this step as long as stack is not empty.

**Step 7 –**The string "postfix" is the postfix expression of the given infix expression.

**Step 8 –** Exit.

**EXAMPLE: Convert following infix expression into its equivalent postfix expression.**

**A+(B*C-(D/E-F)*G)*H**

| Stack | Input | Postfix |
|---|---|---|
| Empty | A+(B*C-(D/E-F)*G)*H | - |
| Empty | +(B*C-(D/E-F)*G)*H | A |
| + | (B*C-(D/E-F)*G)*H | A |
| +( | B*C-(D/E-F)*G)*H | A |
| +( | *C-(D/E-F)*G)*H | AB |
| +(* | C-(D/E-F)*G)*H | AB |
| +(* | -(D/E-F)*G)*H | ABC |
| +(- | (D/E-F)*G)*H | ABC* |
| +(-( | D/E-F)*G)*H | ABC* |
| +(-( | /E-F)*G)*H | ABC*D |
| +(-(/ | E-F)*G)*H | ABC*D |
| +(-(/ | -F)*G)*H | ABC*DE |
| +(-(- | F)*G)*H | ABC*DE/ |
| +(-(- | F)*G)*H | ABC*DE/ |
| +(-(- | )*G)*H | ABC*DE/F |
| +(- | *G)*H | ABC*DE/F- |
| +(-* | G)*H | ABC*DE/F- |
| +(-* | )*H | ABC*DE/F-G |
| + | *H | ABC*DE/F-G*- |
| +* | H | ABC*DE/F-G*- |
| +* | End | ABC*DE/F-G*-H |
| Empty | End | ABC*DE/F-G*-H*+ |

The postfix expression of given infix expression is: **ABC*DE/F-G*-H*+**

**W.a.p. to convert given infix expression into its equivalent postfix expression using OOP Concept.**

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

class stack
{
  int n,top;
  char s[50], in[50], post[50];

  public:
   stack()
   {
     n=5;
     top=-1;
   }
   stack(int no)
   {
      n=no;
      top=-1;
   }
 void push(char data)
 {
   top++;
   s[top]=data;
 }
 char pop()
 {
   char data;
   data=s[top];
   top--;
   return data;
 }

 char peep()
 {
    char data;
```

```
                cout<<s.pop();
              }
              s.push(in[i]);
            break;

        case '+':
              while(s.peep()=='^' || s.peep()=='/' || s.peep()=='*' || s.peep()=='+')
              {
                cout<<s.pop();
              }
              s.push(in[i]);
            break;

        case '-':
              while(s.peep()=='^' || s.peep()=='/' || s.peep()=='*' || s.peep()=='+' ||
                                                                        s.peep()=='-')
              {
                cout<<s.pop();
              }
              s.push(in[i]);
            break;

        case ')':
              while(s.peep()!='(')
              {
                cout<<s.pop();
              }
              s.pop();
            break;

        default:
              cout<<in[i];
      }
    }
    getch();
}
```



**Infix to Prefix Conversion:**

**Following steps may be involved to convert infix expression into its equivalent prefix expression: -**

Here we need to reverse the input string before conversion and then reverse the final output string before displaying it.

Suppose we have an arithmetic expression written in infix form. Following algorithm finds equivalent prefix expression 'B'.



**Step 1 –** Reverse the infix expression.

**Step 2 –** Push ')' onto STACK, and add '('to end of the new reversed expression.

**Step 3 –** Scan new reversed expression from left to right and repeat step 4 to 7 for each character of the expression.

**Step 4 –** If an operand is encountered add it to B.

**Step 5 –** If a right parenthesis ')' is encountered push it onto STACK

**Step 6 –** If an operator is encountered then,

1. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the arrived operator.
2. Push the new arrived operator to STACK.

**Step 7 –** If left parenthesis '(' is encountered then,



1. Repeatedly POP from the STACK and add to B (each operator on top of stack until a right parenthesis ')' is encountered)
2. Pop the right parenthesis ')'.

**Step 8 –** Reverse the expression B. [i.e. The prefix expression of the given infix expression.]

**Step 9 –** Exit.

**EXAMPLE: Convert following infix expression into its equivalent prefix expression.**

**A+(B*C-(D/E-F)*G)*H**

Reverse form of given infix expression: H*)G*)F-E/D(-C*B(+A

Add left parenthesis ' ( ' at the end : H*)G*)F-E/D(-C*B(+A(

Push right parenthesis ' ) ' into stack.

| Stack | Input | B |
|---|---|---|
| ) | H*)G*)F-E/D(-C*B(+A( | - |
| ) | *)G*)F-E/D(-C*B(+A( | H |
| )* | )G*)F-E/D(-C*B(+A( | H |
| )*) | G*)F-E/D(-C*B(+A( | H |
| )*) | *)F-E/D(-C*B(+A( | HG |
| )*)* | )F-E/D(-C*B(+A( | HG |
| )*)*) | F-E/D(-C*B(+A( | HG |
| )*)*) | -E/D(-C*B(+A( | HGF |
| )*)*)- | E/D(-C*B(+A( | HGF |
| )*)*) - | /D(-C*B(+A( | HGFE |
| )*)*) -/ | D(-C*B(+A( | HGFE |
| )*)*) -/ | (-C*B(+A( | HGFED |
| )*)* | -C*B(+A( | HGFED/- |
| )*) - | C*B(+A( | HGFED/-* |
| )*) - | *B(+A( | HGFED/-*C |
| )*) -* | B(+A( | HGFED/-*C |
| )*) -* | (+A( | HGFED/-*CB |
| )* | +A( | HGFED/-*CB*- |
| )+ | A( | HGFED/-*CB*-* |
| )+ | ( | HGFED/-*CB*-*A |
| Empty | End | HGFED/-*CB*-*A+ |

Now reverse the expression B : +A*-*BC*-/DEFGH

**So the prefix expression of given infix expression is : +A*-*BC*-/DEFGH**

**W.a.p. to convert given infix expression into its equivalent prefix expression using OOP Concept.**

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

class stack
{
   int n,top;
   char s[50];
   public:
    stack()
    {
       n=10;
       top=-1;
    }
    stack(int no)
    {
       n=no;
       top=-1;
    }

    void push(char data)
    {
       top++;
       s[top]=data;
    }

    char pop()
    {
       char data;
       data=s[top];
       top--;
       return data;
    }

  char peep()
  {
       char data;
       data=s[top];
       return data;
  }
};
```

```
void main()
{
        char in[50],pre[50];
        stack s;
        clrscr();

        cout<<endl<<"\t\t CONVERT INFIX EXPRESSION TO PREFIX EXPRESSION "<<endl;
        cout<<endl<<"Enter the infix expression:";
        cin>>in;

        int l,i,j=0;
        s.push('(');

        l=strlen(in);
        strrev(in);

        for(i=0;i<l;i++)
       {
         switch(in[i])
         {
            case ')':
                    s.push(in[i]);
                    break;

            case '^':
                     while(s.peep()=='^')
                           pre[j++]=s.pop();

                     s.push(in[i]);
                    break;

            case '/':
                     while(s.peep()=='^' || s.peep()=='/')
                     {
                           pre[j++]=s.pop();
                     }
                     s.push(in[i]);
                    break;

            case '*':
                     while(s.peep()=='^' || s.peep()=='/' || s.peep()=='*')
                           pre[j++]=s.pop();
                     s.push(in[i]);
                    break;
```

```
        case '+':
                while(s.peep()=='^'||s.peep()=='*'||s.peep()=='/' || s.peep()=='+')
                        pre[j++]=s.pop();
                s.push(in[i]);
                break;

        case '-':
                while(s.peep()=='^'||s.peep()=='*'||s.peep()=='/' || s.peep()=='+' ||
                                                                        s.peep()=='-')
                        pre[j++]=s.pop();
                s.push(in[i]);
                break;

        case '(':
                while(s.peep()!=')')
                        pre[j++]=s.pop();

                s.pop();
                break;

      default:
                pre[j++]=in[i];
    }
  }
  while(s.peep()!='(')
      pre[j++]=s.pop();

  cout<<"The prefix expression :"<<strrev(pre);
  getch();
}
```

## Algorithm for Evaluation of Postfix Expression:

Following **algorithm evaluate the given postfix expression:**

**Step 1 –** Scan the given postfix expression from left to right.

**Step 2 –** If an operand is encountered, put it on STACK.

**Step 3 –** If an operator is encountered, then

1. Remove the two top elements of STACK, where variable **A** is the top element and variable **B** is the next-to-top element.
2. Evaluate B **(operator)** A
3. Place the result in to the STACK.

**Step 4 –** Repeat Step 2 and 3 for each elements of the expression.

**Step 5 –** Result will be the top element on STACK.

**Step 6 –** Exit.

## W.a.p. to evaluate given postfix expression using OOP concept.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>

class stack
{
    int s[10],top;

    public:
    stack()
    {
            top=-1;
    }

    void push(int data)
    {
        top++;
        s[top]=data;
    }

    int pop()
    {
        int data;
        data=s[top];
        top--;
        return data;
    }

    void display()
    {
      cout<<"\n Evaluation of Prefix Expression :"<<s[top];
    }
};
```

```
void main()
{
  char post[50],x[1];
  int l,a,b,rs,i;
  stack s;

  clrscr();

  out<<"\nEnter Postfix Expression :";
  cin>>post;

  l=strlen(post);

  for(i=0;i<l;i++)
  {
    switch(post[i])
    {
    case '^':
            a=s.pop();
            b=s.pop();
            rs=pow(b,a);
            s.push(rs);
        break;

    case '/':
            a=s.pop();
            b=s.pop();
            rs=b/a;
            s.push(rs);
        break;

    case '*':
            a=s.pop();
            b=s.pop();
            rs=b*a;
            s.push(rs);
        break;

    case '+':
            a=s.pop();
            b=s.pop();
            rs=b+a;
            s.push(rs);
        break;
```

```
        case '-':
                a=s.pop();
                b=s.pop();
                rs=b-a;
                s.push(rs);
            break;

        default:
                x[0]=post[i];
                x[1]='\0';
                a=atoi(x);
                s.push(a);
            break;
      }
    }
    s.display();

    getch();
}
```

## Algorithm for Evaluation of Prefix Expression:

**Following algorithm evaluate the given prefix expression:**

**Step 1 –** Reverse the given prefix expression.

**Step 2 –** Scan this reversed expression from left o right.

**Step 3 –** If an operand is encountered, put it on STACK.

**Step 4 –** If an operator is encountered, then

1. Remove the two top elements of STACK, where variable A is the top element and variableB is the next-to-top element.
2. Evaluate A (operator) B
3. Place the result in to the STACK.

**Step 5 –** Repeat Step 3 and 4 for each elements of the expression.

**Step 6 –** Result will be the top element on STACK.

**Step 7 –** Exit.

**W.a.p. to evaluate given prefix expression using OOP concept.**

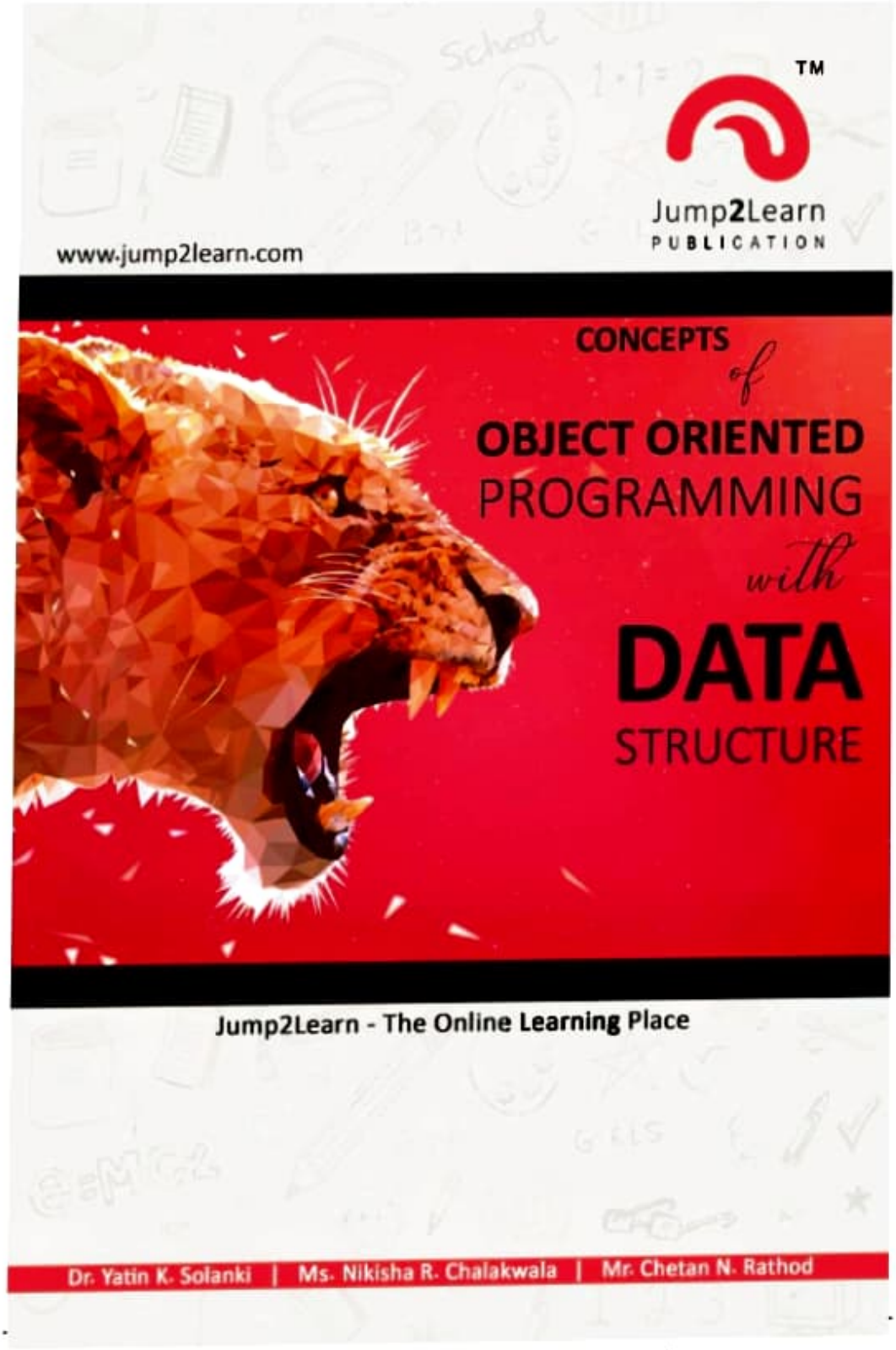
```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
class stack
{
     int s[10],top;
     public:
     stack()
     {
          top=-1;
     }
    void push(int data)
    {
          top++;
          s[top]=data;
    }
    int pop()
    {
         int data;
         data=s[top];
         top--;
         return data;
    }
    void display()
    {
      cout<<"\n Evaluation of Prefix Expression :"<<s[top];
    }
};
void main()
{
  char pre[50],x[1];
  int l,a,b,rs,i;
  stack s;
  clrscr();
  cout<<"\nEnter Prefix Expression :";
  cin>>pre;

  l=strlen(pre);
  strrev(pre);
```

**Short Questions:**

1. Define data structure. List various data structures.
2. Differentiate Linear and nonlinear data structure.
3. What is stack? How it is differ from array?
4. List out applications of stack.
5. What is TOP pointer in stack?
6. Discuss the real world example of stack.
7. What is the initial value of TOP pointer in stack?
8. Convert infix to postfix:- Z + (Y*X— ( W / V ^ U) * T) * S
9. Convert infix to postfix : O + (D * F - (G / V ^ P) * C) * S
10. Evaluate postfix expression: 5 , 6 , 2, + , * , 12 , 4 , / , -
11. Convert Infix to Prefix: A / ( B – C + D ) * E + F ^ G
12. Which condition is not required in dynamic stack?
13. What is Recursion? Define recursive function.
14. What are the necessary conditions to terminate recursive function?

**Long Questions:**

1. What is linear data structure? Discuss difference between FIFO and LIFO concept.
2. What is stack? Write a program to perform PUSH, POP, PEEP and UPDATE operations.
3. What is stack? Write algorithms of various stack operations.
4. Write an algorithm to evaluate prefix expression. Convert the expression P*(Q+R/S)^T into prefix and evaluate this prefix expression using stack tracing with suitable value of P, Q, R, S, T.
5. Explain Tower of Hanoi as application of stack.
6. Explain the concept of stack. Write an algorithm to reverse string using stack.
7. What do you mean by non-linear programming? Discuss difference between non-linear programming and linear programming.
8. Write short note on Polish notation.
9. Write short note on Reverse Polish notation.
10. What is recursion? Write an algorithm to find factorial number.
11. What is recursion? Solve Tower of Hanoi with an example
12. Write an algorithm to convert Infix expression to prefix.
13. Write an algorithm to convert Infix expression to postfix.
14. Convert following expressions into postfix :
    a. ( i ) A + (B*C-D/E*G) + H
    b. ( i i ) (A+B) * (C-D/E)* G+H


TM
Jump2Learn
PUBLICATION
www.jump2learn.com
CONCEPTS of
OBJECT ORIENTED
PROGRAMMING
with
DATA
STRUCTURE
Jump2Learn - The Online Learning Place
Dr. Yatin K. Solanki | Ms. Nikisha R. Chalakwala | Mr. Chetan N. Rathod

# Unit - 5

# Queue

TM

## QUEUE

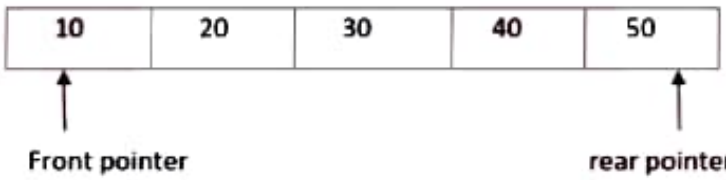
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data and the other end is used to remove data. Queue follows (FIFO) First-In-First-Out methodology, i.e., the data item inserted first will be removed first.The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

A queue is an ordered collection of items where the addition of new items take place at one end, called the "rear," and the removal of existing items occurs at the other end, known as the "front.

Real-world examples can be seen as queues at the ticket windows, where the person who comes first gets the ticket first and another example is Vehicles on toll-tax, where the vehicle that comes first to the toll tax booth leaves the booth first.

### Queue Representation:

As we now understand that in queue, we access both ends for different purpose. Thediagram given below tries to explain queue representation as data structure –




Queues maintain two data pointers, front and rear, initially both the pointers front (f) and rear (r) are set to -1. We have to maintain two data pointers therefore, its operations are comparatively difficult to implement than that of stacks.

### Insert Operation:

In a queue, the new element is always inserted at rear position and existing element is always removed from front position.

Algorithm to perform insert operation on queue –

- **Step 1** – Check, if the queue is full or not.
- **Step 2** – If the queue is full, display overflow error message and return.
- **Step 3** – Otherwise, increment **rear** pointer (r) by 1 to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – if front pointer (f) is at -1, then initialise front pointer to 0.
- **Step 6** – exit

**Delete Operation:**

Algorithm to perform delete operation on queue –

- **Step 1** – Check, if the queue is empty or not.
- **Step 2** – If the queue is empty, display underflow error and return with 0.
- **Step 3** – Remove the data from the queue where **front** is pointing.
- **Step 4** – Check, if rear pointer and front pointer both are equal or not. if yes, then initialise both rear and front pointer to -1.
- **Step 5** – Otherwise Increment **front** pointer by 1 to point to the next available data element.
- **Step 6** – Return Data.
- **Step 7** – Exit.

**Display Operation:**

The display() function is used to display elements of the queue.
We can use the following steps to display the elements of the queue.

- **Step 1 – [Checks if the queue is empty or not.]**
  **[If the queue is empty, display an error message]**

```
If ( f == – 1 )
{
        printf("Queue is Empty");
}
```

- **Step 2 –[If the queue is not empty then define variable 'i' and initialize it with the front Pointer (f) ]**

```
int  i = f;
```

- **Step 3** – [Display Q[i] value.]

```
printf("%d", Q[i]);
```

- **Step 4 –[Increments the value of i by 1]**

```
i = i + 1;
```

- **Step 5** – Repeat step-3 and step-4 until the value of ' i'becomes greater than the value of rear pointer (**r**).

- **Step 6** – Exit.

**W.a.p. to implement Queue. Perform Operations on Queue.**

```
#include<conio.h>
#include<iostream.h>
#include<stdlib.h>
#include<stdio.h>
class queue
{
        int q[10],f,r,n;
        public:
        queue()
        {
                f =-1;
                r=-1;
                n=10;
        }
        void insert(int data)
        {
                if(r==n-1)
                {
                        cout<<"\n\t Queue is overflow";
                        return;
                }
                r++;
                q[r]=data;
                if(f==-1)
                        f=0;
        }
        int del()
        {
                int data;
                if(f==-1)
                {
                        cout<<"\n\t Queue is underflow";
                        return 0;
                }
                data=q[f];
                if(f==r)                                // Q is empty?
                {
                        f=-1;
                        r=-1;
                }
                else
                        f++;
```

```
            return data;
        }
        void display()
        {
            int i;
            if(f==-1)
                cout<<"\n\t Queue is Empty";
            else
            {
                cout<<"\n the Queue is\n";
                for(i=f;i<=r;i++)
                {
                    cout<<q[i]<<"\t";
                }
            }
        }
};
void main()
{
 int e,ch;
 queue q;
 do
 {
            clrscr();
            cout<<"\n\t\t  queue implementation";
            cout<<"\n\t1:Insert";
            cout<<"\n\t2:Delete";
            cout<<"\n\t3:Display";
            cout<<"\n\t4:Exit";

            cout<<"\n\tEnter your choice :";
            cin>>ch;
            switch(ch)
            {
                case 1:
                    cout<<"\n\tEnter the element:";
                    cin>>e;
                    q.insert(e);
                    break;
                case 2:
                    e=q.del();
                    if(e!=0)
                        cout<<"\n\tThe deleted element is "<<e;
                    break;
```

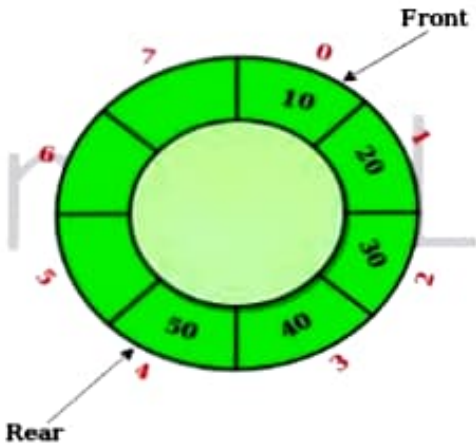
```
                case 3:
                        q.display();
                        break;
                case 4:
                        cout<<"\n Bye Bye";
                        exit(0);
                        break;
                default:
                        cout<<"Wrong choice";
        }
        getch();
    }while(ch!=4);

}
```

## CIRCULAR QUEUE

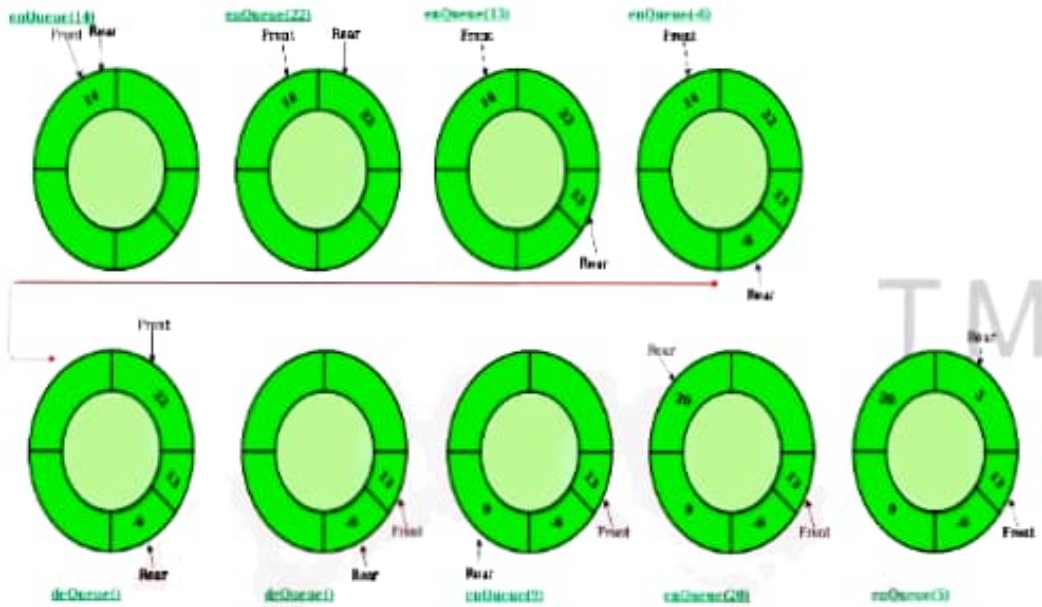
A circular queue is a type of queue in which the last position is connected to the first position to make a circle.

**Circular Queue** is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Circular queue is also known as **'Ring Buffer'**. In circular queue, like the simple queue we have to manage two pointers front and rear. Initially both the pointers front (f) and rear (r) are set to -1.




In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front position of a queue.



**Insert Operation on CQUEUE:**

In a circular queue, the new element is always inserted at Rear position.

Algorithm to perform inserts operation on circular Queue:

**Step 1:** Check whether Cqueue is Full or not.

**Step 2:** If CQueue is full means, if ((rear == size-1 && front == 0) || (front-rear ==1)) then display error message "CQueue is Full" and return.

**Step 3:** If CQueue is not full then, check if (rear == size – 1) then initialise rear pointer to 0 (zero) otherwise increment rear pointer by 1.

**Step 4:** Insert element at the rear position.

**Step 5:** Check, if front pointer is at -1 then initialise front pointer to 0 (zero).

**Step 6:** Exit

**Delete Operation on CQUEUE:**

In a circular queue, the existing element is always removed from the front position.

Algorithm to perform deletes operation on Circular Queue:

**Step 1:** Check whether CQueue is underflow or not.

**Step 2:** If CQueue is underflow means, if (front == -1)
then display error message "CQueue is underflow" and return with 0 (zero).

**Step 3:** If CQueue is not underflow then, remove element from the front position and store it into variable 'data'.

**Step 4:** if front and rear both are at the same position means, if(front == rear) then initialise both front and rear pointer to -1, if (front == size – 1) then initialise front pointer to 0 (zero) otherwise increment front pointer by 1.

**Step 5:** return data: the deleted element.

**Step 6:** Exit

## Display Operation on CQUEUE:

The display() function is used to display elements of the circular queue.
We can use the following steps to display the elements of the circular queue.

- **Step 1 – [Checks if the CQueue is empty or not.]**
  **[If the queue is empty, display an error message]**
  If ( f == – 1 )
  print "CQueue is Empty"

- **Step 2**– [If the CQueue is not empty then checks whether the value of **'f'** (front pointer) is greater than the value of **'r'**(rear pointer) or not.]

- **Step 3** –if the value of **'f'** (front pointer) is greater than the value of **'r'** (rear pointer) Then,
  **a)** Define variable 'i' and initialize it with thefront Pointer **(f)** ]
  int i = f;
  **b)** [Display Q[i] value.]
  print **"%d"**, Q[i]);
  **c)** [Increments the value of i by 1]
  i = i + 1;
  **d)** Repeat **step- b) and step- c)** until the value of ' i' becomes the value of size.

**After that, perform the following steps:**

**w)** Define variable 'i' and initialize it with zero. ]
```
int i = 0;
```

**x)** [Display Q[i] value.]
```
printf("%d", Q[i]);
```

**y)** [Increments the value of i by 1]
```
i = i + 1;
```

**z)** Repeat step- x) and step- y) until the value of ' i' becomes greater than 'r' (rear pointer)

- **Step 4**–if the value of **'f'** (front pointer) is not greaterthan the value of **'r'** (rear pointer)
  Then,
  **a)** define variable 'i' and initialize it with thefront Pointer **(f)** ]
  ```
  int i = f;
  ```
  **b)** [Display Q[i] value.]
  ```
  printf("%d", Q[i]);
  ```
  **c)** [Increments the value of i by 1]
  ```
  i = i + 1;
  ```
  **d)** Repeat **step-b) and step- c)** until the value of ' i' becomes greaterthan **'r'** (rear pointer)

- **Step 5**– Exit.

**W.a.p. to Implement Circular Queue**

```
#include<conio.h>
#include<stdio.h>
class queue
{
    int q[10],f,r,n;
    public:
    queue()
    {
            f=-1;
            r=-1;
```

```
            n=10;
 }
 void insert(int data)
 {
        if((f==0 && r==n-1) || (f-r==1))
        {
              cout<<"\n\t CQueue is overflow";
              return;
        }
        if(r==n-1)
              r=0;
        else
              r++;

        q[r]=data;

        if(f==-1)
            f=0;
}

int del()
{
        int data;
        if(f==-1)
        {
              cout<<"\n\t CQueue is underflow";
              return 0;
        }
        data=q[f];
        if(f==r)                              // CQ is empty?
        {
              f=-1;
              r=-1;
        }
        else if(f==n-1)
              f=0;
        else
            f++;

        return data;
}

void display()
{
```

9. What is double ended queue? Explain difference between input restricted and output restricted Dqueue.
10. What is D-queue? Write an algorithm to insert and delete elements from output restricted D-queue.
11. Write a note on Output restricted D-queue.
12. Write a note on Input restricted D-queue.