Unit-I
Introduction

Object Oriented Programming:

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time.

Benefits (Merits) of OOP:

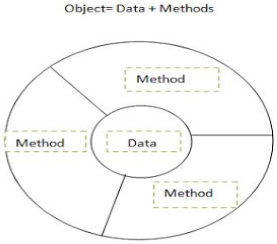
- Importance is given to data rather than algorithm
- Data abstraction is introduced in addition to procedural abstraction.
- Data and associated operations are unified into a single unit
- The objects are grouped with common attribute operation and Semantics
- Programs are designed around the data being operated rather than operate themselves.
- Reusability of code using inheritance.
- Providing various levels of Data protection using visibility control
- Providing Protection to Data and Functions both.
- Helps in creating Flexible, Extensible and Maintainable code.
- Effective solutions for Real World problems
- Quick software development due to Reusability

Demerits of OOP:

- Will not suit for concurrent problems
- Compile time and runtime overhead.
- Unfamiliarity causes training overhead.

Object Oriented Paradigm:

Object-oriented programming (OOP) is a programming paradigm. In this, the program is built around the objects.



Basic Concepts of Object Oriented Programming:

Object-oriented programming (OOP) is a programming paradigm. In this programs are built around the objects.

The basic concepts of OOP includes

Object

Class

Data Abstraction & Encapsulation

Inheritance

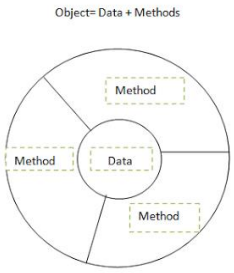
Polymorphism

Dynamic Binding

Message Passing

Object:

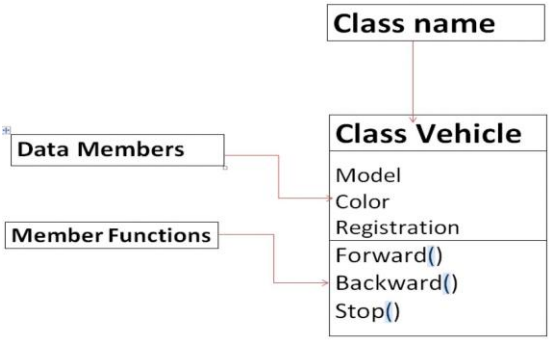
Object is the basic unit of object-oriented programming. An Object consists data members and associated member functions.



Class:

Class is a user defined data type. It combines data members and member functions into single unit. Several individual similar objects form a class.

Class represents the object. And object is a variable of type class.



Encapsulation & Data abstraction:

Method of combining data and its functions into a single unit is called Encapsulation. When using Data Encapsulation, data is not accessed directly, it is only accessible through the functions present inside the class. Data Abstraction It refers to the act of representing essential features without including the background details. It is the Black Box analogy, it means what is there inside we can't see but we can use it. For example in case of Television user don't see inner components of Television but he/she can enjoy the Television programmes.

Inheritance:

Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class, the new class that is formed is called derived class. Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

It is classifieds into different types, they are

Single level inheritance

Multi-level inheritance

Hybrid inheritance

Hierarchical inheritance

Polymorphism:

Poly (Greek term) means ability to take more than one form. Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an existing operator or function begins to operate on new data type, or class, it is considered as overloaded and it is a form of polymorphism.

The two ways of implementing polymorphism is

Function Overloading

Operator Overloading

Dynamic binding:

It contains a concept of Inheritance and Polymorphism. In this the object takes operations at runtime.

Message Passing:

It refers to establishing communication between one member function to other member function.

Applications of OOP:

Main application areas of OOP are:

User interface design such as windows, menus,…

Real Time Systems

Simulation and Modeling

Object oriented databases

AI and Expert System

Neural Networks and parallel programming

Decision support and office automation system

Introduction to C++ and History of C++:

During 1970's C language got popularity as general purpose language.

At the same time there was some other language called Simula-67.

To handle large projects Simula was suitable. And C has lot of features to implement general purpose programs.

So designers combined features of both the languages and formed a new language called 'C WITH CLASSES'

Bjarne Stroustrup worked with this C WITH CLASS as a part of his Ph.D work and added some features to the language.

In 1983 the language is renamed as C++. (An increment of C)

Character set of C++:

The C++ supports a group of characters as listed below:-

1. Digits 0-9
2. Alphabets

i) Lower case letters a-z

ii) Upper case letters A-Z

3. Special characters +, -, *, /, !, #, $, %, &, ‘, <, >, ?, /, \, :, ; etc.

Example Programs of C++:

A simple C++ program to display "Hello, World!" on the screen. Since it's an elementary program; it is often used to illustrate the syntax of a programming language.

Example Program:

```
#include <iostream.h>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}
```

Output:

Hello, World!

Every C++ program starts from the main() function.

The cout is the standard output stream which prints the "Hello, World!" string on the monitor.

Tokens of C++:

The smallest individual unit in C++ program is called token. The tokens in C++ program are listed below:

Keywords

Variables

Constants

Special character

Operators

Keywords:

The C++ keywords are reserved words. They have fixed meanings. All the C keywords are valid in C++. There are 63 keywords in C++. In addition to 32 keywords of C, C++ supports the following key words:

asm	private
catch	protected
class	public
delete	template

friend	This
inline	throw
new	Try
operator	virtual

Variable:

A variable is used to store values. Every variable has memory location. The memory locations are used to store the values of the variables. The variables can be of any type. The variable can hold single value at a time. The program can change value of variable.

Variable declaration:

Syntax:

Data type variable name;

Or

Data type variable name1, variable name2 …….. Variable name n;

Example:

int a,b,c;

float x;

Constants:

The constants in C++ are the values which do not change during execution of program. C++ supports various types of constants including integers, characters, floating and string constants. We can divide the constants into 2 types:

Literal constant:

A literal constant is directly assigned to a variable. Consider the following example

int x=5;

Where x is a variable of type int and 5 is a literal constant.

Symbolic constant:

The symbolic constant can be created in the following ways

#define

the const keyword

The #define preprocessor directive can be used for defining constants.

Example: # define PRICE 152

We can also define a constant using const keyword

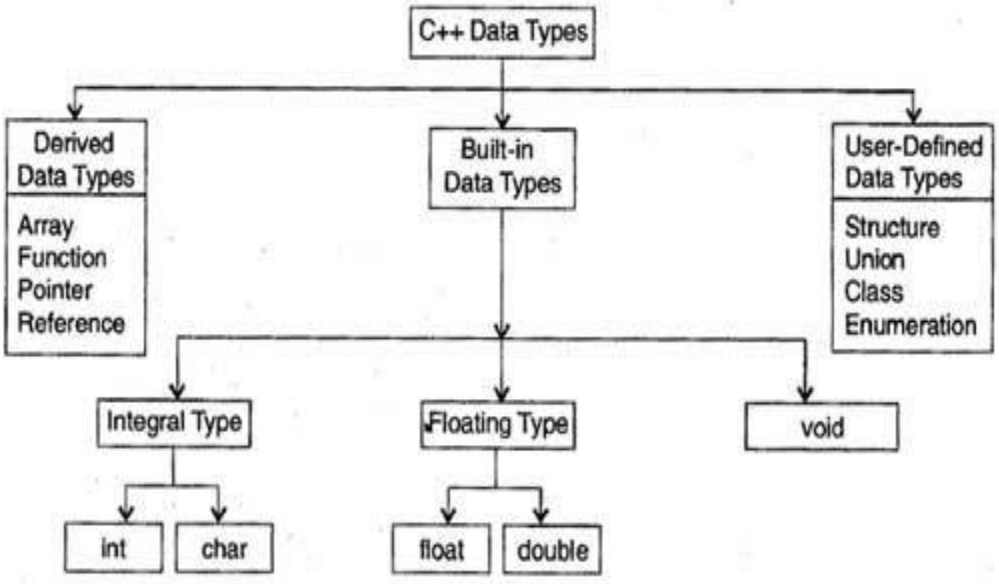
Example: const int price=152;

Special symbols:

In C++ program we use different special symbols such as #, &, ?, :, % etc.

Data types:

It determines the type and the operations that can be performed on the data. C++ provides various data types and each data type is represented differently within the computer's memory. The various data types provided by C++ are built-in data types, derived data types and user-defined data types as shown in Figure.



Various Data Types in C++

Built-in data type:

The Built-in data type include the following

Integer type

Floating point type

Character type

Integer Type:

Integers are whole numbers. C has 3 classes of integer storage namely short int, int and long int.

Floating Point Types:

Floating point number represents a real number with 6 digits precision. Floating point numbers are denoted by the keyword float.

Character Type:

A single character can be defined as a character type of data.

The basic data types supported by C are described with their size in bytes and ranges in the following table

Data type	Size in bytes	Range
Char	1	-128 to127
unsigned char	1	0 to 255
int	2	-32768 to 32767
unsigned int	2	0 to 65536

long int	4	-2,147,483,648 to 2,147,483,647
Float	4	3.4 E -38 to 3.4E+38
Double	8	1.7 E -308 to 1.7 E +308
long double	10	3.4 E -4932 to 1.1E +4932

Derived data type:

The derived data types are of following types:

Pointers

Functions

Arrays

Reference

User defined data type:

User defied data types are of the following types:

Structure

Union

Class

Enumeration data type

Operators:

C++ supports wide variety of operators such as

Arithmetic operators

Logical operators

Relational operators

Bit wise operators

Assignment Operators

Special purpose operators

Arithmetic Operators:

The arithmetic operator is a binary operator, which requires two operands to perform its operation of arithmetic. Following are the arithmetic operators that are available:

Operator	Description
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulo or remainder

At the time of using the '/' division operator on two integers, the result will also be an integer. E.g. 100/8 will result in 12 and not 12.50.

Program Example of Arthematic Operators:

```
#include <iostream>
```

```
using namespace std;
main()
{
int a = 30;
int b = 10;
int c ;
c = a + b;
cout << "Line 1 - Value of c is :" << c << endl ;
c = a - b;
cout << "Line 2 - Value of c is  :" << c << endl ;
c = a * b;
cout << "Line 3 - Value of c is :" << c << endl ;
c = a / b;
cout << "Line 4 - Value of c is  :" << c << endl ;
c = a % b;
cout << "Line 5 - Value of c is  :" << c << endl ;
return 0;
}
```

Output:

Line 1 - Value of c is: 40

Line 2 - Value of c is: 20

Line 3 - Value of c is: 300

Line 4 - Value of c is: 3

Line 5 - Value of c is: 1

Logical Operators:

A logical operator is used to compare or evaluate logical and relational expressions. There are three logical operators available in the Turbo C++ language.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Program Example of Logical Operators:

```
include <iostream>
using namespace std;
main()
{
int a = 5;
```

```
int b = 20;
int c ;
if(a && b)
{
cout << "Line 1 - Condition is true"<< endl ;
}
if(a || b)
{
cout << "Line 2 - Condition is true"<< endl ;
}
/* Let's change the values of  a and b */
a = 0;
b = 10;
if(a && b)
{
cout << "Line 3 - Condition is true"<< endl ;
}
else
{
cout << "Line 4 - Condition is not true"<< endl ;
}
if(!(a && b))
{
cout << "Line 5 - Condition is true"<< endl ;
}
return 0;
}
```

Output:

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 4 - Condition is not true
Line 5 - Condition is true
```

Relational Operators:

Relational operators compare two operands and return true or false i.e. 1 or 0.

Following are the various relational operators

Operator	Meaning
<	Less than

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

Program Example of Relational Operators:

```
#include <iostream>
using namespace std;
main()
{
int a = 21;
int b = 10;
int c ;
if( a == b )
{
cout << "Line 1 - a is equal to b" << endl ;
}
else
{
cout << "Line 1 - a is not equal to b" << endl ;
}
if( a < b )
{
cout << "Line 2 - a is less than b" << endl ;
}
else
{
cout << "Line 2 - a is not less than b" << endl ;
}
if( a > b )
{
cout << "Line 3 - a is greater than b" << endl ;
}
else
{
cout << "Line 3 - a is not greater than b" << endl ;
}
```

```
/* Let's change the values of a and b */
a = 5;
b = 20;
if( a <= b )
{
cout << "Line 4 - a is either less than \ or equal to  b" << endl ;
}
if( b >= a )
{
cout << "Line 5 - b is either greater than \ or equal to b" << endl ;
}
return 0;
}
```

Output:

Line 1 - a is not equal to b

Line 2 - a is not less than b

Line 3 - a is greater than b

Line 4 - a is either less than or equal to b

Line 5 - b is either greater than or equal to b

Bit wise operators:

The bitwise operators operate on sequences of binary bits. Bitwise operations are necessary for much low-level programming, such as writing device drivers, low-level graphics.

Bitwise operators include:

Operator	Meaning
&	AND
|	OR
^	XOR
~	one's compliment
<<	Shift Left
>>	Shift Right

Program Example of Bitwise Operators:

```
#include <iostream>
using namespace std;
main()
{
unsigned int a = 60;          // 60 = 0011 1100
unsigned int b = 13;          // 13 = 0000 1101
```

```
int c = 0;
c = a & b;            // 12 = 0000 1100
cout << "Line 1 - Value of c is : " << c << endl ;
c = a | b;            // 61 = 0011 1101
cout << "Line 2 - Value of c is: " << c << endl ;
c = a ^ b;            // 49 = 0011 0001
cout << "Line 3 - Value of c is: " << c << endl ;
c = ~a;               // -61 = 1100 0011
cout << "Line 4 - Value of c is: " << c << endl ;
c = a << 2;           // 240 = 1111 0000
cout << "Line 5 - Value of c is: " << c << endl ;
c = a >> 2;           // 15 = 0000 1111
cout << "Line 6 - Value of c is: " << c << endl ;
return 0;
}
```

Output:

```
Line 1 - Value of c is: 12
Line 2 - Value of c is: 61
Line 3 - Value of c is: 49
Line 4 - Value of c is: -61
Line 5 - Value of c is: 240
Line 6 - Value of c is: 15
```

Assignment Operator:

An assignment operator (=) is used to assign a constant or a value of one variable to another.

Example:

```
a = 5;
b = a;
interestrate = 10.5;
result = (a/b) * 100;
```

Program Example of Assignment Operators:

```
#include <iostream>
using namespace std;
main()
{
int a = 21;
int c ;
c =  a;
```

```
cout << "Line 1 - =  Operator, Value of c = : " <<c<< endl ;
c +=  a;
cout << "Line 2 - += Operator, Value of c = : " <<c<< endl ;
c -=  a;
cout << "Line 3 - -= Operator, Value of c = : " <<c<< endl ;
c *=  a;
cout << "Line 4 - *= Operator, Value of c = : " <<c<< endl ;
c /=  a;
cout << "Line 5 - /= Operator, Value of c = : " <<c<< endl ;
c  = 200;
c %=  a;
cout << "Line 6 - %= Operator, Value of c = : " <<c<< endl ;
c <<=  2;
cout << "Line 7 - <<= Operator, Value of c = : " <<c<< endl ;
c >>=  2;
cout << "Line 8 - >>= Operator, Value of c = : " <<c<< endl ;
c &=  2;
cout << "Line 9 - &= Operator, Value of c = : " <<c<< endl ;
c ^=  2;
cout << "Line 10 - ^= Operator, Value of c = : " <<c<< endl ;
c |=  2;
cout << "Line 11 - |= Operator, Value of c = : " <<c<< endl ;
return 0;
}
```

Output:

```
Line 1 - =  Operator, Value of c = : 21
Line 2 - += Operator, Value of c = : 42
Line 3 - -= Operator, Value of c = : 21
Line 4 - *= Operator, Value of c = : 441
Line 5 - /= Operator, Value of c = : 21
Line 6 - %= Operator, Value of c = : 11
Line 7 - <<= Operator, Value of c = : 44
Line 8 - >>= Operator, Value of c = : 11
Line 9 - &= Operator, Value of c = : 2
Line 10 - ^= Operator, Value of c = : 0
Line 11 - |= Operator, Value of c = : 2
```

Important Note:

Remember that there is a remarkable difference between the equality operator (==) and the assignment operator (=). The equality operator is used to compare the two operands for equality (same value), whereas the assignment operator is used for the purposes of assignment.

Special operators:

The operators such as size of and comma are treated as special operators.

Size of Operator:

The size of operator gives the amount of storage, in bytes, required to store an operand.

Syntax:

sizeof unary-expression

sizeof (type-name)

Example: sizeof(char) is equal to 1.

Comma operator:

We can combine multiple expressions in a single expression using the comma operator

Expressions:

It consists of a single entity such as a constant, a variable or can be a combination of such entities joined together using one or more operators.

Example:

x=y;

c=a+b*d;

Control Structures:

C++ allows many kinds of control structures, which include:

Conditional control structure:

if

if else

else if ladder

nested if

Jumping control structure:

goto

break

continue

return

Iterative control structure:

for loop

while loop

do while loop

Multi-way conditional control structure:

switch

Conditional control structure:

if statement:

if statement checks whether the test expression inside parenthesis ( ) is true or not. If the test expression is true, statement(s) inside the body of if statement is executed but if test is false, statement(s) inside body of if is ignored.

syntax:

```
if (test expression)
{
statement(s) to be executed if test expression is true;
}
```


Explanation:

Statements get executed only when the condition is true. Braces are not necessary if only one statement is related to the condition.

Example:

```
if(x>0)
cout<<“positive”;
```

Program Example of if Condition:

```
#include <iostream>
using namespace std;
int main()
{
int a = 10, b = 30;
if (b > a)
{
cout << "b is greater" << endl;
}
system("PAUSE");
}
```

Output:

b is greater

press any key to continue . . . .

if else statement:

The if...else statement is used if the programmer wants to execute some statement/s when the test expression is true and execute some other statement/s if the test expression is false.

syntax:

```
if (condition)
{
statements
}
else
{
statements
}
```


Explanation:

Statements in if block get executed only when the condition is true and statements in else block get executed only when condition is false.

Example:

```
if(x>y)
cout<<“x is big”;
else
cout<<”Y is big”;
```

Program Example of if else Condition:

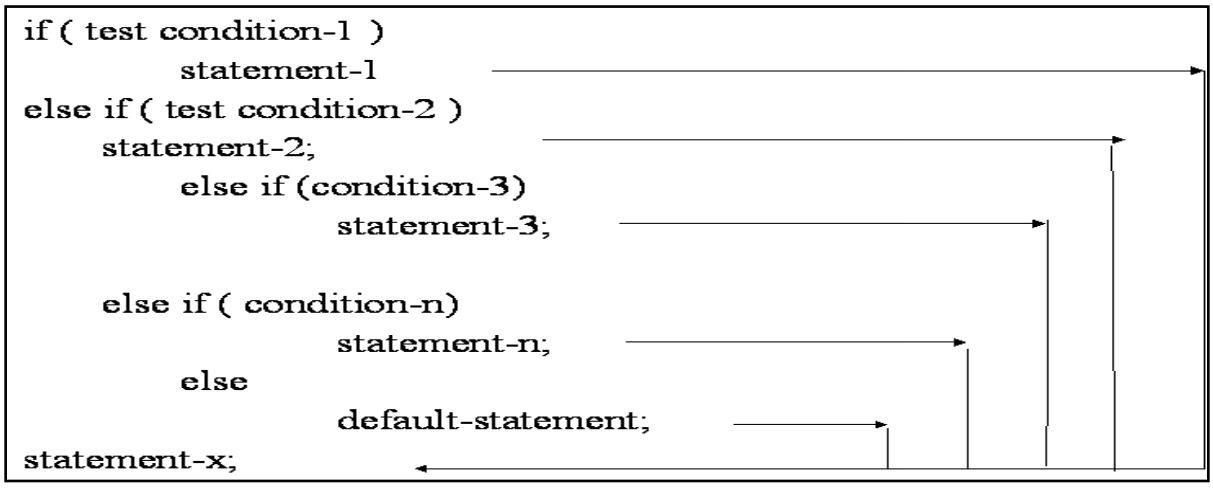
```
#include <iostream>
using namespace std;
int main()
{
int a = 15, b = 20;
if (b > a)
{
cout << "b is greater" << endl;
}
else
{
cout << "a is greater" << endl;
}
system("PAUSE");
}
```

Program Output:

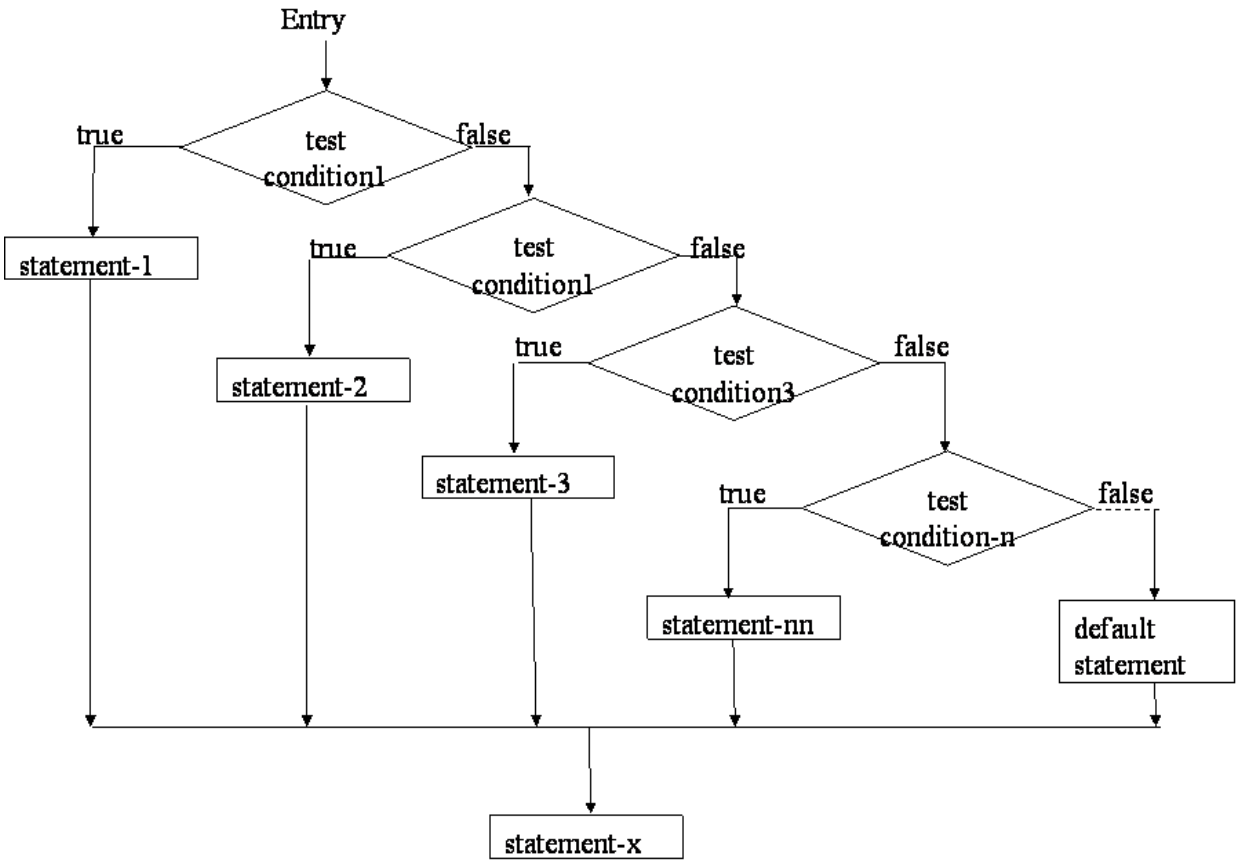
```
b is greater
press any key to continue . . . .
```

else if ladder:

When multi path decisions are involved we use chain of if else statements. It forms else if ladder.



The flow chart for else if ladder is given below:



syntax:

```
if (condition1)
{
statements
}
else if (condition2)
{
statements
}
else if (condition3)
{
statements
}
else
{
statements
}
```

Explanation:

Statements get executed in if blocks only when the corresponding conditions are true. Statements in else block get executed when all other conditions are false.

Example:

```
if(x>0)
cout<<“positive”;
else if (x<0)
cout<<“negative”;
else
cout<<“zero”;
```

Program Example of if else ladder Condition:

```
#include <iostream>
using namespace std;
int main()
{
int score;
cout << "Enter your score between 0-100\n";
cin >> score;
/* Using if else ladder statement to print
Grade of a Student */
if(score >= 90)
{
// Marks between 90-100
cout << "YOUR GRADE : A\n";
}
else if (score >= 70 && score < 90)
{
// Marks between 70-89
cout << "YOUR GRADE : B\n";
}
else if (score >= 50 && score < 70)
{
// Marks between 50-69
cout << "YOUR GRADE : C\n";
}
else
{
// Marks less than 50
```

```
// if none of the conditions is true
cout << "YOUR GRADE : Failed\n";
}
return 0;
}
```

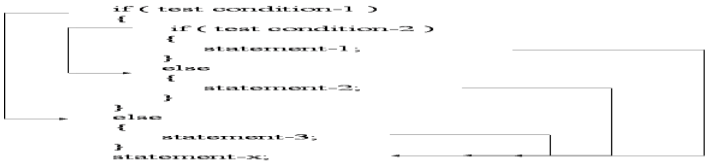
Output:

```
Enter your score between 0-100
72
Your Grade: B
Enter your score between 0-100
10
Your Grade: Failed
```

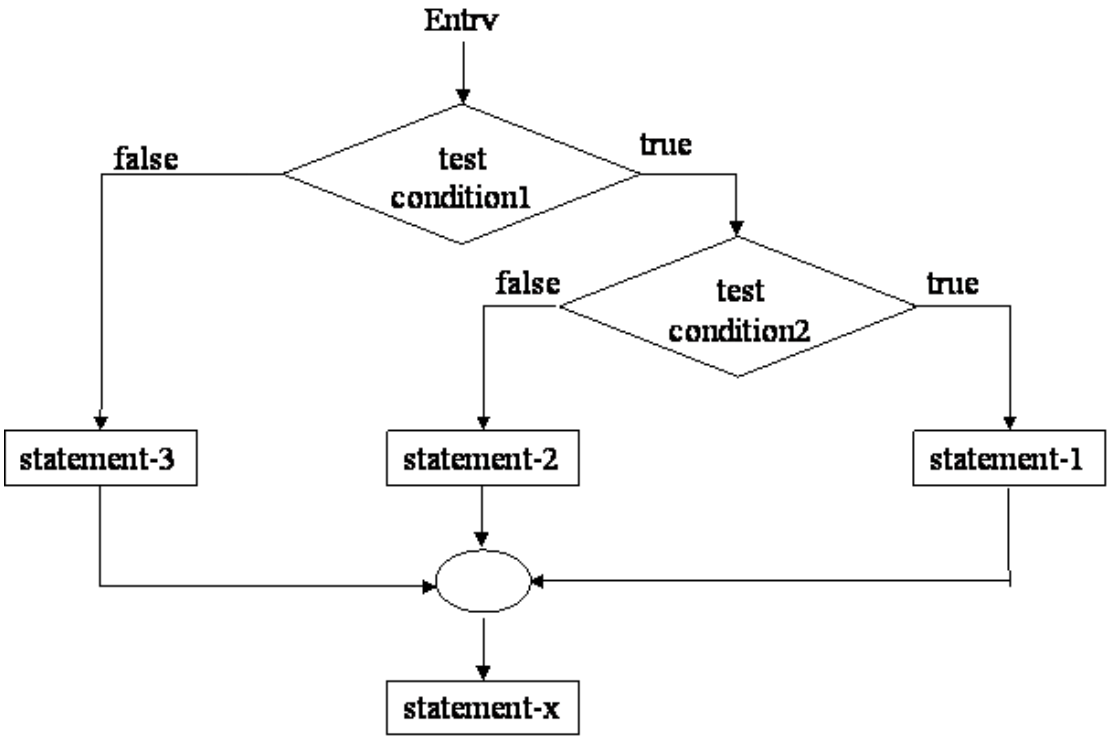
Nested if else statement:

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

Syntax:



The flow chart for the nested if else statement is given below:



Syntax:

```
if(condition1)
{
if (condition2)
{
statements
}
}
```

Explanation:

Statements get executed only when condition1 and condition2 are true.

Example:

```
if(a>b)
{
if(a>c)
cout<<“ a is the biggest”;
}
```

Program Example of if else Condition:

```
#include <iostream>
using namespace std;
int main ()
{
// local variable declaration:
int a = 100;
int b = 200;
// check the boolean condition
if( a == 100 )
{
// if condition is true then check the following
if( b == 200 )
{
// if condition is true then print the following
cout << "Value of a is 100 and b is 200" << endl;
}
}
cout << "Exact value of a is : " << a << endl;
cout << "Exact value of b is : " << b << endl;
return 0;
}
```

Jumping Control Structure:

goto statement:

syntax:

jumping forward:

statement1

goto label;

statement 2

label:

statement 3

jumping back:

statement 1

label:

statement 2

goto label:

statement 3

Explanation:

Statement 2 gets omitted because after statement 1 gets executed goto moves the control to the labeled statement i.e. statement 3

After statement 1 gets executed statement 2 gets executed for infinite times, because goto moves the control to statement 2 repeatedly.

Example:

Jumping forward

```
x=3;
if(x>0)
goto ABC:
x=-x;
ABC:
cout<<“x=%d”,x;
```

Break statement:

break statement quits the corresponding iterative loop. Valid only in iterative loops and switch loop.

Example:

```
for(i=0;i<10;i++)
{
cout<<“hello”;
if(i==5)
break;
}
```

/* prints hello 5 times.*/

Continue statement:

Continue moves the control the first statement in the corresponding iterative loop. Valid only in iterative loops.

Example:

```
i=0;
while(1)
{
cout<<“hello”;
if(i<5)
continue;
break;
}
/* prints hello 5 times.*/
```

Return statement:

Return is used in functions. It moves the control to calling function form the called module.

Example:

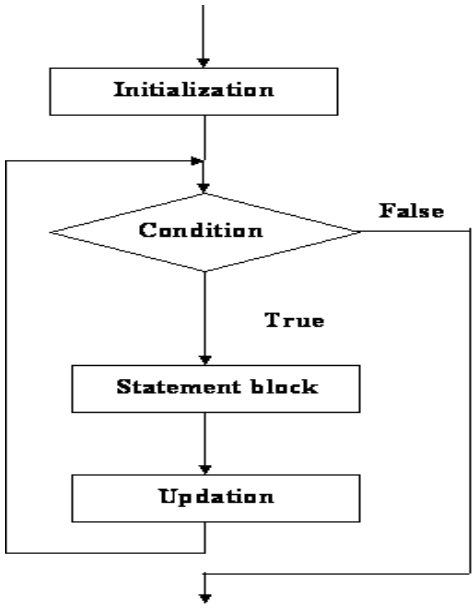
```
int sum( int x, int y)
{
return (x+y);
}
```

Iterative Control Structure:

for loop:

The for loop is also an entry controlled loop like while loop. In this loop we combine initialization, condition and updation at one location.

The flow chart of for loop is given below:



The general form of for loop is given below:

for (initialization; condition; updation)

{

statement block

}

The initialization is used to set a counter variable to its starting value. The condition is generally a relational statement that checks the counter variable against a termination value the updation increments (or decrements) the counter value. The loop repeats until the condition becomes false.

syntax:

for(expression1 ; condition; expression2)

{

statements

}

Explanation:

Statements get executed as long as the condition is true. Generally expression1 includes initializing statements and expression2 is the condition and expression 3 is the updation.

Example:

for(i=1;i<=10;i++)

cout<<“hello”;

//prints hello 10 times

Example program of For Loop:

#include <iostream>

using namespace std;

```
int main ()
{
// for loop execution
for( int a = 10; a < 20; a = a + 1 )
{
cout << "value of a: " << a << endl;
}
return 0;
}
```

Output:

Value of a: 10

Value of a: 11

Value of a: 12

Value of a: 13

Value of a: 14

Value of a: 15

Value of a: 16

Value of a: 17

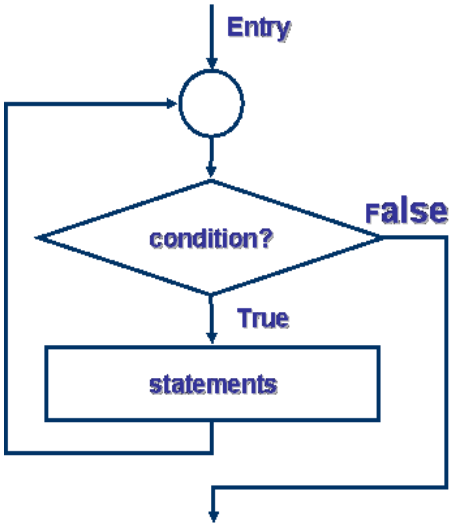
Value of a: 18

Value of a: 19

While loop:

The while loop executes a simple statement or a block of statements until the conditional expression becomes FLASE.

The flow chart for the while loop is given below:



syntax:

while(condition)

```
{
statements
}
```

Explanation:

Statements get executed as long as the condition is true. It checks condition first and executes the statements later.

Example:

```
i=0;
while(i<10)
{
cout<<“hello”;
i++;
}           /* prints hello 10 times*/
```

Example program of while Loop:

```
#include <iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int a = 10;
// while loop execution
while( a < 20 )
{
cout << "value of a: " << a << endl;
a++;
}
return 0;
}
```

Output:

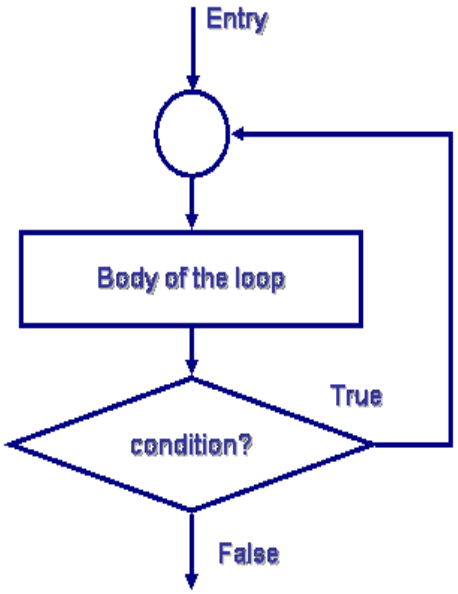
```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
```

Value of a: 19

do while loop:

On some occasions, it might be necessary to execute the body of the loop before the test is performed. This can be handled by *do…while* statement.

The flow chart for do while is given below:



syntax:

do

{

statements

} while(condition);

Explanation:

Statements get executed as long as the condition is true except for the for the first time. It executes the statements first and checks the condition later.

Example:

i=0;

while(i<10)

{

cout<<“hello”;

i++;

} /* prints hello 10 times*/

Example program of do while Loop:

#include <iostream.h>

using namespace std;

int main ()

```
{
// Local variable declaration:
int a = 10;
// do loop execution
do
{
cout << "value of a: " << a << endl;
a = a + 1;
}
while( a < 20 );
return 0;
}
```

Output:

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 15
Value of a: 16
Value of a: 17
Value of a: 18
Value of a: 19
```

Multi-way conditional control structure:

Switch statement:

Syntax:

```
Switch(variable)
{
case constant1: statement1
break;
case constant2: statement2
break;
case constant3: statement3
break;
default: statements
break;
}
```

Explanation:

Switch statement is similar to if else if statement. Statement 1 gets executed when the variable is equal to constant1 and so on. Control comes to default portion when all the constants, which have been mentioned do not match with the variable. Here break and default are optional. When there is no break for a case it continues till it encounters break or the end of the switch.

Example:

```
k=x-y;
switch(k)
{
case 0 :
cout<<“x and y are equal”;
break;
default:
cout<<“x and y are not equal”;
break;
}
```

Example program of Switch Statement:

```
#include <iostream.h>
using namespace std;
int main ()
{
// local variable declaration:
char grade = 'D';
switch(grade)
{
case 'A' :
cout << "Excellent!" << endl;
break;
case 'B' :
case 'C' :
cout << "Well done" << endl;
break;
case 'D' :
cout << "You passed" << endl;
break;
case 'F' :
cout << "Better try again" << endl;
```

```
break;
default :
cout << "Invalid grade" << endl;
}
cout << "Your grade is " << grade << endl;
return 0;
}
```

Output:

```
You passed
Your grade is D
```

Differences between while and do while loop:

while loop:

while loop gets executed as long as the condition is true

Condition is evaluated first and then statements.

Minimum number of execution in this case is zero.

Syntax:

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

do while loop:

Do while loop gets executed as long as the condition is true, but not in case of first time execution

Statements get executed first and then condition is checked

Minimum number of execution in this case is one

Syntax:

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

Functions:

Function is a block of statements that solve a task or sub task of other task.

Types of functions:

C++ supports two types of functions. They are **library functions** and **user defined functions**.

The library functions can be used in any program by including respective header files.

The header files must be included using # include preprocessor directive.

The programmer can also define and use his/her own functions for performing some specific tasks. Such functions are called as user-defined functions.

The following are the advantages of functions:

Support for modular programming

Reduction in program size

Code duplication is avoided

Code reusability is provided

Functions can be called repetitively

A set of functions can be used to form libraries

Parts of Functions:

A Function has the following parts:

Function prototype

Definition of a function

Function call

Actual and formal arguments

The return statement

Function prototype:

A function prototype declaration consists of the function return type, name, and arguments list. It tells the compiler

(a) the name of the function,

(b) the type of value returned

(c) the type and number of arguments.

When the programmer defines the function, the definition of function must be same as its prototype declaration. If the programmer makes a mistake, the compiler flags an error message.

The prototype declaration statement is always terminated with semi-colon.

The following statements are the examples of function prototypes:

```
void show(void);
float sum(float, int);
```

Function Definition:

The first line is called function header and is followed by function body.

The block of statements followed by function header is called as function definition.

The header and function prototype declaration should match each other.

The function body is enclosed with curly braces.

The function can be defined anywhere.

If the function is defined before its caller, then its prototype declaration is optional

Function Call:

The function gets activated only when a call to function is invoked. A function must be called by its name, followed by argument list enclosed in parenthesis and terminated by semi colon.

Actual and formal Arguments:

The arguments declared in caller function and given in the function call are called as actual arguments. The arguments declared in the function header are known as formal arguments.

UNIT-II

CLASSES, OBJECTS, CONSTRUCTORS AND DESTRUCTORS

Encapsulation:

Method of combining data and its functions into a single unit is called Encapsulation. When using Data Encapsulation, data is not accessed directly, it is only accessible through the functions present inside the class.

Hiding:

Data hiding is a process of combining data and functions into a single unit. The ideology behind data hiding is to conceal data within a class, to prevent its direct access from outside the class. It helps programmers to create classes with unique data sets and functions, avoiding unnecessary penetration from other program classes.

Discussing data hiding & data encapsulation, data hiding only hides class data components, whereas data encapsulation hides class data parts and private methods.

Now you also need to know access specifier for understanding data hiding.

private, public & protected are three types of protection/ access Specifiers available within a class. Usually, the data within a class is private & the functions are public. The data is hidden, so that it will be safe from accidental manipulation.

Private members/methods can only be accessed by methods defined as part of the class. Data is most often defined as private to prevent direct outside access from other classes. Private members can be accessed by members of the class.

Public members/methods can be accessed from anywhere in the program. Class methods are usually public which is used to manipulate the data present in the class. As a general rule, data should not be declared public. Public members can be accessed by members and objects of the class.

Protected member/methods are private within a class and are available for private access in the derived class.

Abstract Data Type:

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

Object:

Object is the basic unit of object-oriented programming. An Object consists data members and associated member functions.

Class:

Class is a user defined data type. It combines data members and member functions into single unit. Several individual similar objects form a class.

Class represents the object. And object is a variable of type class.

Attributes:

Attributes are one of the key features of modern C++ which allows the programmer to specify additional information to the compiler to enforce constraints(conditions), optimise certain pieces of code or do some specific code generation**.** In simple terms, an attribute acts as an annotation or a note to the compiler which provides additional information about the code for optimization purposes and enforcing certain conditions on it. Introduced in C++11, they have remained one of the best features of C++ and are constantly being evolved with each new version of C++.

Methods:

A **method** is a member function of a class, but in **C++** they are more commonly called member functions than **methods** (some programmers coming from other languages like Java call them **methods**). A function is usually meant to mean a free-function, which is not the member of a class.

C++ Class Declaration:

A **class** in C++ is a user-defined type or data structure declared with keyword **class** that has data and functions (also called member variables and member functions) as its members whose access is governed by the three access Specifiers private, protected or public. By default access to members of a **C++ class** is private.

Constructors:

Constructors are special class functions which performs initialization of every object.

The Compiler calls the Constructor whenever an object is created.

Constructors initialize values to object members after storage is allocated to the object.

```
class A
{
int x;
public:
A();  //Constructor
};
```

The name of constructor will be same as the name of the class

Constructors never have return type.

Constructors can be defined either inside the class definition or outside of class definition using class name and scope resolution :: operator.

```
class A
{
int i;
public:
A(); //Constructor declared
};
A::A()   // Constructor definition
{
i=1;
}
```

Types of Constructors:

Constructors are of three types:

Default Constructor

Parameterized Constructor

Copy Constructor

Default Constructor:

Default constructor is the constructor which doesn't take any argument. It has no parameter.

Syntax:

```
class_name ()
{ Constructor Definition }
```

Example:

```
class Cube
{
int side;
public:
Cube()
{
side=10;
}
};
int main()
{
Cube c;
cout << c.side;
```

```
}
```

Output: 10

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

Parameterized Constructor:

These are the constructors with parameter. Using this Constructor we can provide different values to data members of different objects, by passing the appropriate values as argument.

Example:

```
class Cube
{
int side;
public:
Cube(int x)
{
side=x;
}
};
int main()
{
Cube c1(10);
Cube c2(20);
Cube c3(30);
cout << c1.side;
cout << c2.side;
cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized constructor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

Copy Constructor:

These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.

Syntax of Copy Constructor

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

```
}
class Cube
{
int side;
public:
Cube(Cube& ob)  //copy constructor
{
side=ob.size;
}
};
```

Constructor Overloading

Constructors can also be overloaded. We can have any number of Constructors in a class that differ in parameter list.

Example:

```
class Student
{
int rollno;
string name;
public:
Student(int x)
{
rollno=x;
name="None";
}
Student(int x, String str)
{
rollno=x ;
name=str ;
}
};
int main()
{
Student A(10);
Student B(11,"Ram");
}
```

Destructors:

Destructor is a special class function which destroys the object as soon as the scope of object ends.

The destructor is called automatically by the compiler

The syntax for destructor is same as constructor, the class name is used for the name of destructor, with a tilde ~ sign as prefix to it.

```
class A
{
public:
~A();
};
```

Destructors will never have any arguments.

Example to see how Constructor and Destructor is called

```
class A
{
A()
{
cout << "Constructor called";
}
~A()
{
cout << "Destructor called";
}
};
int main()
{
A obj1;   // Constructor Called
int x=1
if(x)
{
A obj2;  // Constructor Called
}   // Destructor Called for obj2
} //  Destructor called for obj1.
```

C++ garbage Collection:

Garbage collection is a form of automatic memory management. The garbage collector or collector attempts to reclaim garbage, or memory used by objects that will never be accessed or mutated again by the application.

Tracing garbage collectors require some implicit runtime overhead that may be beyond the control of the programmer, and can sometimes lead to performance problems. For example, commonly used Stop-The-World garbage collectors, which pause program execution at arbitrary times, may make garbage collecting languages inappropriate for some embedded systems, high-performance server software, and applications with real-time needs.

A more fundamental issue is that garbage collectors violate locality of reference, since they deliberately go out of their way to find bits of memory that haven't been accessed recently. The performance of modern computer architectures is increasingly tied to caching, which depends on the assumption of locality of reference for its effectiveness. Some garbage collection methods result in better locality of reference than others. Generational garbage collection is relatively cache-friendly, and copying collectors automatically defragment memory helping to keep related data together. Nonetheless, poorly timed garbage collection cycles could have a severe performance impact on some computations, and for this reason many runtime systems provide mechanisms that allow the program to temporarily suspend, delay or activate garbage collection cycles.

Despite these issues, for many practical purposes, allocation/deallocation-intensive algorithms implemented in modern garbage collected languages can actually be faster than their equivalents using explicit memory management (at least without heroic optimizations by an expert programmer). A major reason for this is that the garbage collector allows the runtime system to amortize allocation and deallocation operations in a potentially advantageous fashion. For example, consider the following program in C++:

Dynamic Memory Allocation:

Programmers can dynamically allocate storage space while the program is running, but programmers cannot create new variable names "on the fly", and for this reason, dynamic allocation requires two criteria:

- Creating the dynamic space in memory
- Storing its address in a pointer (so that space can be accessed)

Memory de-allocation is also a part of this concept where the "clean-up" of space is done for variables or other data storage. It is the job of the programmer to de-allocate dynamically created space. For de-allocating dynamic memory, we use the delete operator. In other words, dynamic memory Allocation refers to performing memory management for dynamic memory allocation manually.

Memory in your C++ program is divided into two parts:

Stack: All variables declared inside any function takes up memory from the stack.

Heap: It is the unused memory of the program and can be used to dynamically allocate the memory at runtime.

Meta Class:

In object-oriented programming, a metaclass is a class whose instances are classes. Just as an ordinary class defines the behavior of certain objects, a metaclass defines the behavior of certain classes and their instances. Not all object-oriented programming languages support metaclasses. Among those that do, the extent to which metaclasses can override any given aspect of class behavior varies. Metaclasses can be implemented by having classes be first-class citizen, in which case a metaclass is simply an object that constructs classes. Each language has its own metaobject protocol, a set of rules that govern how objects, classes, and metaclasses interact.

Abstract Class:

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

UNIT-III

Overloading, Conversions, Derived Classes and Inheritance

Operator overloading:

In C++ programming, it allows the programmer to redefine the meaning of operator when they operate on class objects is known as operator overloading.

Operator overloading cannot be used to change the way operator works on built-in types. Operator overloading only allows us to redefine the meaning of operator for user-defined types.

There are two operators assignment operator(=) and address operator(&) which does not need to be overloaded. Because these two operators are already overloaded in C++ library. For example: If obj1 and obj2 are two objects of same class then, we can use code obj1=obj2; without overloading = operator. This code will copy the contents object of obj2 to obj1. Similarly, we can use address operator directly without overloading which will return the address of object in memory.

Operator overloading cannot change the precedence of operators and associativity of operators. But, if we want to change the order of evaluation, parenthesis should be used.

Not all operators in C++ language can be overloaded. The operators that cannot be overloaded in C++ are ::(scope resolution), .(member selection), .*(member selection through pointer to function) and ?:(ternary operator).

To overload a operator, a operator function is defined inside a class as:

```
class class_name
{
........ ...... ......
public:
   return_type operator sign (argument/s)
   {
     ...... ..... ......
   }
........ ...... ......
};
```

/* Simple example to demonstrate the working of operator overloading*/

```
#include <iostream.h>
class temp
{
private:
int count;
public:
temp() {count=5 }
void operator ++() {
count=count+1;
}
void Display() { cout<<"Count: "<<count; }
```

```
};
int main()
{
temp t;
++t;        /* operator function void operator ++() is called */
t.Display();
return 0;
}
```

Output:

Count: 6

Overloading Arithmetic Operator (overloading binary operator)

Arithmetic operator is most commonly used operator in C++. Almost all arithmetic operators can be overloaded to perform arithmetic operations on user-defined data type. In the below example we have overridden the + operator, to add to Time(hh:mm:ss) objects.

Example:

```
overloading '+' Operator to add two time object
#include< iostream.h>
#include< conio.h>
class time
{
int h,m,s;
public:
time()
{
h=0, m=0; s=0;
}
void getTime();
void show()
{
cout<< h<< ":"<< m<< ":"<< s;
}
time operator+(time);   //overloading '+' operator
};
time time::operator+(time t1)          //operator function
{
time t;
int a,b;
```

```
a=s+t1.s;
t.s=a%60;
b=(a/60)+m+t1.m;
t.m=b%60;
t.h=(b/60)+h+t1.h;
t.h=t.h%12;
return t;
}
void time::getTime()
{
cout<<"\n Enter the hour(0-11) ";
cin>>h;
cout<<"\n Enter the minute(0-59) ";
cin>>m;
cout<<"\n Enter the second(0-59) ";
cin>>s;
}
void main()
{
clrscr();
time t1,t2,t3;
cout<<"\n Enter the first time ";
t1.getTime();
cout<<"\n Enter the second time ";
t2.getTime();
t3=t1+t2;            //adding of two time object using '+' operator
cout<<"\n First time ";
t1.show();
cout<<"\n Second time ";
t2.show();
cout<<"\n Sum of times ";
t3.show();
getch();
}
```

Overloading I/O operator (insertion and extraction operator)

In this case the object must be returned by reference overloaded function.

For insertion operator iostream and for extraction operator iostream objects are used.

The overloaded functions must be friend functions.

The arguments must be sent by reference

Example:

overloading '<<' Operator to print time object

```
#include< iostream.h>
#include< conio.h>
class time
{
int hr,min,sec;
public:
time()
{
hr=0, min=0; sec=0;
}
time(int h,int m, int s)
{
hr=h, min=m; sec=s;
}
friend iostream& operator << (iostream &out, time &tm);  //overloading '<<' operator
};
Iostream & operator<< (iostream &out, time &tm)      //operator function
{
cout << "Time is " << tm.hr << "hour : " << tm.min << "min : " << tm.sec << "sec";
return out;
}
void main()
{
time tm(3,15,45);
cout << tm;
}
```

Output:

Time is 3 hour : 15 min : 45 sec

Overloading Relational operator:

We can also overload Relational operator like `==` , `!=` , `>=` , `<=` etc. to compare two user-defined object.

Example:

```
#include< iostream.h>
class time
```

```
{
int hr,min,sec;
public:
time()
{
hr=0, min=0; sec=0;
}
time(int h,int m, int s)
{
hr=h, min=m; sec=s;
}
friend bool operator==(time &t1, time &t2);   //overloading '==' operator
};
bool operator== (time &t1, time &t2)                //operator function
{
return ( t1.hr == t2.hr &&
t1.min == t2.min &&
t1.sec == t2.sec );
}
void main()
{
time tm1(3,15,45);
time tm1(3,15,46);
cout << tm1==tm2;
}
```

Output:

False

Object Conversion:

The type conversion is carried out when the expression contains different types of data items. When the compiler carries such type conversion itself by using in built data types then it is called implicit type conversion. The variable of lower type (small range) type when converted to higher type (large range) is known as promotion. When the variable of higher type is converted to lower type, it is called demotion.

The below table describes various implicit type conversion rules that a compiler follows.

Argument 1	Argument 2	Result
char	int	int
int	float	float
int	long	long

double	float	double
int	double	double
long	double	double
int	unsigned	unsigned

Derived Classes:

- Allows you to build new classes making use of previously defined ones.
- Use it to expand or modify the operation of a class.
- Derived class inherits all of the base classes members.
- Derived class has access to public and protected members.
- Base class constructor is called first. Then member constructors.

Concept of Reusability:

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirement. This done by creating new class reusing the properties of the existing ones is called reusability.

Visibility Modes:

When a **base class** is derived by a **derived class** with the help of inheritance, the accessibility of base class by the derived class is controlled by visibility modes. The derived class doesn't inherit access to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

There are three types of Visibility modes:

Public Visibility mode: If we derive a subclass 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 the derived class.

Protected Visibility mode: If we derive a subclass from a Protected base class. Then both public member and protected members of the base class will become protected in the derived class.

Private Visibility mode: If we derive a subclass from a Private base class. Then both public member and protected members of the base class will become Private in the derived class.

Inheritance:

- Inheritance is the capability of one class to acquire properties and characteristics from another class.
- The class whose properties are inherited by other class is called the Parent or Base or Super Class.
- And, the class which inherits properties of other class is called Child or Derived or Sub class.
- Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.
- All members of a class except Private, are inherited

Purpose of Inheritance:

Code Reusability

Method Overriding (Hence, Runtime Polymorphism.)

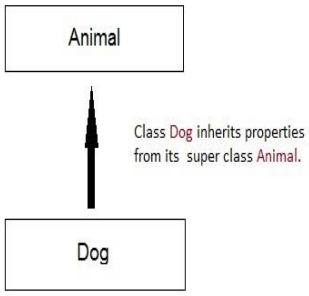
Use of Virtual Keyword

Basic Syntax of Inheritance:

Class subclass_name : access_mode superclass_name

Access Mode is used to specify, the mode in which the properties of super class will be inherited into subclass, public, private or protected.

Example of Inheritance:



Example:

```
class Animal
{
public:
int legs = 4;
};
class Dog : public Animal
{
public:
int tail = 1;
};
int main()
{
Dog d;
cout << d.legs;
cout << d.tail;
}
```

Output:

4 1

Inheritance Visibility Mode

We can inherit a class in 3 ways. It can either be private, protected or public.

Public Inheritance:

This is the most widely used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : public Superclass

Private Inheritance:

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // By default its private inheritance

Protected Inheritance:

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : protected Superclass

Table showing all the Visibility Modes

	Derived Class	Derived Class	Derived Class
Base class	Public Mode	Private Mode	Protected Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Types of Inheritance:

In C++, we have 5 different types of Inheritance. Namely,

Single Inheritance

Multiple Inheritance

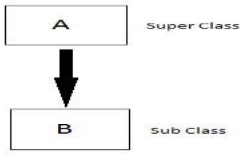
Hierarchical Inheritance

Multilevel Inheritance

Hybrid Inheritance (also known as Virtual Inheritance)

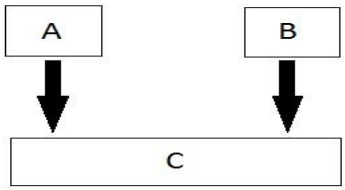
Single Inheritance:

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



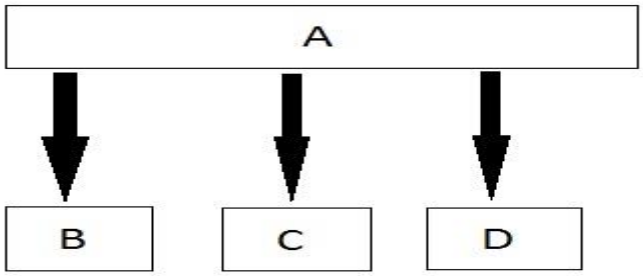
Multiple Inheritance:

In this type of inheritance a single derived class may inherit from two or more than two base classes.



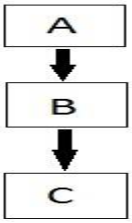
Hierarchical Inheritance:

In this type of inheritance, multiple derived classes inherits from a single base class.



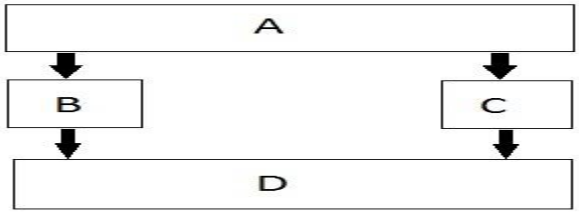
Multilevel Inheritance:

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid (Virtual) Inheritance:

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



UNIT-IV

POLYMORPHISM, VIRTUAL FUNCTIONS, STREAMS AND FILES

Polymorphism:

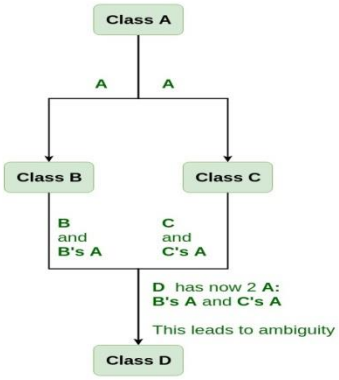
Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration but different definition.

Virtual Classes:

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.

Need for Virtual Base Classes:

Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a 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.

Pointer to derived class:

Pointer to Derived Class Object. In C++ you can declare a pointer that contains the address of the object of type class. ptr = &D1; // assign address of derive class object to base class pointer.

Virtual Functions:

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rules for Virtual Functions:

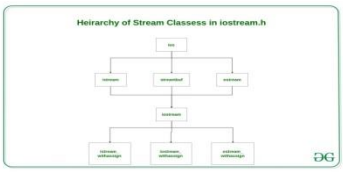
- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

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.

Stream Classes:

In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations. All these classes are defined in the file iostream.h. Figure given below shows the hierarchy of these classes.



ios class is topmost class in the stream classes hierarchy. It is the base class for istream, ostream, and streambuf class.

istream and ostream serves the base classes for iostream class. The class istream is used for input and ostream for the output.

Class ios is indirectly inherited to iostream class using istream and ostream. To avoid the duplicity of data and member functions of ios class, it is declared as virtual base class when inheriting in istream and ostream as

The _withassign classes are provided with extra functionality for the assignment operations that’s why _withassign classes.

The ios class: The ios class is responsible for providing all input and output facilities to all other stream classes.

The istream class: This class is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as get, getline, read, ignore, putback etc..

The ostream class: This class is responsible for handling output stream. It provides number of function for handling chars, strings and objects such as write, put etc..

The iostream: This class is responsible for handling both input and output stream as both istream class and istream class is inherited into it. It provides function of both istream class and istream class for handling chars, strings and objects such as get, getline, read, ignore, putback, put, write etc..

istream_withassign class: This class is variant of istream that allows object assigment. The predefined object cin is an object of this class and thus may be reassigned at run time to a different istream object.

ostream_withassign class: This class is variant of ostream that allows object assigment. The predefined objects cout, cerr, clog are objects of this class and thus may be reassigned at run time to a different ostream object.

Unformatted and Formatted I/O Operations:

Unformatted Console I/O Operations:

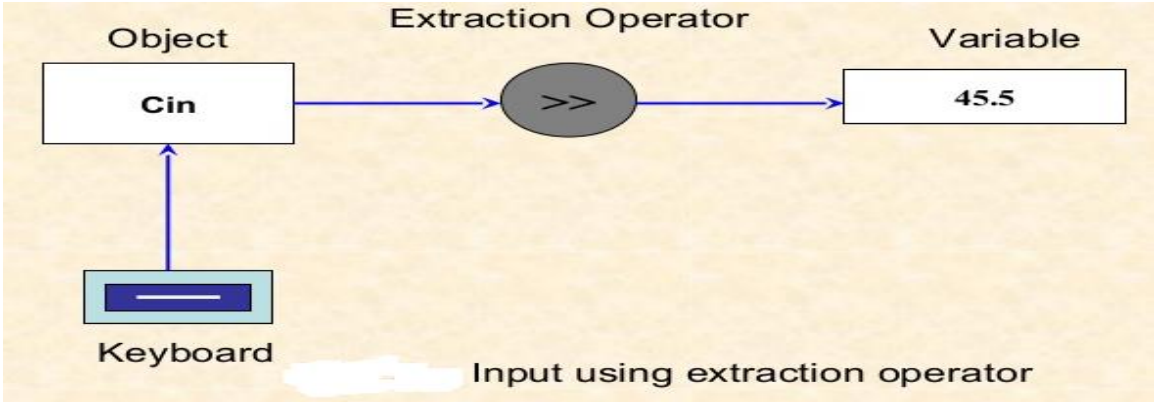
INPUT STREAM: the input stream does read operation through keyboard. It uses cin as object. The cin statement uses >> (extraction operator) before variable name. The cin statement is used to read data through the input device. Its syntax and example are as follows.

Syntax:

```
cin>>variable;
```

Example:

```
int v1;
float v2;
char v3;
cin>>v1>>v2>>v3;
```



Unformatted input functions with cin object

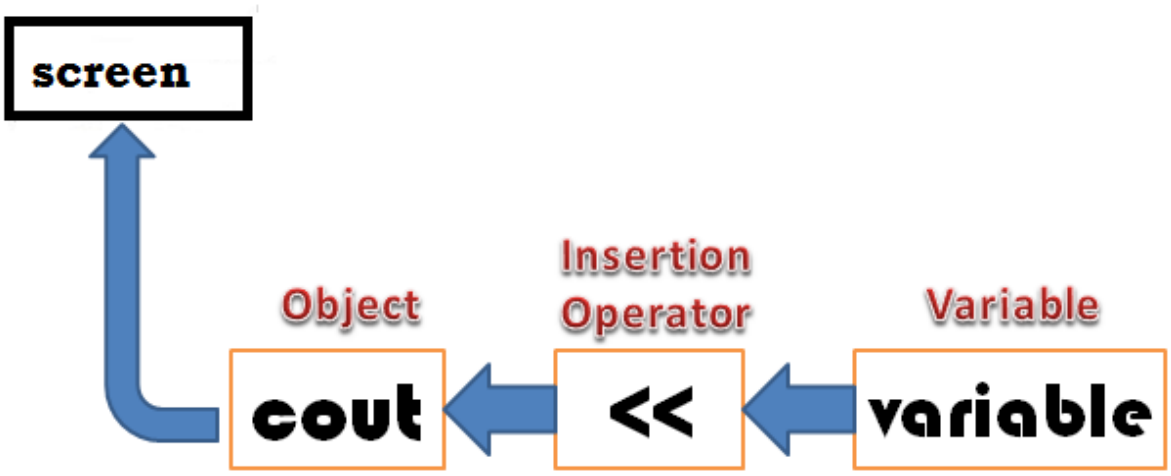
OUTPUT STREAM: The output stream manages output of the stream i.e., it displays contents of variables on the screen. It uses << insertion operator before variable name. It uses the cout object to perform console write operation. The syntax and example of cout statements are as follows:

Syntax:

Example

```
cout<<variable;
```

cout<<v1<<v2<<v3……<<vn;



Output operation with cout

get() : This function is used to read one character. It also accepts space, new line, tab character as input.

Syntax

cin.get(variable);

example

cin.get(ch);

getline(): This function is used to get input until the user press the return key.

Syntax

cin.getline(variable, size);

example

cin.getline(x,30);

read(): This function is used to get the text from the keyboard. Here it is necessary to enter characters equal to the number of size specified.

Syntax

cin.read(variable, size);

example

cin.read(x,30);

Unformated Output Functions with Cout Object:

Put (): this function is used to display one character on the screen.

Syntax

cout.put(variable);

example

cout.put(ch);

write (): This function is used to display the string on the screen.

Syntax

cout.write(variable, size);

example

cout.write(x,30);

Formatted Console I/O Operations:

The formatted functions with cout object are given below:

Function	Purpose
Width()	To set the required field width. The output will be displayed in given width
Precision()	To set number of digits after decimal point for a float value.
fill()	To set a character to fill in the blank space of a field
setf()	to set various flags for formatting output.

Manipulators:

Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

There are various types of manipulators:

Manipulators without arguments: The most important manipulators defined by the IOStream library are provided below.

endl: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.

ws: It is defined in istream and is used to ignore the whitespaces in the string sequence.

ends: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std:: ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.

flush: It is also defined in ostream and it flushes the output stream i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same but may not appear in real-time.

UNIT-V

EXCEPTION HANDLING AND DATA STRUCTURES IN C++

Exception Handling:

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

throw: A program throws an exception when a problem shows up. This is done using a throw keyword.

catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
// protected code
}catch( ExceptionName e1 )
{
// catch block
}catch( ExceptionName e2 )
{
// catch block
}catch( ExceptionName eN )
{
// catch block
}
```

We can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using throw statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b) {
if( b == 0 ) {
```

```
throw "Division by zero condition!";
}
return (a/b);
}
```

Catching Exceptions:

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
// protected code
}catch( ExceptionName e ) {
// code to handle ExceptionName exception
}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try {
// protected code
}catch(...) {
// code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

```
#include <iostream>
using namespace std;
double division(int a, int b) {
if( b == 0 ) {
throw "Division by zero condition!";
}
return (a/b);
}
int main () {
int x = 50;
int y = 0;
double z = 0;
try {
z = division(x, y);
cout << z << endl;
}catch (const char* msg) {
```

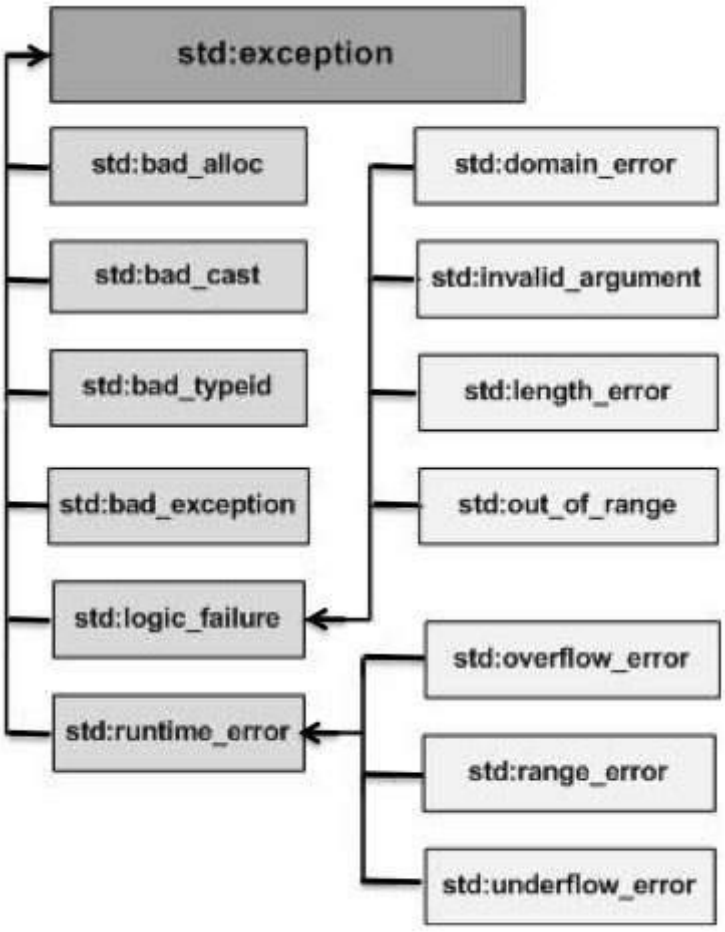
```
cerr << msg << endl;
}
return 0;
}
```

Because we are raising an exception of type const char *, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

```
Division by zero condition!
```

C++ Standard Exceptions:

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by new.
std::bad_cast	This can be thrown by dynamic_cast.

std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by typeid.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Define New Exceptions:

We can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how we can use std::exception class to implement our own exception in standard way:

```
#include <iostream>
#include <exception>
using namespace std;
struct MyException : public exception {
const char * what () const throw () {
return "C++ Exception";
}
};
int main() {
try {
throw MyException();
}catch(MyException& e) {
std::cout << "MyException caught" << std::endl;
std::cout << e.what() << std::endl;
} catch(std::exception& e) {
//Other errors
}
}
```

This would produce the following result:

```
My Exception caught
```

C++ Exception

Here, what () is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Object Oriented Exception Handling with Classes:

The C++ standard library provides a base class specially designed to declare objects to be thrown exceptions. It is called std::exception and is defined in the <exception> header. This class has a virtual member function called what () that returns a null – terminated character sequence and that can be overwritten in derived classes to contain some sort of description of the exception.

Multiple Exceptions:

A single try statement can have multiple catch statements. Execution of particular catch block depends on the type of exception thrown keyword. If throw keyword send exception of integer type, catch block with integer parameter will get execute.

Catch All Exceptions:

The above example will catch only three types of exceptions that are integer, character and double. If an exception occurs of long type, no catch block will get execute and abnormal program termination will occur. To avoid this, we can catch statement with three dots as parameter (…) so that it can handle all types of exceptions.

Extracting Data from the Exception class:

The C++ standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called std::exception and is defined in the <exception> header. This class has a virtual member function called what() that returns a null-terminated character sequence (of type char*) and we have not overwritten in the derived class to get the exception class implementation i.e. data from the base class. The base class implementation the what() function returns string “std::exception”, which is displayed as the output.

Re-Throwing as Exception:

Re-throwing exception is possible, where we have an inner and outer try-catch statements (nested try-catch). An exception to be thrown from inner catch block to outer catch block is called re-throwing exception.

Handling the bad-alloc Exception:

A typical example where standard exceptions need to be checked for is on memory allocation. The bad-alloc exception is thrown by new operator on memory allocation failure.

Data Structure:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

Basic Terminology:

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures:

As applications are getting complexes and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store; If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

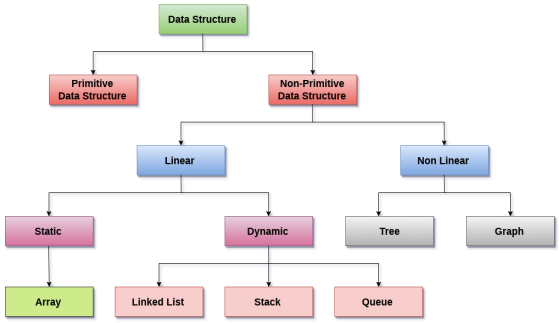
Advantages of Data Structures:

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification:



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have almost one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Operations on data structure

Traversing: Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

Insertion: Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

Deletion: The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

Linked List:

- A linked list is a sequence of data structures, which are connected together via links.
- Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

Link: Each link of a linked list can store a data called an element.

Next: Each link of a linked list contains a link to the next link called Next.

LinkedList: A Linked List contains the connection link to the first link called First.

Linked List Representation:

Linked list can be visualized as a chain of nodes, where every node points to the next node.

As per the above illustration, following are the important points to be considered.

Linked List contains a link element called first.

Each link carries a data field(s) and a link field called next.

Each link is linked with its next link using its next link.

Last link carries a link as null to mark the end of the list.

Types of Linked List:

Following are the various types of linked list.

Simple Linked List: Item navigation is forward only.

Doubly Linked List: Items can be navigated forward and backward.

Circular Linked List: Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations:

Following are the basic operations supported by a list.

Insertion: Adds an element at the beginning of the list.

Deletion: Deletes an element at the beginning of the list.

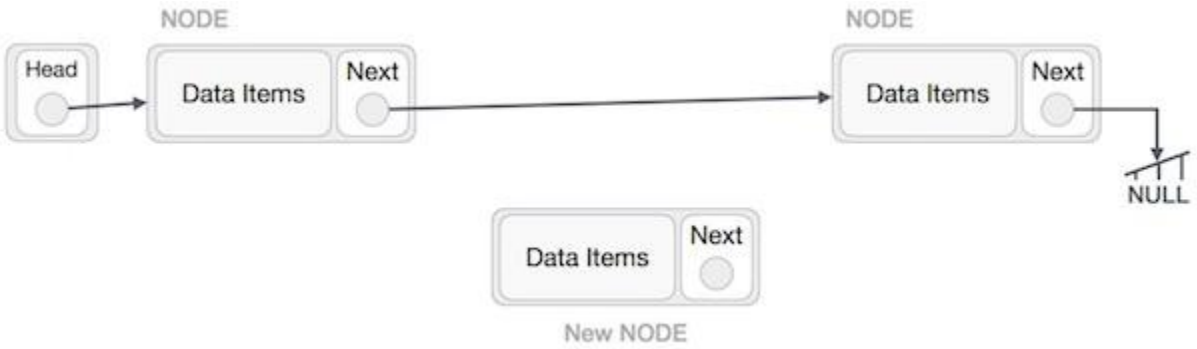
Display: Displays the complete list.

Search: Searches an element using the given key.

Delete: Deletes an element using the given key.

Insertion Operation:

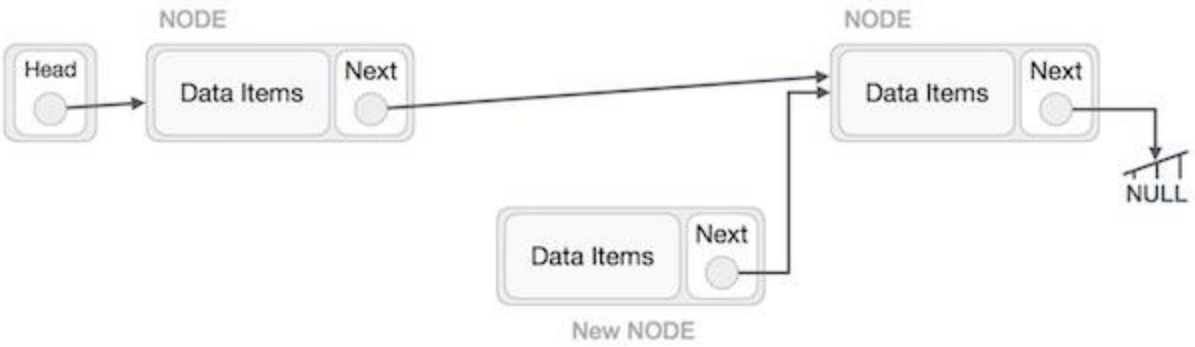
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

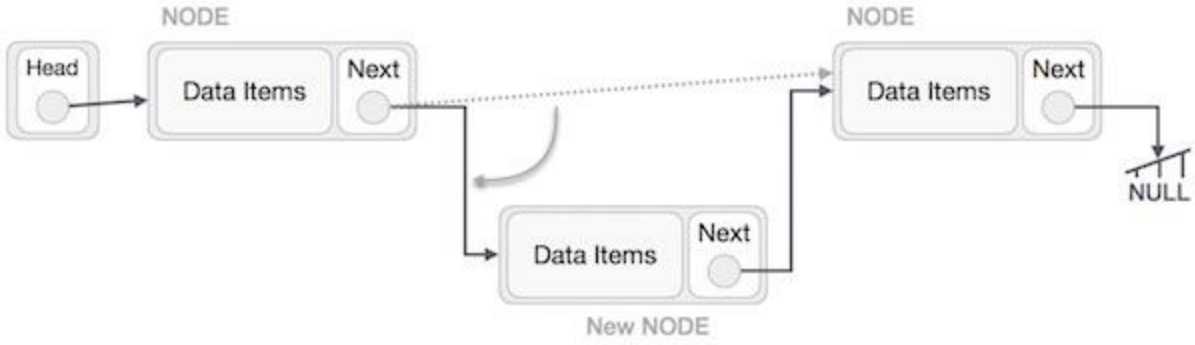
NewNode.next −> RightNode;

It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

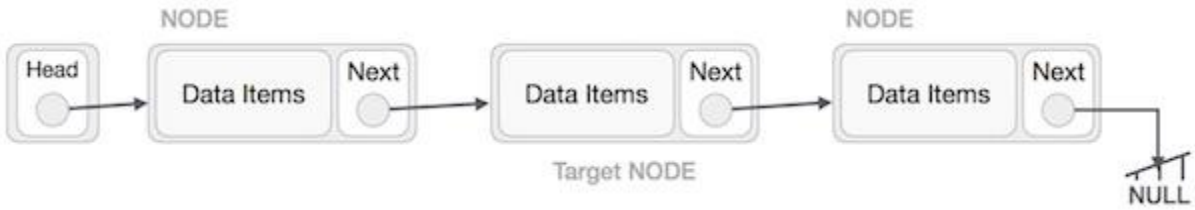


This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

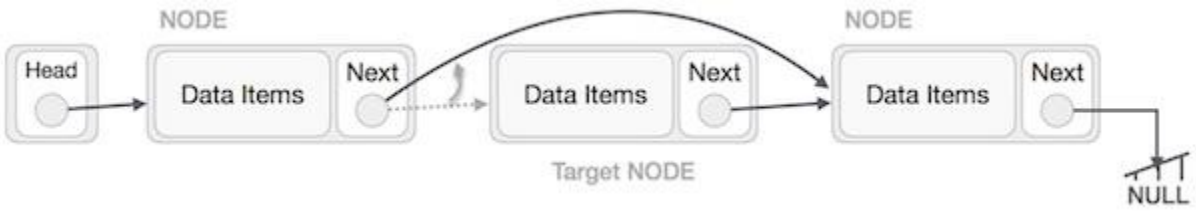
Deletion Operation:

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



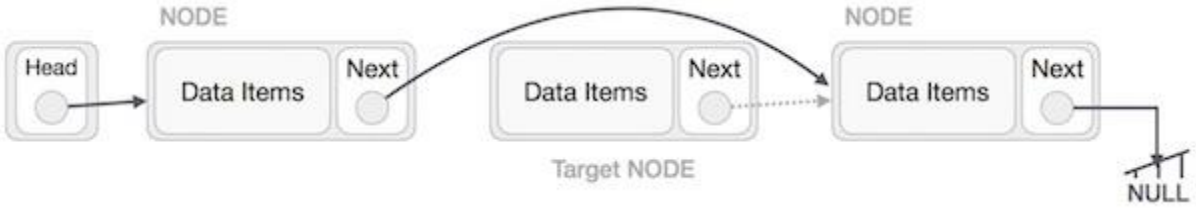
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;

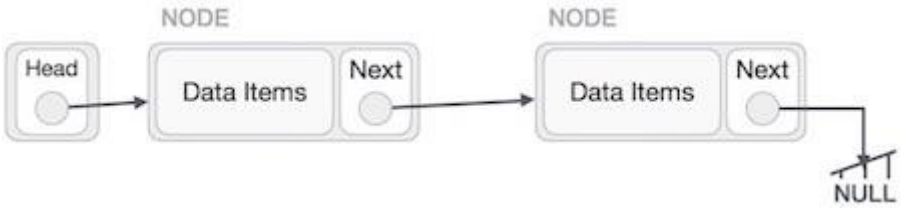


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

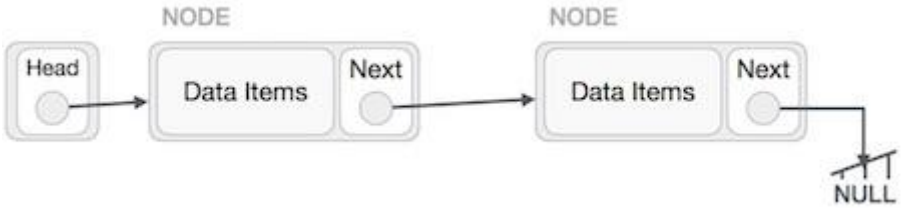


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

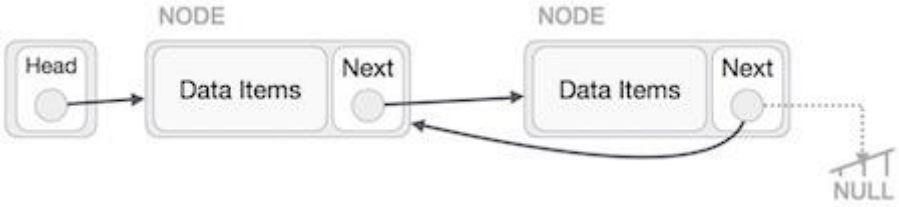


Reverse Operation:

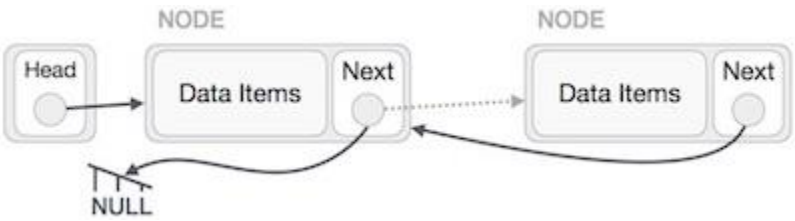
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



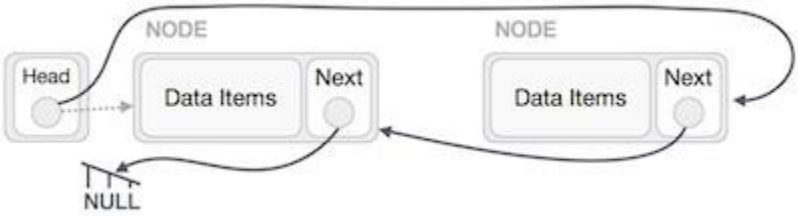
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



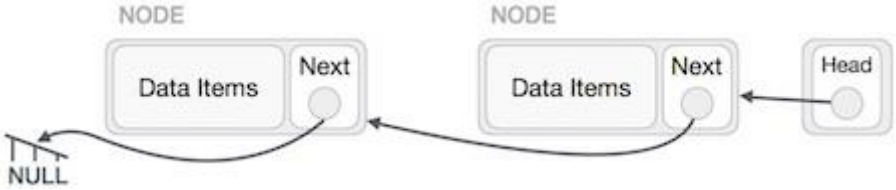
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.

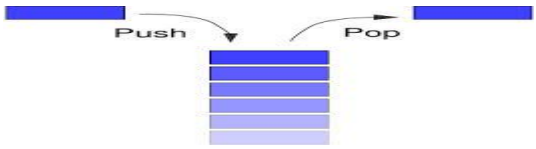


We'll make the head node point to the new first node by using the temp node.



Stack:

In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



Queue:

An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access. 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.