EC 560: Data Structures and Algorithms

Course code	Course title	Hours/week			Credits	CIE Marks	SEE Marks	Total Marks
		L	T	P				
EC 560	Data Structures and Algorithms	3	0	0	3	50	50	100

Overview of C++:

- C++ is an extension of C programming language.
- C++ was first invented by ‘Bjarne Stroustrup’ in 1979 at Bell Laboratories, USA.
- Initial name of C++ was **“C with Classes”,** renamed as C++ in 1983 by ‘Rick Mascitti’
- The invention of C++was necessitated by major programming factor: increasing complexity.
- C program is so complex that is difficult to grasp as a totality if the program exceeds from 25000 to 100000 lines of code. C++ overcomes this problem.
- C++ allows us to comprehend and manage larger, more complex programs.

- Most additions made to C, supports Object-Oriented Programming (OOP).
- Some features of OOP were inspired by another programming language Simula67.
- C++ is blending of two powerful languages C and Simula67.
- First revision of C++ was made in the year 1985.
- Second revision of C++ was made in the year 1990.
- The first draft of the proposed standard C++ was jointly created by ANSI and ISO on January 25th 1994.
- The second draft of the proposed standard C++ was on November 11th 1997.
- Final standard C++ became reality in 1998
- C++ contains many advanced and new features along with the C features.

Characteristics of OOP

- Emphasis is on data rather than procedure
- Programs are divided into objects
- Data can be hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up programming approach.

Application of OOPs

- Real-time systems and online application systems such as Air traffic control, Airline seat reservation, Railway ticket reservation, Bank transactions etc.,
- Simulation and modeling of vehicle design and performance in motor industry.
- Object-oriented database design and applications.
- Multimedia, Hypertext, hypermedia applications and other internet applications.
- Neural networks and parallel programming
- Office automation and data processing systems
- CIM / CAM / CAD systems.

Benefits of OOPs

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.

- Through inheritance, we can increase reusability of code.

- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. 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 an 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 form.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.
- Object-oriented system can be easily upgraded from small to large systems.

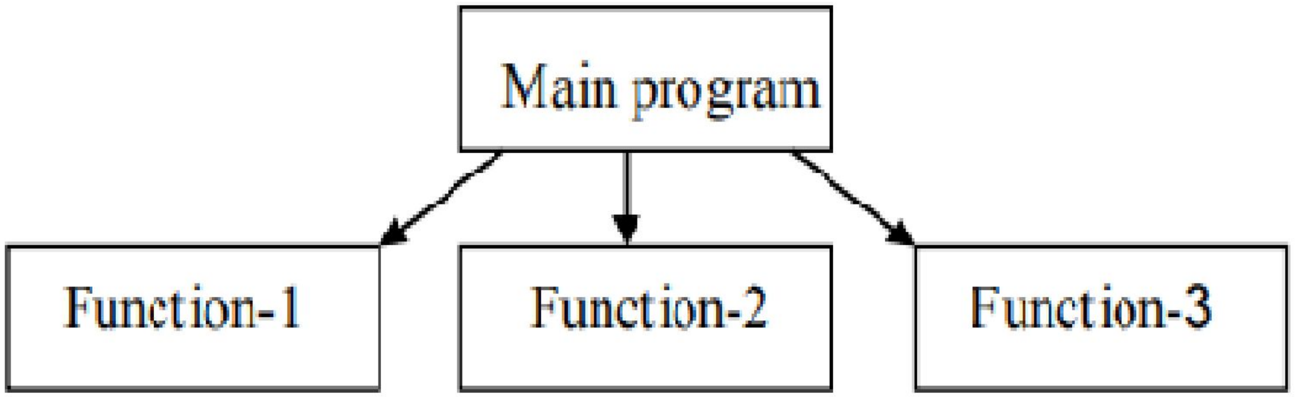
Limitations of OOPs

- Data is not freely available for all functions
- Data cannot move around the program or system to satisfy the requirement of individual function.
- Defined action can only be implemented on associated data members. Instantaneously, it is not possible to define random actions.
- Sometimes one may not feel it convenient to write programs if data members and functions could not be separated out.

- For every new operation, user or programmer must define a member function and appropriate changes must reflect in class definition.
- OOP is thus complex and tedious if one does not know the design features of his problem to be solved interns of objects and its methods.

Procedure Oriented Programming Language

- In the procedure oriented approach, the problem is viewed as sequence of things to be done such as reading , calculation and printing.
- Procedure oriented programming basically consist of writing a list of instruction or actions for the computer to follow and organizing these instruction into groups known as functions.



Procedure Oriented Programming

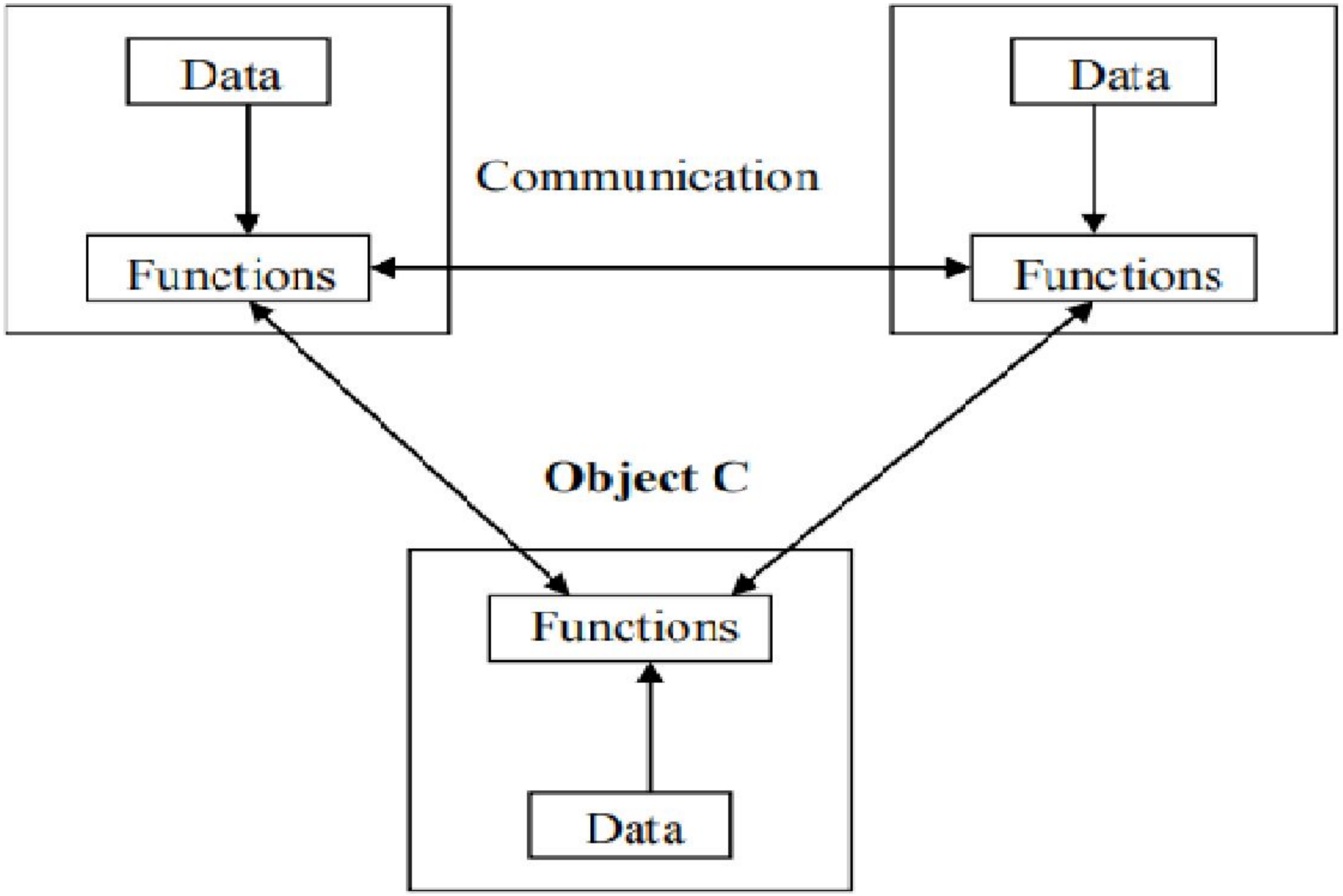
The disadvantage of the procedure oriented programming languages is:

1. Global data access
2. It does not model real word problem very well
3. No data hiding

Object Oriented Programing

- Object oriented programming as an approach that provides a 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.

Object Oriented Programing



Organization of data and functions in OOP

Object Oriented Programming (OOP)

- OOP is a powerful way to approach the job of programming. OOP was created to help programmers break through the several C barriers.
- To support the principles of OOP, all OOP languages have three traits in common:

  Encapsulation

  Polymorphism

  Inheritance

Basic Concepts of OOPs

It is necessary to understand the basic concepts used extensively in object-oriented programming.

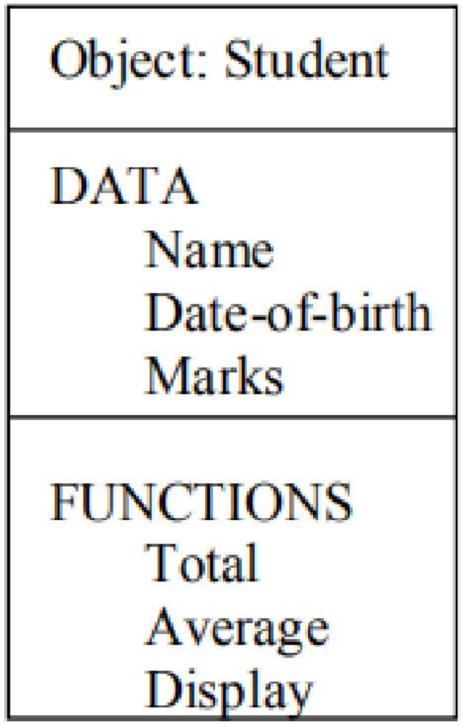
These include:

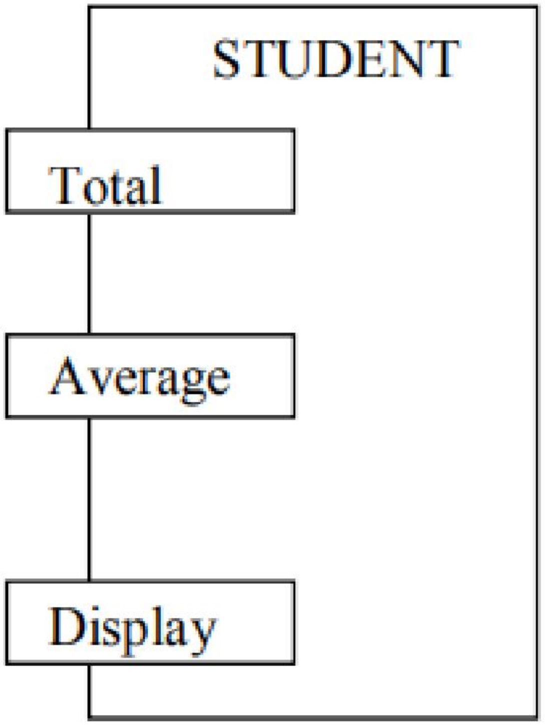
- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

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.
- Program objects should be chosen such that they match closely with the real-world objects.
- The fundamental idea behind object oriented approach is to combine both data and function into a single unit and these units are called objects.

Objects





Two Ways to represent Object

Classes

- A class is thus a collection of objects of similar type. A class is a specification describing a new data form and an object in a particular data structure constructed according to the requirement.
- The entire set of data and a code of an object can be made a user defined data type with the help of a class.
- In fact, objects are variables of the type 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 type class with which they are created.

Classes

- A class is a user defined data type consisting of private and public data members and member functions in a single unit.
- It is the fundamental building block of object oriented program.
- A class definition contains two parts:

i) class head

ii) class body

- The **class head** is made up of the keyword class followed by the name of the class.
- The **class body** is the portion of the class definition enclosed within a pair of curly braces. The class body consists of both data and functions.
- The data items used in a class are called **data members** and the functions defined are called as **member functions**.
- The class definition is terminated by a semicolon (;)

The general form of a class

```
class class_name
{
   private:                     // Not Mandatory, Default
    Variable declarations;          is private
    Function declarations;     //Normally it is not be
                available in program
    public:
    Variable declarations;       //Normally it is not be
Function declarations;         available in program
  };
```

Sample C++ Program

```
#include<iostream.h>
 class Sample
  {
        public:
        void display ( )
      {
        cout << " This is my first program in C++ \n ";
       }
  } ;

 void main ( )
 {
        Sample S;
        S.display ( );
        getch( );
 }
```

Output of the program :

```
This is my first program in C++
```

In this example program,

iostream.h - is an header file in C++ like stdio.h in C

class - is a keyword

Sample - is a name of the class (It is user defined)

public - is a keyword (which is a access specifier/visibility mode)

cout -is a keyword used for printing (Same as printf)

<< - is an insertion (output) operator. Can take any data type like C.

}; - Indicates end of the class.

s - is an object (User Defined) of the class Sample

s.display( ) - is the way of access the member function in C++.

Private and Public: (Data Hiding)

- A key feature of object-oriented programming is data hiding means, the data is concealed within a class.
- The primary mechanism for hiding data is to put it in a class and make it private. The private data and functions can only be accessed within the class.
- Public data or functions on the other hand are accessible from outside the class. The keyword private and public are known as **visibility labels.** These keywords are followed by a colon.

- The use of the keyword private is optional. By default, the data members of a class are private.
- If both the labels are missing, then, by default, all the members are private.
- Such a class is completely hidden from the outside world and does not serve any purpose.

```
class student
{
    private:

     int regno;
     char name[10];

     public:

     void getdata()
     {
     cout << "Enter the register no. and Name\n";
     cin >> regno>>”\n”>>name;
     }

     void putdata()
     {
      cout << "Register No.: " << regno << “\n”;
      cout << "Name : " << name << “\n”;
     }
};
   void main()

      { student S;

           S.getdata();

           S.putdata();
```

Output of the program

```
>>Enter the register no. and Name
>>3456
>>Pavithra D R

Register No.: 3456
Name: Pavithra D R
```

- We usually give a class some meaningful name, such as student.
- This name now becomes a new type identifier that can be used to declare instances of that class type. The class item contains two data members and two function members.
- The function getdata( ) can be used to read reg no and name , and putdata( ) for displaying reg no and name .
- The data members are usually declared as **private** and the member functions as **public**.

Creating Objects

- Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable).
- For example,

```
  student s;            //memory for the object s is created
```

  creates a variable s of type student.
- In C++, the class variables are known as objects. Therefore, s is called an object of type student
- We may also declare more than one object in one statement.
- For example

```
      student s1, s2, s3;
```

- The declaration of an object is similar to that of a variable of any basic type.
- The necessary memory space is allocated to an object at this stage.
- Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures.
- That is to say, the definition

```
class student
 {

    ………
  } s1, s2, s3;
```

would create the object s1, s2 and s3 of type student.

Accessing Members

- A member function of a class can be called by an object of that class followed by a dot operator followed by member function name.

For example,

- s.getdata( ); // Here, s is an object, and getdata is a member function of the class student
- s.putdata( ); // Here, s is an object, and putdata is a member function of the class student

Defining Member Functions

Member functions can be defined in two places:

1. Inside the class definition

2. Outside the class definition

- It is obvious that, irrespective of the place of definition, the function should perform the same task.
- Therefore, the code for the function body would be identical in both the cases. But there is a subtle difference in the way the function header is defined.

Inside the class definition

In this method defining a member function is to replace the function declaration by the actual function definition inside the class. For example we could define the student class as follows:

```
class student
    {
        int regno;
        float percentage;
        public:
        void getdata()
          {   cout << "Enter the register no. and Percentage \n";
            cin >> regno >> percentage;
          }
        void putdata()
        { cout << "Reg No.: "<< regno << " \nPercetage : " << percentage;
        }
    };
```

- When a function is defined inside a class, it is treated as an **inline function**.
- Therefore, all the restrictions and limitations that apply to an **inline function** are also applicable here.

Outside the class definition

- Member functions that are declared inside a class have to be defined separately outside the class.
- Their definitions are very much like the normal functions. They should have a function header and a function body.
- An important difference between a member function and a normal function is that a member function incorporates a membership ‘identity label’ in the header. This ‘label’ tells the compiler which class the function belongs to.

- The general form of a member function definition is:

return-type class name **::** function-name (argument declaration)

```
        {
        function body
        }
```

- The membership label class-name with :: tells the compiler that the function **function-name** belongs to the class **class-name**.
- That is, the scope of the function is restricted to the class name specified in the header line.
- The symbol :: is called the **scope resolution operator**.

- For instance, consider the member functions getdata( ) and putdata( ) of class student

They can be coded as follows:

```
 void student :: getdata( )
 { cout << "Enter the register no.and Percentage \n";
   cin >> regno << percentage;
  }
void student :: putdata( )
{
cout << "Reg No.: "<< regno << " \nPercetage : "<< percentage;
}
```

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 is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called **data hiding or information hiding.**

Abstraction

- **Abstraction** refers to the act of representing essential features without including the background details of explanations.
- Classes use the concept of abstraction and are defined as a list of abstract attributes like size, weight, cost, and functions to operate on these attributes.
- They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called methods or member functions.
- Since the classes use the concept of data abstraction, they are known **as Abstract Data Types (ADT).**

Inheritance

- Inheritance is a process of acquiring (copying) the properties from one class/object to another class/object. In OOP, the concept of inheritance provides the idea of reusability.
- This means that we can add additional features to an existing class without modifying it.
- The new class is called **child class (derived class).** The old class is called **parent class (base class).**
- The child class defines only the additional features and will have all the features of parent class.

- Once a class has been written and tested, it can be adapted by other programmers to suit their requirements.
- This is basically done by creating new classes and reusing the properties of the existing ones. Reuse of code gives saving memory and time of writing code; reduce the length of the program.

Base class and Derived class

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived_class_name : visibility-mode base-class-name
 {
   …….
 };
```

where,

class - is a keyword

derived_class_name - name of the derived class

: shows the derivation from the base-class

visibility mode - specifies the types of derivation

base-class-name - name of the base class

Example:

```
class A
{
            int x, y;
public:
void setvalue (int a, int b)
{
x = a; y = b;
}
void display ( )
{
cout << x << " " << y << "\n" ;
}
};
class B : public A
{
            int z;
            public:
            assign (int k)
        {
            z = k;
       }
            void displayz ( )
             {
            cout << z << "\n" ;
            }
};
void main( )
{       B dobj(5);
        dobj.setvalue(10, 20);
        dobj.display( );
        dobj.displayz( );
        getch( );
}
```

- Output of the program :

```
10   20
  5
```

Inheritance and its types

- A derived class extends its features by inheriting some or all the properties from its base class and adding new features of its own.
- While inheriting, the derived class can share properties from only one class, more than one class, and more than one level.
- Based on the relationship, inheritance can be classified into five forms.

1. Simple or single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Single Inheritance

- Deriving a single new class from a single base class is called simple or single inheritance. The single inheritance possesses one-to-one relation.

```
class A
{ public: int x,y;
      …….
};
class B : public A
{
      public: int a,b;
      ……
};
//Here, public members of baseclass are become public members of
derivedclass
```

2. **Multiple Inheritance**

- Multiple inheritance is a mechanism in which a new class is derived from more than one class. It follows many-to-one relation.

```
class A
{ public: int x,y;
……
};
class B
{ public: int a,b;
 ……
};
class C
{ public: int m,n;
 ……
};
class D : public A, public B, public C
{
……
};
//Here, public members of all the 3 baseclasses are become public members of D.
```

3. Multilevel inheritance

- Multilevel inheritance is a mechanism in which we derive a class from another derived class. It’s just like grandfather-father-son relationship for exhibiting multilevel inheritance.
- Class A serves as base class for the derived class B, which in turn serves as a base class for the derived class C.
- Class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

Example:-

```
class A
{ public: int x,y;
 ……
};
class B : public A
{ public: int p, q;
………………
};
class C : public B
{
…………..
};
```

// Here, public members of calss A (i.e., x, y) become public member class B and finally public members class B (i.e., p, q, x, and y) become public members of derivedclass C.

4. **Hierarchical Inheritance**

- Hierarchical inheritance is a mechanism in which the features of one class may be inherited by more than one class.
- These classes, in turn, may be inherited. In the hierarchical inheritance, there will be only one base class from which several other classes are created.
- This form of inheritance follows one-to-many relationship.
- A derived class can be created by deriving the properties of base class and adding its own properties.

```
class A
{ public: int x, y;          // parent class
……
};
class B : public A                    //child or derived class
{ public: int p, q;
……
};
class C : public A                     //child or derived class
{ public: int m, n;
……
};
class D : public A                    //child or derived class
{ public: int i, j;
……
};
//Here, public members of A (i.e., x, y) become public member B, C, D
```

5. Hybrid Inheritance

- Hybrid inheritance is a mechanism in which the program design would require two or more forms of inheritance.
- Here, the derived class may be created from the multilevel and multiple classes. Or, it could be created from the hierarchical and multiple classes.

```
class A
{ public: int x, y;
……
};
class B : public A
{ public: int a, b;
……
};
class C : public A
{ public: int p, q;
…… };
class D : public B, public C
{
public: int m, n;
……
};
```

Polymorphism

- Polymorphism is an ability to take more than one form. An operation may exhibit different behaviors in different instances.
- The behaviors depend upon the types of data used in operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum.
- If the operands are strings, then the operation would produce a third string by concatenation.
- The process of making a function to exhibit different behaviors in different instances is known as**function overloading (function polymorphism).**

- In other words, the same function name with different types of parameters are used to perform various task is called as **function overloading.**
- The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading** (**operator polymorphism).**

For Example **(FUNCTION OVERLOADING:) c++ PROGRAM**

```
void main ( )
{ int a, b, sum;
float x, y, total;
double m, n, gross;
Cin>>a>>b>>x>>y>>m>>n;
sum = add(a, b);                    // Function name is add( )
total = add(x, y);                  // Function name is add( )
gross = add(m, n);                  // Function name is add( )
cout<<< gross <<"\n";
}
void add(int a, int b)              // Function name is add( )
{
return(a+b);
}
void add(float x, float y)          // Function name is add( )
{
return(x+y);
}
void add(double m, double n)              // Function name is add( )
 {
return(m+n);
}
```

- In the above program, contains three user defined functions in the name add( ) but in all the three functions, type of parameters used are dissimilar (i.e., a and b are integer, x and y are float, m and n are double datatype).

- A same function name used for multiple purposes with dissimilar type of parameter is called as **function overloading**

Operator Overloading

- In C++, we can make operators to work for user defined classes.
- This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
- For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +.
- Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

The general form of an operator function is:

return-type **class** class-name **::** operator op(arg list)

{

Function body**;**   //any task defined

};

Where

return type - is the type of value returned by the specified operation

op – is the operator being overloaded

operator op- name of the function

Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding 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.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other.

The process of programming in an object-oriented language, therefore, involves the following basic steps:

a) Creating classes that define objects and their behavior.

b) Creating objects from class definitions, and

c) Establishing communication among objects.

- A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result.
- Message passing involves specifying the name of the function (message) and the information to be sent.

Declaration of variables

- There are two types of variables declared in functions.

  i) Local variable   ii) Global variable

i) Local variable

- Variables declared within functions are called local variables. The scope of a local variable is confined to the function; in particular its value and meaning are well defined within the body of that function.
- These local variables cannot be accessed outside that function. But they can be redefined in some other function can locally redefine an identifier or variable defined outside the function.
- Thus the scope of a local variable is limited to the extent of the body of the function in which it is declared.

- **Example for Local variables:**

```
void main()
{
int i=3;
cout << "Before Function Call: i=" << i << “\n”;
modify ();
cout << "After Function Call: i=" << i << “\n”;
}
void modify()
{
int i=5;
}
```

OUT PUT:

```
Before Function Call: i=3
After Function Call: i=5
```

Global variables

- A global variable is declared outside of any functions. Hence they do not belong to any particular function.
- Scope of a global variable specify that the extent of the meaning of that variables is throughout the program and its meaning is not at all local to any particular function.
- Therefore any function in a C program can access the value and meaning of the global variable and these functions can change the value of global variable.

Example for Global variables:

```
a=30, b=20;
void main()
{
 cout << "The value of a and b in Main function \n" ;
 cout << "a = " << a << “\n” << "b = " << b << “\n”;
global( );
}
void global()
{
cout << "The value of a and b in function global( ) \n";
cout << "a = " << a << “\n” << "b = " << b << “\n”;
}
```

OUT PUT :

```
The value of a and b in function Main( )
a=30
b=20
The value of a and b in function global( )
a=30
b=20
```

Dynamic Initialization of Variables

- The process of initializing variable at the time of its declaration at run time is known as dynamic initialization of variable.
- Thus in dynamic initialization of variable a variable is assigned value at run time at the time of its declaration.

- Example for dynamic initialization of variables

```
void main( )
{
int a;
cout << “Enter Value of a”;
cin >> a;
int cube = a * a * a;
}
```

In above example variable cube is initialized at run time using expression a * a * a at the time of its declaration

Reference Variables

- Reference Variables (or references in short), is an alias, or an alternate name to an existing variable, that we can use to read or modify the original data stored in that variable.
- When we declare a reference and assign it a variable, it will allow us to treat the reference exactly as though it were the original variable for the purpose of accessing and modifying the value of the original variable even if the second name (the reference) is located within a different scope.
- Declaring a variable as a reference rather than a normal variable simply entails appending an ampersand to the type name.

Syntax :

Datatype& reference_variable_name = original_variable_name ;

Example:

int X;

int& foo = X;   // Here, X is an Original
    Variable and foo is a   reference
variable to X

foo = 56;

- In a way, this is similar to having a pointer that always points to the same thing.
- One key difference is that references do not require dereferencing like pointers do; we just treat them as normal variables.
- A second difference is that when we create a reference to a variable, we need not do anything special to get the memory address. The compiler figures this out for us.

Operators in C++

- C++ supports all the operators (Such as arithmatic, logical, relational, conditional, assignment, increment/decrement, bitwise operators, and special operators like comma operator, dot operator, arrow operator, * operator, & operator, and sizeof( ) operator) available in C Programming. Along with these operators C++ provides following operator.
- >> operator
- << operator
- cast operator
- Memory Management operator

Input Operator (>>)

- The symbol **>>** is called an extraction operator. The >> operator for accepting standard data types viz., int, float, char, double, long etc., The input operator >> sends the data on its right .

  For example, **cin >> radius;**

which causes the program execution to wait for the user or programmer to type in a number or numeric value as radius of a circle.

- Consider another example,

  cin >> a>>b>>c;

- The input statement **cin** prompts the user to type-in 3 integer numbers. If we enter the numbers 3 2 4 the first operator >> takes value 3 from its stream object cin and places it in the variable a that follows on its right.
- A second operator >> reads a value 2 and stores it in the variable b. Similarly third operator >> extracts a value 4 from its input stream on its left and copies it into third variable c.
- The multiple use of >> in statement is known as **cascading.**

Output Operator (<<)

- The output operator **<<** is called the **insertion or put to operator**. It inserts the contents of the variable on its right to the object on its left.
- We can also include variables with single cout whose contents are displayed simultaneously.
- For example, **cout << "SUM ="<<sum << "\n";** sends the string "SUM =" to cout and then sends the value of sum. Finally it sends the new line character.
- The multiple use of << in one statement is called **cascading.**
- When cascading an output operator, we should ensure necessary blank space between different items. The above cout statement can be written as two cout statements to give same output effect i.e.,

  cout << "SUM=";

  cout << sum;
- Using the cascading technique, the last two statements can be combined as follows:

cout << "SUM = " << sum << "\n" << "AVERAGE = " << average << "\n" ;

Memory Management Operator

- The two operators **new** and **delete** are the C++ mechanism that perform dynamic memory **allocation** and **deallocation** respectively.
- An advantage of using these operators involve the existence of a data object or value created by **new**, until it is explicitly destroyed by **delete** operator. Thus user has an explicit control over the allocation and deallocation of memory for variables of fundamental types, arrays, structures etc., Since these operators manipulate memory on the free store, they are also known as **free store operators.**

- The **new** operator with the pointer to an object allocates memory for that object and assigns the address of that memory to the pointer. But the delete operator does the reverse. i.e., it returns the memory occupied by the object back to the heap.
- The new operator can be used to create objects of any type.

Syntax:

pointer-variable **= new** data-type**;**

- Here, pointer-variable is a pointer of type data-type.
- The new operator allocates sufficient memory to hold a data object of type data-type and return the address of the object.
- The data-type may be any valid data type.
- The pointer_variable holds the address of the memory space allocated.
- **For example:**

int *p = new int;

float *q = new float;

- When a data object is no longer needed, it is destroyed to release the memory space for reuse.
- The general form of its use is:

  delete pointer-variable**;**
- Since heap is finite, it can become exhausted. If there is insufficient memory to fill an allocation request, then new will fail and a **bad_alloc** exception (generated by the header new) will be generated.

  Example: **delete** p**;**
- Finally it frees the dynamically allocated memory by the delete operator only with the valid pointer previously allocated by using the **new**.
- If sufficient memory is not available for allocation the **new** returns a **null pointer**

The **new** operator offers the following advantages over the function **malloc( )**.

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.

2. It automatically returns the correct pointer type, so that there is no need to use a type **cast**.

3. It is possible to initialise the object while creating the memory space.

4. Like any other operator, **new** and **delete** can be overloaded.

Program to illustrate memory management operators

```
#include<iostream.h>
#include<new.h>
void main()
{
        int *a = new int;
        int *b = new int;
        int *sum = new int;
        cout<<"Enter the values of a,b\n";
        cin>>*a>>*b;
        *sum=*a+*b;
        cout<<"Sum = "<<*sum<<endl;
        delete a;
        delete b;
        delete sum;
        getch();
}
```

```
OUTPUT:
Enter the values of a,b
4
6
Sum = 10
```

For array Allocation:

ptr_var = **new** array_type[size];

delete [] ptr_var;

Example:

Ptr = **new** int[5];

delete [] ptr;

The cast operator (Type Casting)

Converting one type of data item to another type using an operator named cast is called casting or type casting. Type casting in C++ is same as the type casting in C Programming. (Refer C programming for more about type casting).

Syntax:

(type) expression;

Or type (expression);

Example 1:

```
(float) x/2;          // Here x/2 evaluates to type float
```

Example 2:

```
float ( x/2 );        // Here also x/2 evaluates to type float
```

- Putting bracket either to datatype or to expression is called **casting**

Functions in C++

- A **function** is a self-contained block or a sub-program of one or more statements that perform a special task when called.
- Every program starts with user defined function **main( )**
- The **main( )** calls another function to share the work.
- C++ language supports two types of functions,

i) Library functions

ii) User defined functions

C++ Library Functions

- A library function is a built-in, pre-defined subprogram.
- It defines a series of such ready made functions, which are supposed to do the assigned work. Their task is limited.
- The user can only use the function but cannot change or modify them.
- For example,

  sqrt(x), **pow**(x, y), and **strcmp**(a, b) etc.

User Defined Functions

- In addition to the standard library functions supplied by the C++ system, the user can define a function according to his requirement are called as **user-defined functions**.
- Programmer can write his own function to perform a specific sub-task. Just as you define a **main( )** function.
- Function definition begins with a name and a set of statements enclosed in braces.
- The user can modify the function according to the requirement.

Actual arguments and Formal arguments

- These are used as a means for communication between user defined functions and the main( ) function.
- The variables to be passed to the functions are called **actual arguments,** they are enclosed in a parentheses following the functions name. They may have to either convey input values necessary for computations within the user-defined function or they may be used to return the computed results from user-defined functions to main( ) function.
- **The formal arguments,** which receives the value of the actual arguments or address of the actual arguments from the main program and send to the body of the user defined function for specified operation.

Return Statement

- A function can display the results after computation by including a **cout** statement within it.
- Sometimes user may wish that the results need no to be printed by the function itself, instead let the function return the result to its **main** function (calling **routine**).
- The general format of **return** statement is **return** (value);
- The **return** statement when executed from within a user-defined function returns the value. This value may be a constant, variable or expression that represents useful results of computation.
- This statement specifies that the function has to return a value to its calling function. Therefore if the function has to return a value to its calling routine, then the last statement executed in the function definition must be a **return** statement.

Function Declaration

- Function declaration means specifying the function as a variable depending on the return value. It is declared as the part of the main program. It helps the compiler to treat functions differently from other program elements.
- A function declaration has two principal components, one the **name of the function** and the **pair of parenthesis with or without arguments.** The arguments are called **formal arguments**, because they represent the names of the data items that are transferred into the function from the calling portion of the program.
- The **identifiers** used as formal arguments are 'local' in the sense that they are not recognized outside the function. Hence, the names of the arguments **<u>need not be the same as</u>** the names of the actual arguments in the calling portion of the program, however it must be the same data type

Write a program to add two numbers by user-defined function

```
int add(int x, int y);            // Function Prototype
void main()
{
int a, b, sum;
 cout << "Enter the values of a and b:\n";
 cin >> a >> b;
sum=add(a,b);              // Function Declaration or function call , a and b are
                    actual arguments
cout << "The sum of a and b = " << sum;
getch();
 }

int add(int x, int y)      // Function Definition, x and y are Formal Arguments
{
return(x + y);       //The return statement, returns sum of x+y to main function
}
```

In this program

a, b - Actual arguments or Arguments

x, y - Formal Arguments or Formal Parameters or Local Parameters

void main( ) - The calling function (It is a Built in Function)

int add( ) - The int is a return type of the return value. The add( ) is user defined Function name

int add(int x,int y) - The called function. (Function Definition)

return - It is a keyword to send the output of the function back to the Calling function.

{ - beginning of the main function or user defined function

} - end of the main function or user defined function

body of the function - all the statements placed between { and }

- **Note:** The number of the actual arguments should be equal to number of formal arguments.
- There must be one-to-one mapping between arguments i.e., they should be in the same order and should be in same data type.
- A user defined function can be defined in a C++ program, which may / may not have class and objects concepts in it. If a C++ program does not have class concepts in it, then, we can define a user defined function exactly similar to defining user defined function in C programming .
- If a C++ program contains a class concepts in it, then, we can define user defined functions either in inside the class definition and in outside the class definition

Argument Passing:

There are two ways in which we can pass arguments to the function.

i) Call by value

ii) Call by reference

Call by Value

- The value of actual arguments are passed to the formal arguments and the operation is done on the formal arguments.
- Any change made in the formal argument doesnot affect the actual arguments because formal arguments are photocopy of actual arguments.
- Changes made in the formal arguments are local to the block of the called function.
- Once control returns back to the calling function the changes made vanish.

Write a program to send values using call by value procedure

```
void main( )
{
 int x, y;
 cout << "Enter the values of x and y:\n";
 cin >> x >> y;
 change(x,y);
cout << "In main() x = ” << x << “ y = ” << y;
return 0;
 }
void change(int a, int b)
{
int temp;
temp = a;
a = b;
b= temp;
 cout << "In change() x =” << a << “y = ”<< b;
}
```

OUTPUT

```
Enter the values of x and y:
4
8
In change() x=8 y=4
In main() x=4 y=8
```

Note: In this program when the change( ) is called, the value of x and y are passed to a and b respectively. In such cases, the changes made inside the function cannot affect the main program.

Call by Reference

- The variables are stored somewhere in memory.
- Instead of passing the value of the variables, we can pass the address of the variable.
- Here, the formal arguments are pointers to the actual arguments. Hence they point to the actual argument.
- Changes made in the arguments are permanent.

Write a program to send values using call by reference procedure

```
void main()
{
 int x, y;
cout << "Enter the values of x and y:\n";
cin >> x >> y;
change(&x, &y);
cout << "In main( ) x = ” <<x<< “y = ” << y;
getch();
}
void change(int *a, int *b)
{
int *temp;
*temp = *a;
*a = *b;
*b= *temp;
 cout << “In change() x = ” << *a << “ y = ” << *b;
}
```

OUTPUT

```
Enter the values of x and y:
4
8
In change() x=8 y=4
In main() x=8 y=4
```

Note: In this program when the change( ) is called, the value of x and y are passed to a and b respectively. In such cases, the changes made inside the function affect the main program. This is called as the function **call by reference**

Recursive Functions

- In C++, it is possible for the functions to call themselves. Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

- Recursive functions are commonly used in such applications where the solution to the problem can be found in terms of successively applying the same solution (intermediate solution) to the same subsets (sub-tasks) of the problem.

- For example to compute the factorial of a given number n we write as

  fact = fact * i;   where i = 1 to n

In the recursive method to compute the factorial of a given number n as follows:

n! = n * (n-1)!;

or

fact(n) = n * fact(n-1)!;

Write a C++ Program to calculate factorial of entered number using recursive function

```
void main( )
{
int x, f;
cout << "Enter a number \n";
cin >> x ;
f = fact(x);
cout << "Factorial of” << x << “is” << f ;
}
int fact (int n)
{
int f = 1;
 if(n= =1)
return(1);
else
 f = n * fact (n-1);
return (f);
}
```

Inline functions

- One of the objectives of using functions in a program is to save memory space, when a function is likely to be called many times.
- However, every time a function is called, it takes a lot of time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.
- When a function is small, a substantial percentage of execution time may be spent in such overheads.
- An **inline function** is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code. Each time the actual code from the function is inserted into the program in place of function call instead of taking jump into the function.

Syntax:

```
inline function-header
{
function body
}
```

Example:

```
inline double cube(double a)
 {
return(a*a*a);
 }
```

It is easy to make a function **inline**. All we need to do is to prefix the keyword inline to the function definition. All **inline** functions must be defined before they are called.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch, or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

Inline expansion makes a program run faster because the overhead of a function call and return is eliminated.

Write a C++ Program to find the largest of three numbers using inline function

```
void main()
{
 int a, b, c;
 cout<<"Enter 3 numbers \n" ;
 cin>> a >> b >> c;
 large(a, b, c);
 getch();
}
inline void large(int x, int y, int z)
{
 int big;
if(x > y)
big = x;
else
big = y;
 if(z > big)
 big = z;
cout<< "The largest number is "<<big;
}
```

OUTPUT:

```
Enter 3 numbers
7
2
6
The largest number is 7
```

Friend functions

- The private members of a class cannot be accessed by non- member functions. But there may be some situations where it is needed to access the private members of a class by the non-member functions. This can be achieved by changing the access specifier private members to public.
- The attempt to change the access specifier of private members to public violates the concept of data hiding and encapsulation. So, we can overcome this problem by declaring the non-member functions as friend functions to the class.
- To make a non-member function “friendly” to a class, we have to simply declare this function as a friend of the class.

```
class ABC
{
public:
......
friend void xyz(void);      // friend function
          declaration
};
```

Important points to be observed while using 'friend functions':

- The function declaration should be preceded by the keyword friend.
- A friend function is not in the scope of the class to which it has been declared as friend.
- A friend function is just like a normal C++ function.

- A friend function can be declared either in the private or the public section of a class.
- Usually, a friend function has the objects as its arguments.
- A friend function cannot access the class members directly. An object name followed by dot operator, followed by the individual data member is specified for valid access. Example, if we want to access the member x of an object obj, then it specified as obj.x
- Member functions of one class can be friend functions of another class.
- A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access right to private member of the class.

Write a C++ Program to illustrate friend function

```
class sample
{
int a, b;
 public:
void setvalue()
{
a=25;
 b=40;
}
friend float mean(sample s);
};

float mean(sample s)
{
return float(s.a + s.b)/2.0;
}
void main()
{
sample x;                    //object x
x.setvalue();
cout<<"Mean value = " << mean(x) << "\n";
getch();
}
```

The friend function accesses the class variables a and b by using the dot operator and the object passed to it (Example: s.a, s.b). The function call mean(x) passes the object x by value to the friend function.

Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator, which is shown as follows:

```
class X
{
int fun1( );            //member function of X
.....
};
class Y
{
friend int X :: fun1( );          //fun1 of X is friend of Y
.....
};
```

Here, the function fun1( ) is a member of class X and a friend of class Y

We can also declare all the member functions of one class as the friend function of another class.

In such cases, the class is called a friend class. This can be specified as

follows:

```
class Z
{
.....
friend class X;          // All the member functions
            of X are friend to class Z
};
```

Constructors and Destructors

- A constructor is a special member function which is used to initialize the data members automatically the moment the objects are created. This is known as automatic initialization of objects.

A constructor is declared and defined as follows:

```
class sample
{
int m, n;
 public:
 sample(void);           //constructor declared
};
sample :: sample(void)     //constructor defined outside class
{
 m = n = 0;
}
```

OR

```
class sample
 {
 int m, n;
public:
 sample(void);     //constructor defined inside class
{
 m = n = 0;
}
};
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically.

Characteristics of a Constructor:

- A constructor has the same name as that of class.
- A constructor is declared in the public section of a class definition.
- A constructor is invoked automatically as soon as the class object is created.
- A constructor does not return any value, so return type is not associated with its definition (even void is not used).
- The programmer has no direct control over constructor functions.
- A constructor are not inherited
- The const or volatile class objects cannot be initialized with a constructor
- A constructor cannot be declared as virtual (Virtual functions).
- A constructor can be overloaded.
- Like other C++ functions, they can have default arguments.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make ‘implicit calls’ to the operators new and delete when memory allocation is required.

Types of Constructors:

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Dynamic Constructor
- Overloaded Constructor

Default Constructors:

A constructor that accepts no parameters is called the default constructor.

The default constructor for class Sample is sample::sample( ).

sample a; // Invokes the default constructor of the compiler to create the object a.

Program to illustrate default constructor

```
class box
{
int length, breadth, height;
public:
 box(void)
 {
 length = breadth = height = 0;
 }
void print(void)
 {
 cout << "length = " << length << endl;
 cout << "breadth = " << breadth << endl;
 cout << "height = " << height << endl;
 }
};
void main()
{
box obj1;
obj1.print();
getch();
}
```

OUTPUT:

```
length = 0
breadth = 0
height = 0
```

Parameterized Constructors:

In practice it may be necessary to initialise the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterised constructors.

The constructor sample( ) may be modified to take arguments as shown below:

```
 class sample
{
int m, n;
 public:
 sample(int x, int y);      // Parameterised Constructor
{
 m = x;
 n = y;
};
```

When a constructor has been parameterized, the object declaration statement such as

sample obj1;

may not work. We must pass the initial values as arguments to the constructor function to which an object is declared.

This can be done in two ways:

By calling the constructor explicitly

By calling the constructor implicitly

The following declaration illustrates the first method:

```
sample obj1 = sample(0, 100);          //explicit call
```

This statement creates a sample object obj1 and passes the values 0 and 100 to it.

The second is implemented as follows:

```
sample obj1(0, 100);          //implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. The following program demonstrates the passing of arguments to the constructor functions.

Program to illustrate parameterized constructor

```
class box
{
int length, breadth, height;
 public:
 box(int l, int b, int h)     // parameterized constructor
 {
 length = l;
 breadth = b;
 height = h;
 }
 void print(void)
 {
 cout << "length = " << length << endl;
 cout << "breadth = " << breadth << endl;
 cout << "height = " << height << endl;
 }
};
void main()
{
box obj1(2,4,3);
obj1.print();
getch();
}
```

OUTPUT:

```
length = 2
breadth = 4
height = 3
```

Copy Constructor

A constructor can accept a reference to its own class as a parameter.

For example,

sample(sample &i);   is valid.

In such cases, the constructor is called the copy constructor.

A copy constructor is one which constructs another object by copying the member of a given object.

It is used to declare and initialize an object from another object.

For example, the statement

sample X2(X1);

would define the object X2 and at the same time initialize it to the values of X1.

Another form of this statement is   sample X2 = X1;

The process of initializing through a copy constructor is known as copy initialization.

Program to illustrate copy constructor

```
class point
{
int x,y;
public:
point()    //default constructor
 {
 x = y = 0;
 }
point(int xp,int yp)        //parameterized
constructor
 {
 x = xp;
 y = yp;
 }
point(point &p)             //copy constructor
 {
 x = p.x;
y = p.y;
 }
 void display()
 {
cout << "x = " << x << "\t" << "y = " << y << “\n”;
 }
};
```

```
void main()
{
point p1(20,30);        //object p1 is created
                 and initialised
point p2(p1);       //copy constructor is called
point p3;
p1.display();
p2.display();
p3.display();
getch();
}
```

Destructor

- A destructor is a special member function which is also having the same name as that of its class but prefixed with tilde(~) symbol.

  For example, if sample is the name of the class then

  ~sample ( ); // is a destructor for the constructor sample( ).

Characteristics of a Destructor:

- A destructor is invoked automatically by the compiler upon exit from the program
- A destructor does not return any value
- A destructor cannot be declared as static, const or volatile.
- A destructor does not accept any arguments and therefore cannot be overloaded.
- A destructor is declared in the public section of a class.
- If the constructor uses the **new** expression to allocate memory, then the destructor should use the **delete** expression to free that memory.

Program to illustrate the destructors

```
class sample
{
 int x;
 public:
 sample(void)
 {
x = 0;
cout<< "In constructor, x = "<<x<<"\n";
 }
 void printx(void)
 {
 x = 25;
 cout<< "In printx, x = "<<x<<"\n";
 }
 ~sample()
 {
cout<< "In destructor, object destroyed";
 }
};
void main()
{
 sample s1;
 s1.printx( );
 getch();
}
```

OUTPUT:

```
In constructor, x = 0
In printx, x = 25
In destructor, object destroyed
```

Virtual Functions

- Polymorphism is one of the crucial features of OOP.
- It simply means ‘having many forms’. Polymorphism can be achieved in two levels, one at compile time and another at run-time.
- In C++, compile time polymorphism is achieved using overloaded functions and operators, and the run time polymorphism is achieved using the virtual functions. The implementation of virtual functions requires the use of pointers to objects.
- When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration.
- When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.
- Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual functions.

Rules for Virtual Functions

Following are the some of the basic rules that satisfy the compiler requirements:

1. The virtual functions must be member of some class.
2. They cannot be static member function.
3. They are accessed by using object pointers.
4. A virtual functions can be a friend of another class.
5. They should be declared in public section of the class.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. (If two functions with the same name have different prototypes, C++ considers them as overloaded functions.)
7. While a base pointer can point to any type of the derived object, the reverse is not true.
8. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Write a C++ Program to illustrate Virtual Functions

```
class baseclas
{
public:
virtual void virtfunc( )
{
cout<<"This is baseclas's virtfunc ( )\n";
}
};
class derivedclas1 : public baseclas
{
public:
void virtfunc( )
{
cout<<"This is derivedclas1's virtfunc ( )\n";
}
};
class derivedclas2 : public baseclas
{
public:
void virtfunc( )
{
cout<<"This is derivedclas2's virtfunc ( )\n";
}
void main( )
{
baseclas *ptr, b;
derivedclas1 d1;
derivedclas2 d2;
ptr = &b;
ptr -> virtfunc( );       // Access baseclas's
virtfunc( )
ptr = &d1; // Point to derivedclas1
ptr -> virtfunc( );    // Access derivedclas1's
virtfunc( )
ptr = &d2; // Point to derivedclas2
ptr -> virtfunc( ); // Access derivedclas2
virtfunc( )
getch( );
}
```

OUTPUT:

```
This is baseclas's virtfunc ( )
This is derivedclas1's virtfunc ( )
This is derivedclas2's virtfunc ( )
```

- Here, the address of the object b is stored in ptr (is a pointer variable). Then we are accessing the member function virtfunc( ) through the ptr.
- Next the address of the object d1 is stored in ptr and accessing the member function virtfunc( ) through the ptr.
- Finally the address of the object d2 is stored in ptr and accessing the member function virtfunc( ) through the ptr.
- It is also possible to access virtfunc( ) through the objects directly. (i.e., b.virtfunc( ))

Unit 2: Arrays, Matrices and Linked lists

- Array is a collection of similar data items in which each element is unique and located in separate continuous memory locations. The amount of storage required for holding elements of the array depends on its type and size.
- For example, an internal has been conducted to 4th Semester ECE students, they are 70 in number. So 70 IA marks have to be stored in an array. Let array name be IA. It can be declared as

int IA[70]; // IA is an Array Name, 70 is the size of the array

- Here, members of the array IA are IA[0], IA[1], … IA[69], the roll number of the student from 1 to 70. The subscripted variable IA[0] stores the marks of the 1st student.

- Always an array index from 0th location and ends with n - 1 (Here, n is the size of the array).

Array Types

Arrays are classified as 2 types.

1) One-Dimentional Array

2) Multi-Dimentional Array

One Dimensional Array

- One-dimensional array is a linear list of fixed number of data items of the same type. All these data items are accessed using the same name using a single subscript.
- An array name occurs with single subscript ([ ]) is called as One-dimentional Array. One pair of Square bracket is called subscript.

Declaration of One Dimensional Array:

- Array must be declared before like declaring an ordinary variable. At the same time size of an array must be specified. This allows the compiler to decide on how much memory is to be reserved for the array.

  The syntax: **Data_type** arrayname[size];

  Ex: **int** IA[70]; // size of array is 70
  float avg[30]; // size of array is 30

Initialization of One Dimensional Array

- WKT, variables can be initialized to some values when they are declared. Ex: **int** count=0;

Similarly, the arrays can also be initialized to some initial values.

Ex: **int** marks[6] = {35,40,81,56,48,71};

- Arrays elements must be included in a pair of curly braces and separated by comma, initial values of the array elements are assigned to array marks[6].
- If the array is declared to contain integer numbers, all the initial values must be integers only.
- The marks[0] = 35, marks[1]=40 …… ,marks[5]=71

Multidimensional Arrays

- Arrays of more than one dimension is called multi-dimensional arrays. It can be declared merely by adding more subscripts.
- The two dimensional array (contains 2 subscripts) is made up of rows (1st subscript) and columns (2nd subscript).

Declaration of Two Dimensional Arrays:

- Two dimensional arrays have to be declared with two subscripts and declaration must specify the size and data type of the array.

Ex: **int** table[3][3];

- It specifies the table is two dimensional array with 3 row and 3 columns consisting of 3 x 3 = 9 elements of integer type. Here, Array index starts with table[0][0], table[0][1], table[0][2], table[1][0], table[1][1], table[1][2], table[2][0], table[2][1], and array index ended with table[2][2].

Initialization of Two-Dimensional Array:

1. Two dimensional arrays can be initialized like one-dimensional array. For eg.,

```
int table[3][2] = {   {14,8},
          {10,12},
          {15,14}
              };
```

- Two-dimensional array named **table** elements are initialized row by row. Thus each row elements are enclosed in a pair of brace.
- Observe that each pair of braces is separated by comma to indicate separate rows.
- Note that if the array named table is initialized completely for all of its elements then inner pairs of braces are not required. Thus it is sufficient to write.

```
int table[3][2] ={14,8,10,12,15,14};
```

2. Like one-dimensional arrays, it is not compulsory to initialize all members of the two-dimensional arrays also.

```
Ex: int table[3][2] = {    {14},
                      {10,12},
                      {}
                      };
```

- Remaining elements of the array report are considered to be zeros.

Data Structure

- The data structure is logically or mathematically organized data items. Representing or Storing the data in a systematic way in computer memory is called data structure.

Operations on Data Structures

- Creation of data structure.
- Deletion of data structure.
- Insertion of data structure.
- Accessing elements within a data structure, called selection operation.
- Modification of contents in the data structure.
- Sort operation to arrange elements within a structure.

The overall intention of all these operations is to access data elements to provide useful results.

Classification of Data structures

- A systematic grouping of data structure broadly classified into

  Primitive data structure and

  Non-primitive data structure

- **Primitive data structure:**

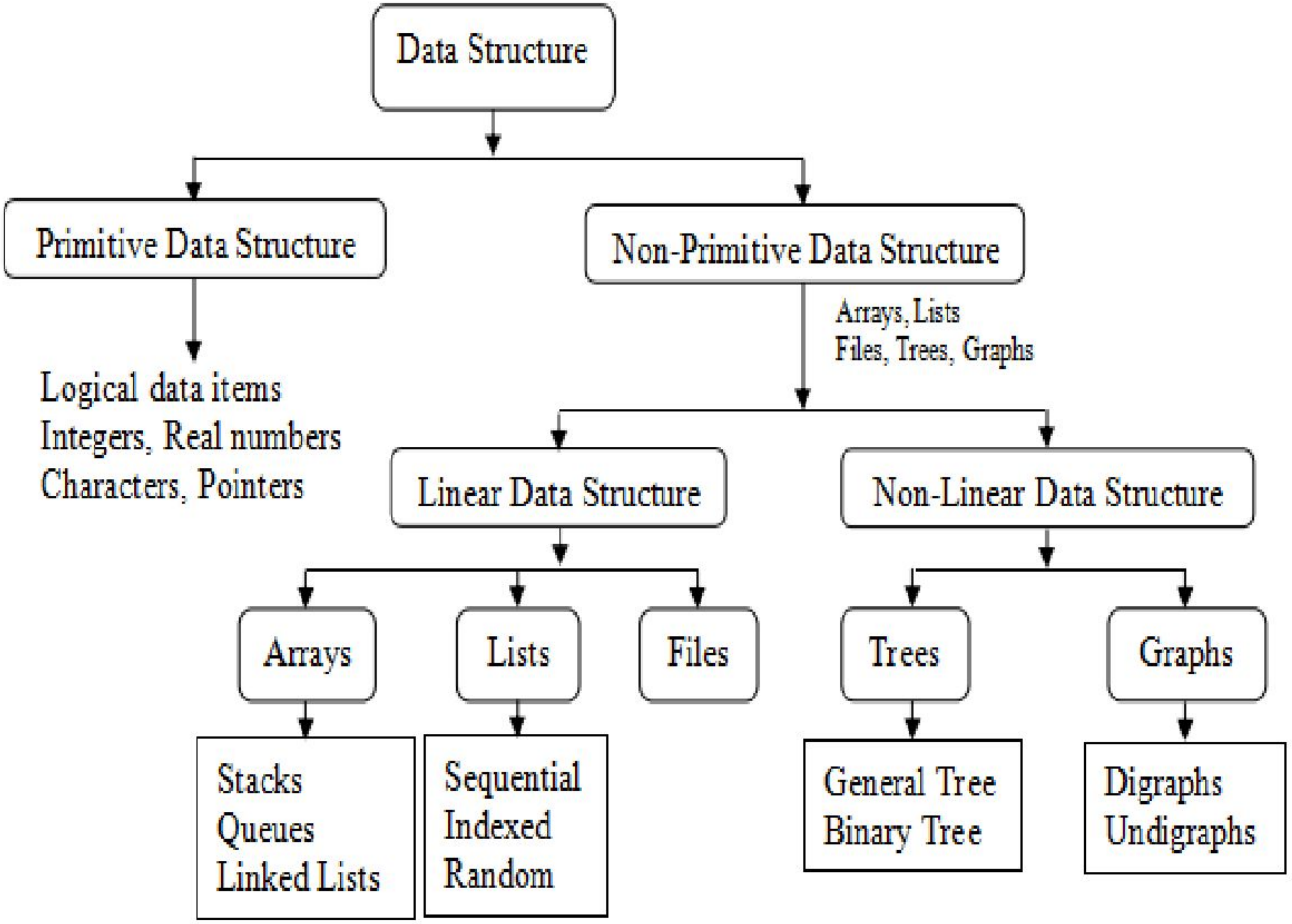
Primitive data types include integers, real numbers, characters, logical data items and pointer data types.

- **Non-primitive data structure**

Non-primitive data structure include arrays, lists, files, trees, graphs. Non-primitive data structure further classified into

1. Linear data structure
2. Non-linear data structure.

The data structure classification details are as shown below:



Non-primitive data structure

- A data structure, which is derived from primitive data types, is referred to as non-primitive data types or non-primitive data structures.

For example an array consisting if integer elements (primitive data type) is classified as Non-primitive data structure.

Operations on Non-primitive Data Structures

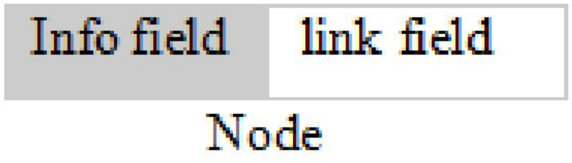
- The type of operation that can be applied on a particular non-primitive data structure varies functionally depending upon their logical organisation and storage structure.

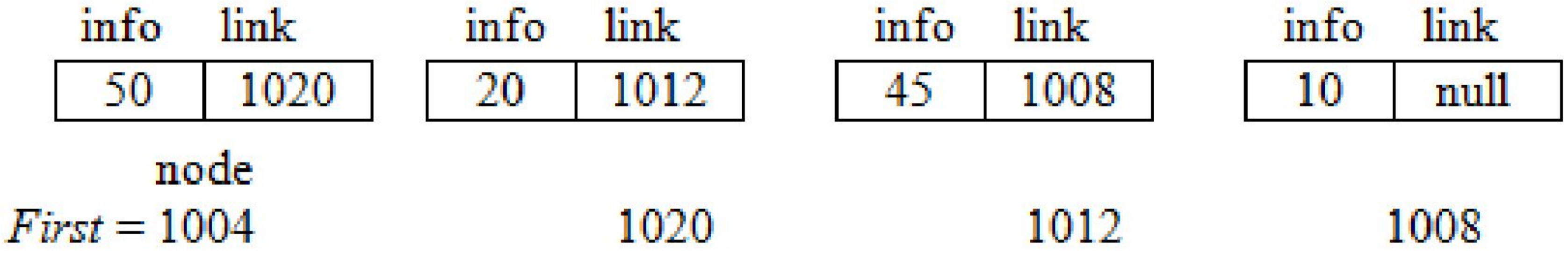
- Create or define an array / list / file
- Search for an element in an array / list, locate a record in a file
- Sort the elements of an array into ascending or descending order
- Combine or merge two or more lists to form another new list.
- Insert elements into and delete elements from a list
- Copy a list, Update a file, rewind a file, delete records etc.,
- Delete a node from a binary tree, insert a new node in a binary tree.
- Traverse a tree in inorder, postorder, preorder.

LINKED LISTS

- We call stacks and queues as restricted lists because insertions and deletions are restricted to occur in these structures only at the ends of the list.
- In stacks and queues all the items are kept sequentially in arrays, declaring a maximum size of arrays. We cannot increase the amount of storage allotted at run time, which would cause overflow.
- The problems discussed on sequential storage representation can be solved by the introduction of a new data structure called **linked lists.**

- A linked list is a series of data items with each item containing a pointer giving the location of the link item in the list.
- We can place the elements anywhere in the memory, but to make them a part of the same list it is required to be linked with the previous element of the list.
- This can be done by storing the address of the link element in the previous element itself. This requires that every element must be capable of holding the data as well as the address of the link element.
- Thus every data element (called *nodes*) must be a record with a minimum of two fields, one for holding the data value (info), which we call a *data (info) field*, and the other for holding the address of the link element, which we call *linked field (link)*.





Fig(a): Memory representation of singly linked list

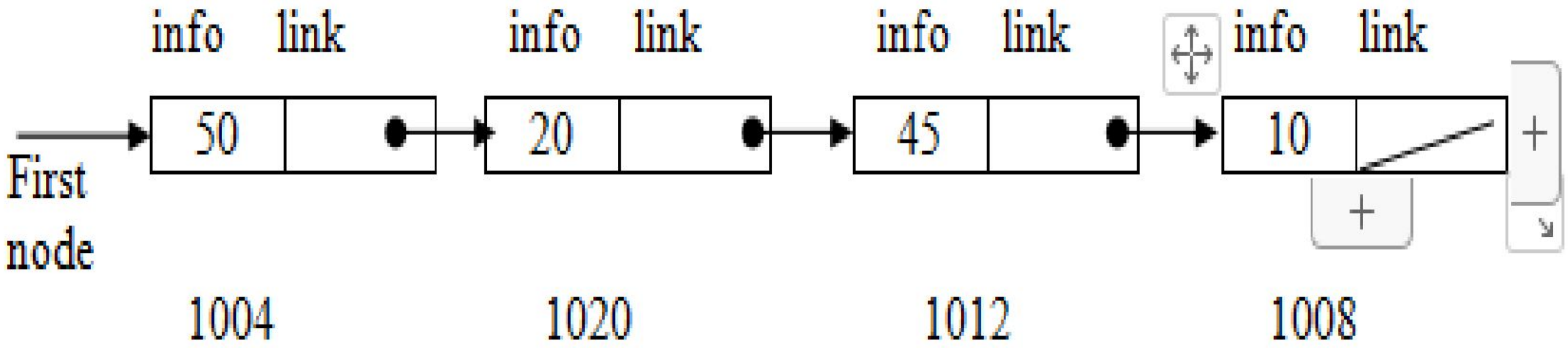


Fig (b):Pictorial representation of singly linked list

The memory representation and pictorial representation of a singly linked list is shown in figure (a) and (b) respectively.

- This list contains 4 nodes and each node consists of two fields, *info* and *link*. The first field of each node i.e., *info* contains the data that has to be stored in a list.
- In this list the data items 50, 20, 45, and 10 are stored.
- The second field i.e., link of each node contains address of the next node.
- Note the arrow originating from link field of each node.
- This arrow indicates that the address of the next node is stored.
- So, using *link* field, any node in the list can be accessed since the nodes in the list are logically adjacent.

- Nodes that are physically adjacent need not be logically adjacent in the list.
- For example, the nodes identified by the address 1004, 1020, 1012, and 1008 are physically not adjacent but they are logically adjacent
- In this list the variable *first* contains the address of the first node. Before creating a list, *first* should be initialized to NULL indicating, the list is to empty to start with.
- Once the list is created, variable *first* contains address of the first node of the list. The nodes in the list can be accessed using a pointer variable, which contains the address of the first node. If *first* points to NULL, it means that the list is empty.

Components of a Linked List

- Nodes: A linked list is a non-sequential collection of data items called nodes. These nodes have two fields called data field and link field.
- Data (info) fields: It contains the actual value to be stored and processed.
- Link fields: It is used to access a particular node and also known as pointers.
- Null pointer: The link field of the last node contains zero rather than a valued address. It is known as null pointer and indicates the end of the list.
- External pointer: This points to very first node in the linked list.
- Empty list: A list with no nodes is called empty list or null list.

Operations on Linked Lists

- The basic operation that can be performed on a linked list are
- Create: Used to create a linked list with one or more nodes and their information field is initialized with some value.
- Insert: Used to insert a new node in the existing list the new node can be inserted.

  a) At the front of a list.

  b) At rear end of a list.

  c) At the specified position in a linked list. This operation is termed as place.
- Delete (Remove): Used to delete a node from the existing list. The node can be deleted from.

  Front of the list.
  - Rear end of the list.
  - From the specified position in the list.
- Traverse: Used to visit all the nodes of the list.
  - Forward traversing: It refers to traveling the list from first node to last node.
  - Backward traversing: It refers to traveling the list from the last node to the first node.

5) Search: Used to search whether a given data item is present in the list or not.

6) Concatenate: Used to appends or joins a second list at the end of first list.

7) Display: Used to_displays all the data items from a list.

8) Sort: Used to sort the elements of the Linked Lists in ascending or descending order

9) Reverse: Used to reverse the Linked Lists.

10) Print: Used to print the Linked Lists

11) Decompose: Used to decomposition of a Linked Lists into Even and Odd Linked Lists.

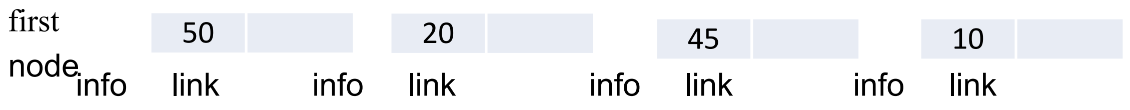
Types of linked lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Non-Integer and Non-Homogeneous Linked List

1) Singly Linked List:

In singly linked list nodes linked together by a single pointer and can be traversed only in the forward direction.

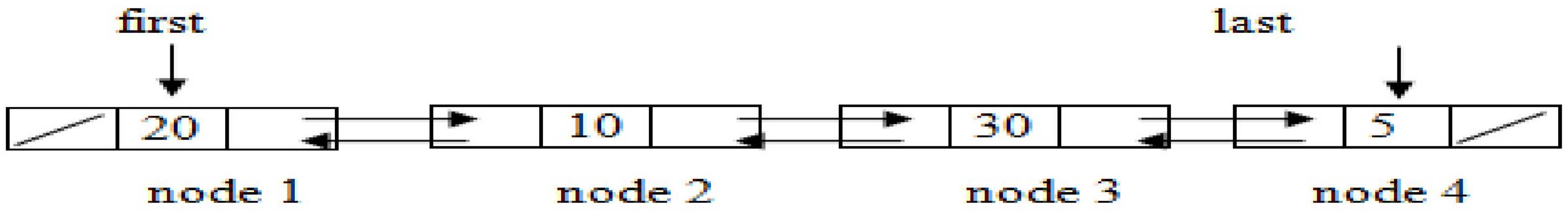
It has two ends a front end and the rear end.



Pictorial representation of singly linked list

2) **Doubly Linked Lists:**

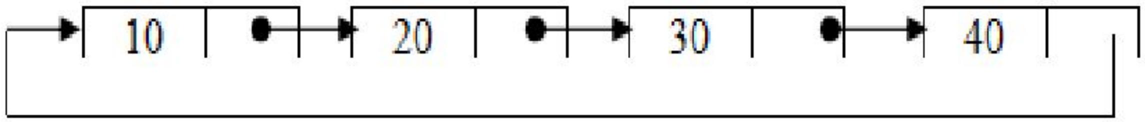
In Doubly linked list adjacent nodes are linked by two pointers a left pointer and a right pointer. The right pointer will point to the next node whereas the left pointer will point to the previous node. Doubly linked list can be traversed in both forward and backward directions and locate given node predecessor and successor.



3) Circular Linked Lists:

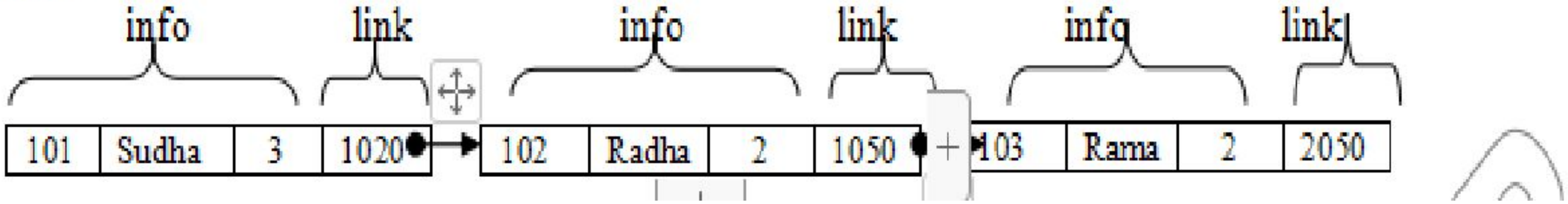
- In the linked lists, the link field of the last node contained a NULL pointer.
- We can modify the linked list by storing address of the first node in the link field of the last node. The resulting list is circular list.
- From any point in such a list it is possible to reach any other point in the list.
- A circular linked list does not have a first or last node. We must, therefore establish a first and last node by convention.
- A circular list can be used to represent a stack or a queue.

- A circular list can be of singly linked or doubly linked.
- A pictorial representation of a circular list is shown in the following figure.



```
struct student
            {
                  int id;
                  char name[10];
                  int sem;
                  struct student *link;
            } *stud;
```

Here, the student details such as name, id and semester in which he is studying are the input.

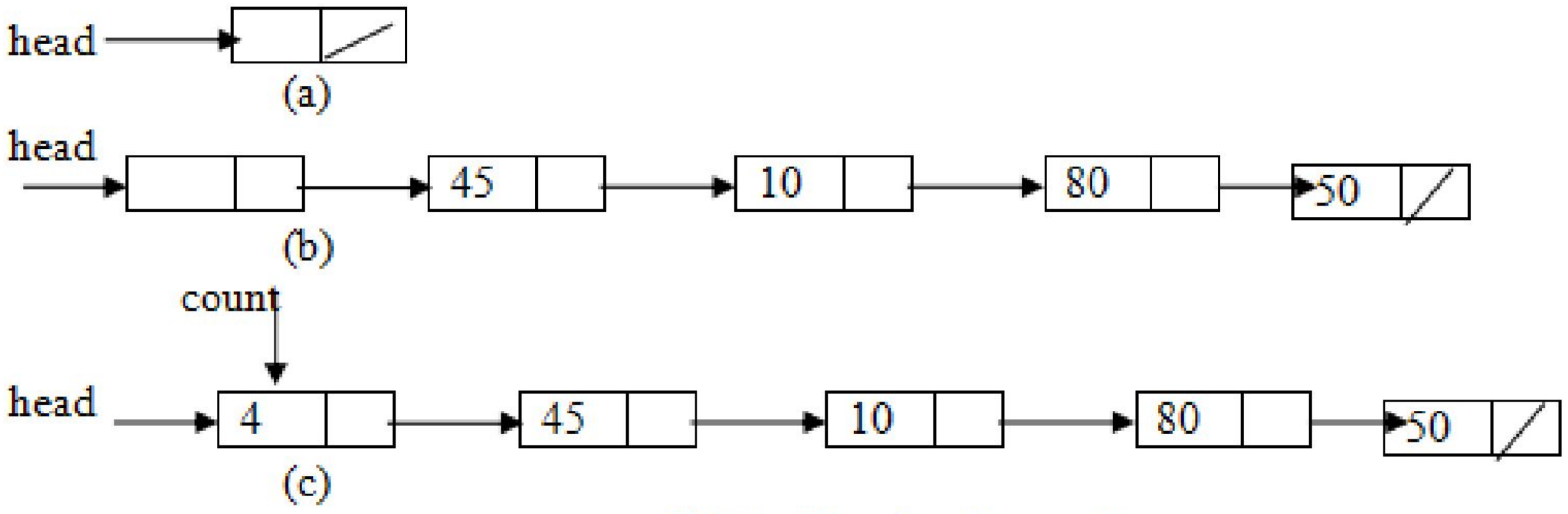


Header Nodes:

- It may be useful to have a node in the beginning of the list. The purpose of which is used to simplify the design. Such a node is called a header node.
- The info field of such node does not represent an item in the list. Sometimes, useful information such as, number of nodes in the list can be stored in the info field.

For example, the **info field** of list contains 4, which represents the total number of nodes present in the list. In such a case, there is an overhead of altering the info field of this node.

Each time the node is added or deleted, the count in this field must be readjusted so as to contain actual number of node to be accessed in the list.



List with a header node

Applications of Linked Lists

1) Linked lists are used to perform polynomial manipulations.

2) The common operations that are performed are addition, subtraction, division, multiplication, integration and differentiation.

3) Linked lists are also used to maintain a dictionary of names.

Advantages of Linked Lists

1.The size is not fixed: The linked list can grow or shrink depending on the data availability. If more memory space is required, the required memory space can be allocated if available. If certain portion of memory is required, the required memory space can be allocated if available. If certain portion of memory is not required, that memory space can be de-allocated so that some other application can use that freed memory space.

2.Data can be stored in non-contiguous blocks of memory: Usually we will not fall shortage of memory space since non-contiguous blocks of memory can be used to store the data.

3.Insertion and deletion of nodes is easier: Unlike arrays, a node can be inserted into any position in the list and a node can be deleted from any position in the list. In such situations, there is no need to shift the nodes.

4.Complex problems can be easily solved by using linked list.

Disadvantages of linked lists

1.The linked list requires extra space because each node in the list has a special field called link field that hold the address of the next node.

2.Accessing a particular node in the list may take more time when compared with arrays. The list has to be traversed from the first node to the particular node. Naturally this process takes more time.

3.Accessing to an arbitrary data item is a little bit difficult and confusing.

4.If the address of first node is lost then the entire list cannot be accessed.

5.Only one element can be accessed in one traversal

<u>Inserting a node at the front end of the list:</u>

- Creating the linked list is nothing but repeated insertion of items into the list. Insertion can be done at the front end of the list.
- To insert an item, a new node is required first. This new node can be obtained from the availability list (availability list is list of free nodes).
- To obtain memory space for a new node, we can use the function getnode( ).
- **Step 1:** Let us assume we have a list of 3 nodes already created and each of its info field contains an integer value. Obtain the new node from the availability list and this node be temp. This can be achieved using the statement

```
temp = getnode( );
```

Also note that the external pointer *first* points to the first node of this list.

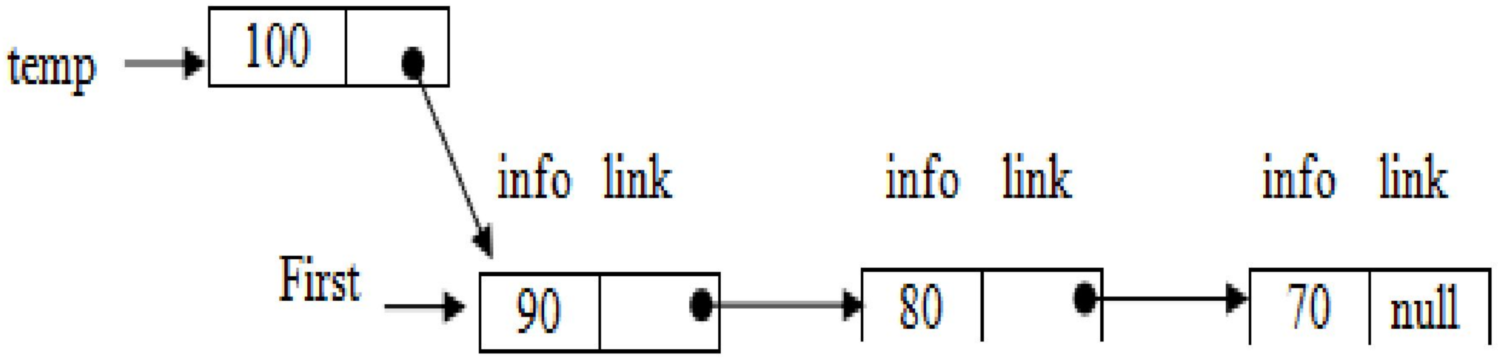
Step 2: Store the item 100 in the information field of node temp using the statement:

info(temp) = item;

Step 3: Attach existing first node to the link field of temp using the following statement

link(temp) = first;

After insertion of an element, the list will be look like as follows:



Step 4: Since temp points to the newly inserted node at the front of the list, return its address to the calling routine using the statement

```
return temp;
```

where temp is the address of the new first node.Hence, the algorithmic function to insert an item is as shown below.

```
insert_front(item, first)
 {
 temp = getnode( );          //get a node from the availability first
 info(temp) = item;     //insert the item    link(temp) =
first;    //attach to the existing first node of the list
 return temp;    //make temp as new first node
 }
```

Note: Let us use first as a pointer variable which always points to the first node of the list.

This can be achieved by calling the function

    Insert_front( ) as shown below:

    first = insert_front(item, first);

When this statement is executed, the variable first points to first node of the modified list.

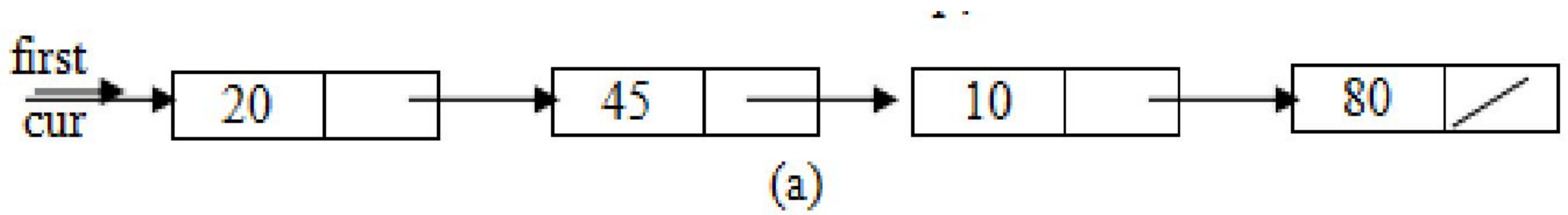
The algorithm has been designed by assuming that the list already exists.

If the list is empty, the pointer variable first contains NULL to start with.

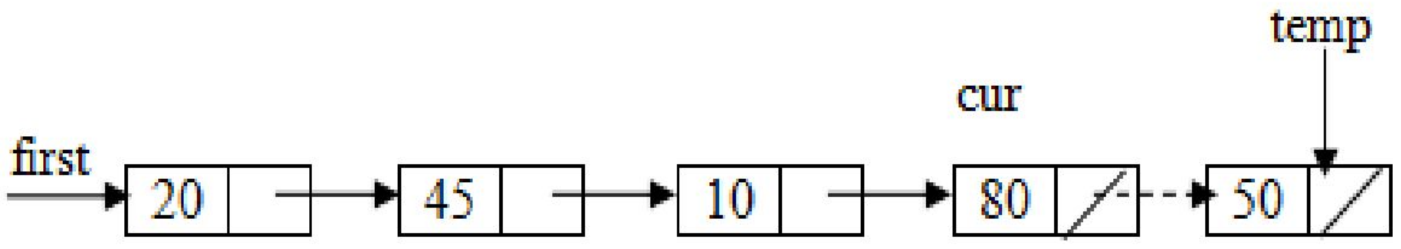
Insert the items by calling the function insert_front( ) again and again to create a list

Inserting a node at the rear end of the list

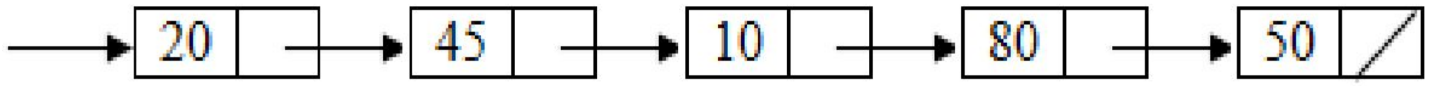
- Consider the list shown in below fig (a). Suppose, temp is the node to be inserted at the rear end with an item 50 as shown in (b).



(a)



(b)



(c)

- To insert node temp at the rear end of the list, the address of last node of the existing list should be obtained. This can be achieved by using an auxiliary pointer variable **cur**.
- Initially **cur** points to the first node of the list as shown in (a). Update the cur pointer, to point its successor node one after the other, until address of the last node is obtained.
- The last node can be achieved by the statement.

```
cur = cur ->link;
```

- Once address of the last node is pointed by cur, the temp can be easily inserted at the end.
- This can be achieved by assigning the address of the node to be inserted i.e., temp, to the link field of the last node. The statement is

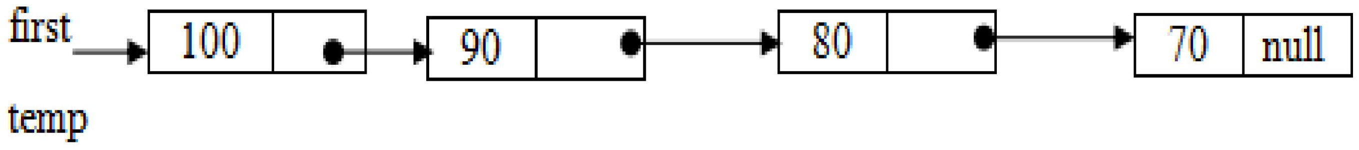
```
cur->link = temp;
```

- All these steps have been designed, by assuming that the linked list already exists. So, the program segment is as follows:

```
if(first!=NULL)    // to check empty list
{
     cur = first;
     while(cur->link!=NULL)
     {
          cur = cur->link;
     }
     cur->link=temp;
}
return first;
```

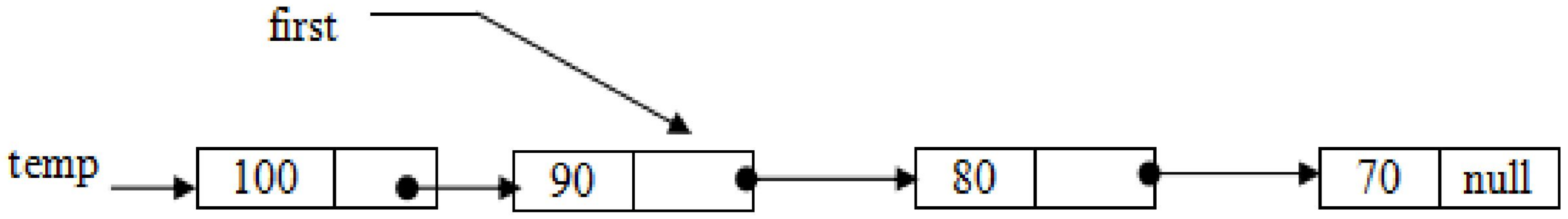
Deleting a node from the front end of the list

- *Step 1*: Let us assume we have a list of 4 nodes and external pointer first points to the very first node of this list as shown in figure below:

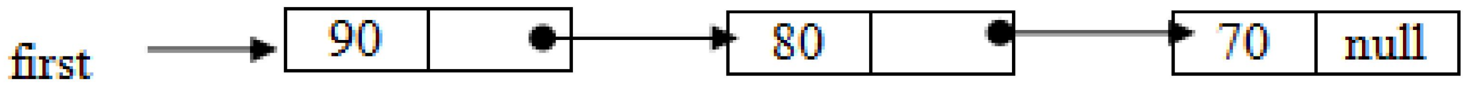


- *Step 2*: To delete a node from the front end of list, apart from the variable *first*, temporary variable *temp* is used. Initially both *first* and *temp* pointers to point to the first node of the list. This can be achieved by the statement:
- temp = first;
- *Step 3*: After deleting the first node, the next should be first node and the variable first should be point to it. Before deleting the first node, update the pointer *first* to point the next node. This can be achieved by the following statement.
- first = first->link;

- *__Step 4:__* The resulting linked list is of the form shown in the following figure.



- *__Step 5__*: To indicate the completion of delete operation, we should discard the pointer variable *temp* and at the same time node is no longer in use. Therefore this *temp* node can be deleted by using the function freenode( ).This can be written as follows:
- freenode(temp);
- *__Step 6__*: The node pointed by *temp* is deleted and is returned to the availability list. Now, the resulting linked list consist of only three nodes as shown below:



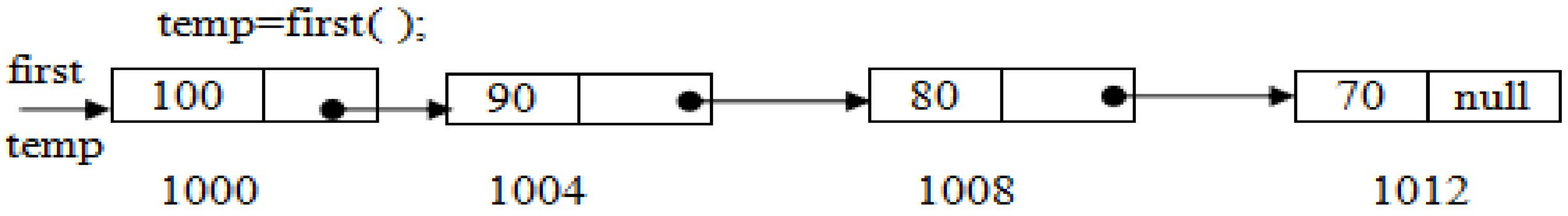
- Once the node is deleted, return the address of the new first node identified by the first, by executing the statement

  ```
  return first;
  ```
- Note that all these statements have to be executed, only when the list exists. If the list is empty, display the message, “List is empty, nothing is to delete” and return NULL.
- Hence, the algorithmic function to delete an item is as shown below.

```
delete_front(node first)
{
node temp;
if(list==NULL)
{
cout<<“List is empty, nothing is to delete\n”;
return first;
}
temp = first;             //retain address of the node to be deleted
first = first->link;    //Make the successor node as the first node
cout<<“The item deleted is \n”<< list;
freenode(temp);
return first;
 }
```

Display of singly linked list

- One the list is created, the next step is to display the contents of the list. Consider the list as shown below, the variable first contains address of the first node.
- Using first, the entire list can be traversed one by one. We use an auxiliary pointer variable temp that initially points to the first node itself.



- The items to be displayed are 100, 90, 80 and 70.
- The first item 100 can be displayed using the following statement.

```
write(info(temp))
```

- The next item to be displayed is 90. To display 90 using the above statement, temp should contain address of its successor.
- This can be done by assigning link(temp) which contains address of its successor to temp using the following statement.

```
temp = link(temp)     //address of the node is copied to temp
```

(Note: For example, if temp points to 1000, when we execute the above statement, the right hand side yields 1004 and this address is copied into temp. Since temp contains 1004, now temp points to the second node).

Thus by executing temp=link(temp) we can point **temp** to point to the next node.

- Now, display the item pointed to by temp as shown earlier. Repeating this process, all the items in the list can be displayed. The formal version of the algorithm can be written as follows.

```
write(info(temp))
temp=link(temp)
```

- After executing these two statements for the first time, the 100 is displayed. Going back as shown using an arrow and executing the statements, the next 90 is displayed.
- If this process is repeated again and again all the items will be displayed. After displaying 70, note that temp points to NULL, indicating all the nodes in the list have been displayed.
- Now, there is no need to go back and execute those statements i.e., these statements have to be repeatedly executed, as long as temp is not pointing to NULL.
- If first points to NULL, it will display the message “List is empty”.

Algorithm to display the contents of singly linked list:

```
    display(first)
{
    if (first == NULL)
    {
    write ‘list is empty’
    return;
    }
    write ‘The contents of the list’
    temp = first
    while(temp!=NULL)
    {
    write(info(temp)
    temp=link(temp)
    }
}
```

The difference between singly linked lists and doubly linked lists

Singly Linked Lists	Doubly Linked Lists
1. Traversing is possible in only one direction	1. Traversing is possible in both the direction
2. To delete a specific node address of its predecessor should be known. To find the address of the predecessor, it is required to traverse the list from the very first node.	2. The address of the previous node can be obtained from the left link of the node to be deleted.

- The difference between Circular Queue and Circular Linked list

Circular Queue	**Circular Linked List**
1) Size of the array is fixed	1) No size of the circular linked list is fixed
2) Stores in contiguous memory spaces	2) Stores in non-contiguous memory spaces
3) Since the data items stores in contiguous memory locations there is front end and rear end	3) Since the data items stores in non-contiguous memory locations there is no front end and rear end
4) In insertions and deletions are in FIFO order	4) In insertions and deletions are in any order ie., front element, rear element, and specified location
5) Manipulation of stored data is sequential	5) Manipulation of stored data is not sequential. Any desired data can be accessed
6) Spent more time in shifting of array elements	6) There is no shifting of nodes

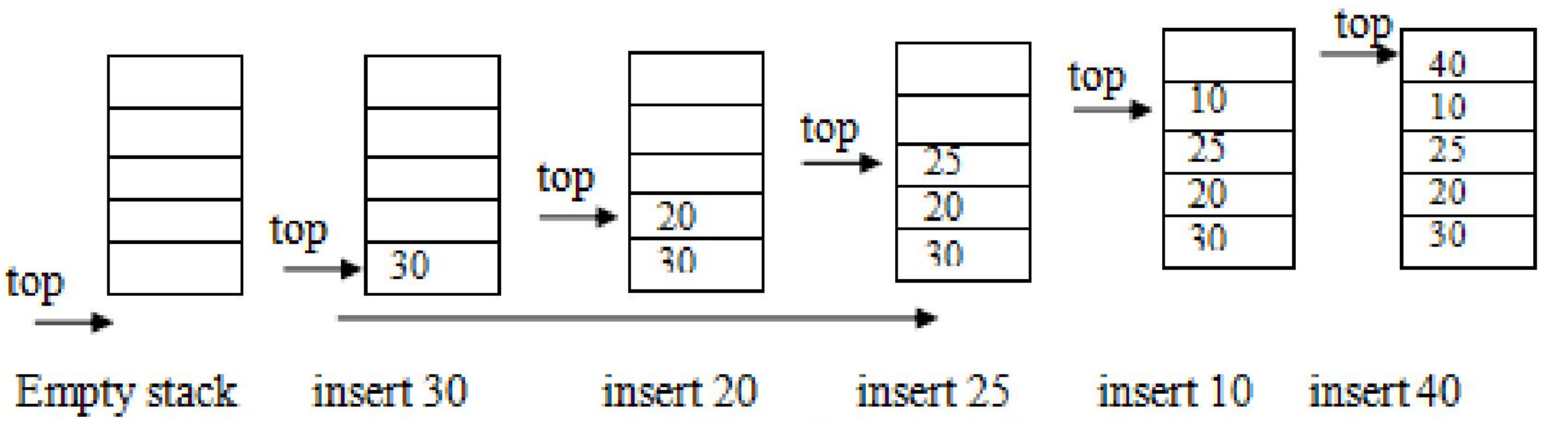
Stacks

We know that in a cafeteria the plates are placed one above the other and every new plate is added at top. When a plate is required, it is taken off from the top and it is used.

We call this process as stacking of plates. It was natural, that when man started programming data, stack was one of the first structures that he thought of when faced with the problem of maintaining data in an orderly fashion.

A stack is a data structure in which addition of new element or deletion of an existing always takes place at the same end. This is often known as top. Here, the last item inserted will be on top of stack. Since the deletion is done from the same end, Last item Inserted is the First item to be deleted Out from the stack and so, stack is also called **Last In First Out (LIFO) data structure.**

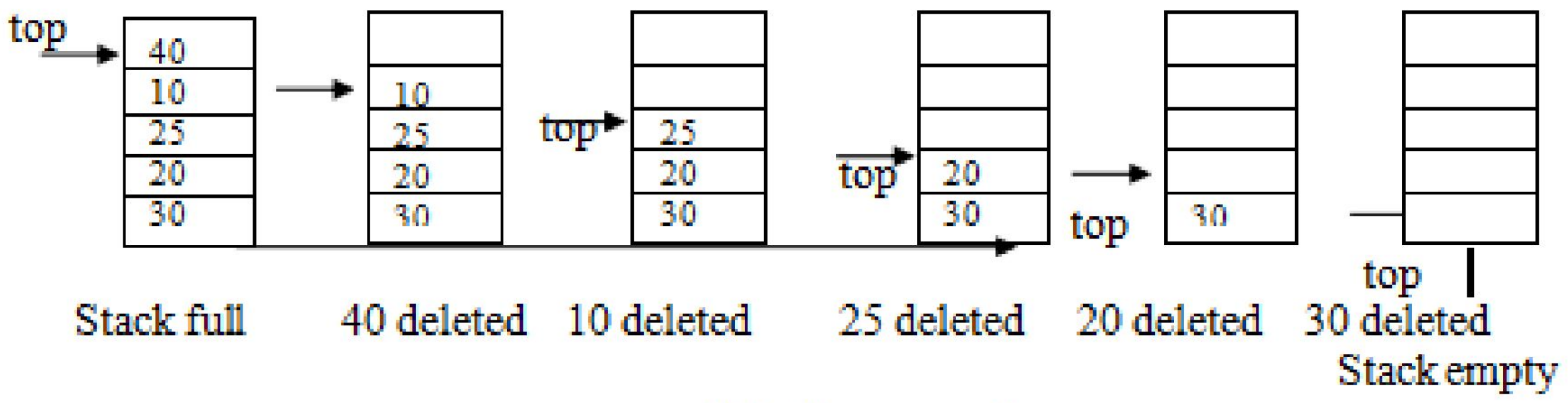
- The various operations that can be performed on stacks are:
  - Insert an item into the stack
  - Delete an item from the stack
  - Display the contents of the stack
- From the definition of the of the stack, it is clear that it is a collection of similar type of items and naturally we can use an array (an array is a collection of similar data types) to hold the items of stack. Since array is used, its size is fixed.



Insertion operation

It is clear from this figure that initially stack is empty and top point at the bottom of the stack. As the items are inserted top pointer is incremented and it points to the topmost item. Here, the items 30, 20, 25, 10 and 40 are inserted one after the other. After inserting 40 the stack is full. In this situation it is not possible to insert any new item. This situation is called ***stack overflow*.**

When an item is to be deleted, it should be deleted from the top as shown in delete operation



Deletion operation

Since items are inserted from one end, in stack deletions should be done from the same end. So, the items are deleted, the item below the top item becomes the new top item and so the position of the top most item is decrements as shown in the figure. The items deleted in order are 40, 10, 25, 20 and 30. Finally, when all items are deleted, top points to bottom of stack. When the stack is empty, it is not possible to delete any item and this situation is called ***under flow of stack***.

So, the main operations to be performed on stacks are insertion and deletion. Inserting an item into stack when stack is not full is called **push operation** and deleting an item from the stack when stack is not empty is called **pop operation.**

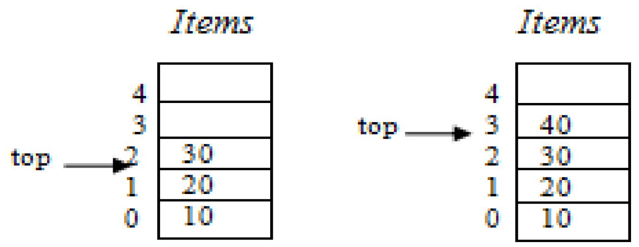
Other operations that can be performed are **display** the contents of the stack, check whether the stack is empty or not, etc.

Insert / Push Operation

Inserting an element into the stack is called **push operation.** Let us assume that three items are already added to the stack and stack is identified by s as shown below.

Here, the index top points to 30 which is the topmost item. The value of top is 2. Now, if an item 40 is to be inserted, the first increment top by 1 and then inserts an item. The corresponding C statements are:

```
top = top + 1;
s[top] = item;
```



These two statements can also be written as

```
s[top+1] = item;     or  s[++top] = item;
```

stack_size should be max = some integer value (Ex: 5) and is called symbolic constant, the value of which cannot be modified.

- As we insert an item we must take care of the overflow situation i.e., when top reaches stack_size–1, stack results in overflow condition and appropriate error message has to be returned and items has to be inserted as shown below:

```
if (top==stack_size-1)
 {
  cout<<“Stack overflow\n”;
  return;
 }
s[top+1] = item;
```

Delete/pop operation

- Deleting an element from the stack is called pop operation. This can be achieved by first accessing the top element s[top] and then decrementing top by one as show below:

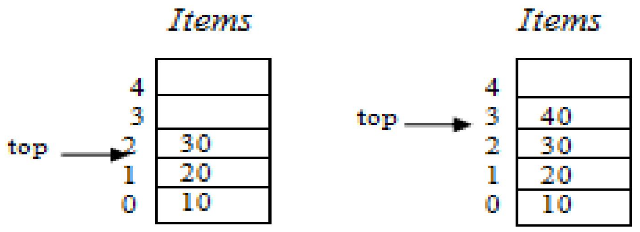
```
        item = s[top- -];
```

- If items are present in the stack, an item is always deleted from the top of the stack. Trying to delete an item from the empty stack results in stack underflow. The above statement has to be executed only if stack is not empty. Hence, the code to delete an item from stack can be written as

```
 if(top < 0)
{
cout<<"Stack is empty";;
}
item =s[top- -];
cout<<item;
```

Display

Assume that the stack contains three elements as shown below:



The item 30 is at the top of the stack and item 10 is at the bottom of the stack. Usually, the contents of the stack are displayed from the bottom of the stack till the top of the stack is reached. So the first item to be displayed is 10, next item to be displayed is 20 and final item to be displayed is 30. So, the code corresponding to this can take the following form

```
for(i=0; i<=top; i++)
{
    cout<<s[i];
}
```

But, the above statement should not be executed when stack is empty i.e., when top takes the value lessthan 0. So, the modified code can be written as shown below.

```
void display()
{
        int i ;
        if(top < 0)
        {
        cout<<“Stack is empty\n”;
        }
cout<<“Contents of the stack\n”;
for(i=0; i<=top; i++)
        {
        cout<< s[i];
        }
    }
```

Abstract Datatypes (ADT)

- An ADT may be defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called **“abstract”** because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as **abstraction.**

Stack ADT

```
Abstract Datatype stack
  {
    instances
    linear lists of elements; one end is called the bottom; the other
    end is the top.
    operations
    Create(): Create an empty stack;
    Isempty(): Return true if stack is empty, return false otherwise;
    Isfull(): Return true if stack is full, return false otherwise;
    Top(): Returns the top element of the stack;
    Add(x) / Push(x): Add element x to the stack;
    Delete(x) / Pop(x): Delete top element from the stack & put it in
    x;
  }
```

List ADT

A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

Abstract Datatype **List**

```
{
 get() – Return an element from the list at any given position.
 insert() – Insert an element at any position of the list.
 remove() – Remove the first occurrence of any element from a
 non-empty list.
 removeAt() – Remove the element at a specified location from a
 non-empty list.
 replace() – Replace an element at any position by another element.
 size() – Return the number of elements in the list.
 isEmpty() – Return true if the list is empty, otherwise return false.
 isFull() – Return true if the list is full, otherwise return false.
 }
```

Queue ADT

The queue abstract data type (ADT) follows the basic design of the stack abstract data type.

A Queue contains elements of the same type arranged in sequential order. Operations take place at both ends, insertion is done at the rear end and deletion is done at the front end.

Following operations can be performed:

```
Abstract Datatype Queue
{
        enqueue() – Insert an element at the end of the queue.
        dequeue() – Remove and return the first element of the queue, if the queue is not empty.
        peek() – Return the element of the queue without removing it, if the queue is not empty.
        size() – Return the number of elements in the queue.
        isEmpty() – Return true if the queue is empty, otherwise return false.
        isFull() – Return true if the queue is full, otherwise return false.
}
```

Pointer / Linked Representation

- While the array representation of stack in the previous section is considered both elegant and efficient but is wasteful of memory when multiple stacks are coexist.
- Multiple stacks can be represented efficiently using a chain for each stack. This representation incurs a space penalty of one pointer field for each stack element.
- When using a chain to represent a stack, we must decide which end of the chain corresponds to the stack top. If we associate the right end of the chain with the stack top, then insertions and deletion operations takes Ɵ(n) time.
- On the other hand, If we associate the left end of the chain with the stack top, then insertions and deletion operations takes Ɵ(1) time. This shows that we should use the left end of the chain to represent the stack top.

Write a complete C++ program to implement stack operations using linked representation.

Applications of Stack

Following are the some of the various applications of stacks.

1. To match left & right parenthesis in an expression.
2. Tower of Hanoi problem.
3. Shutting tracks in a rail yard.
4. CAD of circuit field.
5. Offline equivalence class problem.
6. Rat in maze problem
7. Conversion of infix expressions
8. Evaluation of postfix expressions
9. Recursion
10. Other applications like, to find whether the string is a palindrome or not, topological sort, etc...

Parenthesis Matching

- In a character string, we are to match the left and right parenthesis of type { } and ( ).
- For example, string (a*(b+c)+d) contains left parenthesis at 0th and 3rd positions and right parenthesis at 7th and 10th positions. The parenthesis 0th and 10th positions are matches, while the parenthesis 3rd and 7th positions are matches.
- Our main objective is to write a C++ program that inputs a string and outputs the pair of matched parenthesis as well as no matching parenthesis.
- If we scan an input expression fron left to right, then each right parenthesis is matched to the most recently seen unmatched left parenthesis.
- This motivates us to save the left parenthesis position on stack (because of left to right scan). If a right parenthesis is occurred, it is matched to left parenthesis at the top of the stack. Matched left parenthesis is deleted from stack.

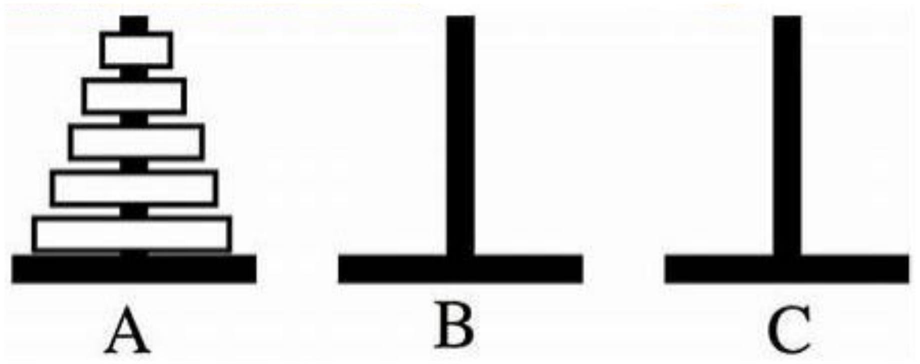
Algorithm for Parenthesis Matching

```
while(not end of line)
  {
          read the current character;
     if(No parenthesis at this place)
          continue;
     if(it is a left parenthesis)
          push it on to the stack;          //may or may not get overflow error.
     if(it is a right parenthesis)
           {
                pop the stack;      //may or may not get underflow error.
                  if (poped left bracket doesnot match the current right bracket)
                    {
                      report malformed expression error;
                      exit;
                    }
           }
    }
```

Assignment:

- Write a C++ code of your own to match the parenthesis in an expression

Tower of Hanoi Problem



This problem is fashioned after the ancient Tower of Brahma ritual. There was a diamond tower (tower A) with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size from bottom to top.
Next to this tower are 2 other diamond towers (towers B and C). Since the time of creation, attempting to move the disks from tower A to B, using C for intermediate storage.

- As the disks are very heavy, they can be moved only one at a time and also at no time can a disk be on the top of a smaller disk.
- A very elegant solution results from the use of recursion. To get the largest disk to the bottom of tower B, we move the remaining n-1 disks to C, then move the largest to tower B.
- Now there are n-1 disks to be moved from C to B. To perform this task, we can use tower A and B. We can safely ignore the fact that tower B has a disk on it as this disk is larger than disks being moved from tower C. Hence, we can place any disk on top of the disk on tower B.

Algorithm for Tower of Hanoi

```
void towerofhanoi(int n, int x, int y, int z)
{
    if(n>0)
    {
        towerofhanoi(n-1, x, z, y)
        cout<<"Move top disk from tower" << x << to top of tower"<<y<<endl;
        towerofhanoi(n-1, z, y, x);
    }
}
```

Infix, Postfix and Prefix notations

- Stacks can be used to implement algorithms involving Infix, postfix and prefix expressions
- Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.
- Ex:

Infix	A+B
Postfix	AB+
Prefix	+ AB

- Rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules.

- For example, the usual rules for associativity say that we perform operations from left to right, usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

Symbol	Priority
^	Highest
* /	Next Highest
+ -	lowest

Why postfix and prefix ?

- Prefix and Postfix expressions are easier for a computer to understand and evaluate.
- Infix expressions are readable and solvable by humans because of easily distinguishable order of operators,but compiler doesn't have integrated order of operators.
- Hence to solve the Infix Expression compiler will scan the expression multiple times to solve the sub-expressions in expressions orderly which is very in-efficient.
- To avoid this traversing, Infix expressions are converted to Postfix expression before evaluation.

1. Infix notation: X + Y

- Operators are written in-between their operands. This is the usual way we write expressions.
- An expression such as A * (B + C) / D is "First add B and C together, then multiply the result by A, then divide by D to give the final answer.
- Infix notation needs extra information to make the order of evaluation of the operators clear.

2 Postfix notation : X Y +

- Operators are written after their operands. The infix expression given above is equivalent to A B C + * D /
- The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.
- Operators act on values immediately to the left of them.

For example, the "+" uses the "B" and "C". We can add brackets to make this explicit ((A (B C +) *) D /)

- Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

3. Prefix notation : + X Y

- Operators are written before their operands. The expressions given above are equivalent to / * A + B C D
- As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right.
- I have again added (totally unnecessary) brackets to make this clear (/ (* A (+ B C)) D)

Some Examples

INFIX	PREFIX	POSTFIX
A + B	+ A B	A B +
A + B – C	– + A B C	A B + C –
(A + B) * C – D	– * + A B C D	A B + C * D –

Conversion from Infix to Postfix expression

- To convert Infix expression to Postfix expression, we use the stack data structure.
- By scanning the infix expression from left to right,if we get any operand, simply add it to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Algorithm

Step 1 : Scan the Infix Expression from left to right.

Step 2 : If the scanned character is an operand, append it with final Infix to Postfix string.

Step 3 : Else,

Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a ‘(‘ or ‘[‘ or ‘{‘), push it on stack.

Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Step 4 : If the scanned character is an ‘(‘ or ‘[‘ or ‘{‘, push it to the stack.

Step 5 : If the scanned character is an ‘)’or ‘]’ or ‘}’, pop the stack and and output it until a ‘(‘ or ‘[‘ or ‘{‘ respectively is encountered, and discard both the parenthesis.

Step 6 : Repeat steps 2-6 until infix expression is scanned.

Step 7 : Print the output

Step 8 : Pop and output from the stack until it is not empty.

Conversion from Infix to Prefix expression

We use the same algorithm to convert Infix to Prefix.

- Step 1: Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.
- Step 2: Obtain the postfix expression of the modified expression i.e CB*A+.
- Step 3: Reverse the postfix expression. Hence in our example prefix is +A*BC.

Convert Postfix Expression into Infix Notation

- Scan the Postfix String from Left to Right.
- If the character is an Operand, then Push it on to the Stack.
- If the character is an Operator, then Pop Operator 1 and Operand 2 and concatenate them using Infix notation where the Operator is in between the Two Operands.
- The resultant expression is then pushed on the Stack
- Repeat the steps

Convert Prefix Expression into Infix Notation

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
- Create a string by concatenating the two operands and the operator between them.
- string = (operand1 + operator + operand2)
- And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

Conversion from Postfix to Prefix expression

We can convert postfix to infix and then convert from infix to pprefix or we can convert directly from postfix to prefix directly as follows,

- Read the Postfix expression from left to right
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
- Create a string by concatenating the two operands after the operator.
- String = operator+ operand1 + operand2
- And push the resultant string back to Stack
- Repeat the above steps until end of Postfix expression.

Conversion from Prefix to Postfix expression

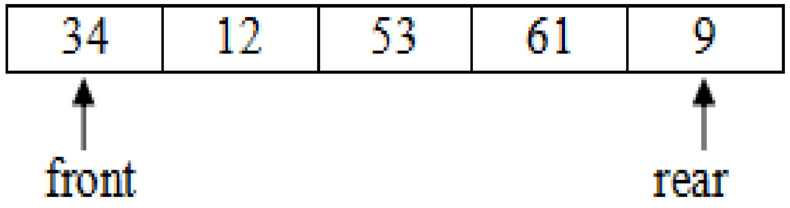
We can convert prefix to infix and then convert from infix to postfix or we can convert directly from prefix to postfix directly as follows,

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
- Create a string by concatenating the two operands and the operator after them.
- string = operand1 + operand2 + operator
- And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

QUEUES

- Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end.
- The end at which the deletion of an element take place is called front and the end at which insertion of a new element can take place is called rear.
- The first element that gets added into the queue is the first one to get removed from the list. Hence, queue is also referred as First-In-First-Out (FIFO) list.

Consider a queue of bus stop. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. The first person in the line is the first person to leave.



In the above figure 34 is the first element 9 is the last element added to the queue. Similarly, 34 would be the first element to get removed and the 9 would be the last element to get removed.

Sequential Representation

- Queue, being a linear data structure can be sequentially represented in various ways such as arrays and linked lists.
- An array is a data structure that can store a fixed number of elements. The size of the array should be fixed before using it.
- Queue on the other hand keeps on changing as we remove element from the front end. Or add new element at the end. Declaring an array with maximum size would solve this problem.

Types of Queues

The different types of queues are

1. Ordinary Queue **2. Circular Queue**

3. Double ended Queue (Dequeue) **4. Priority Queue**

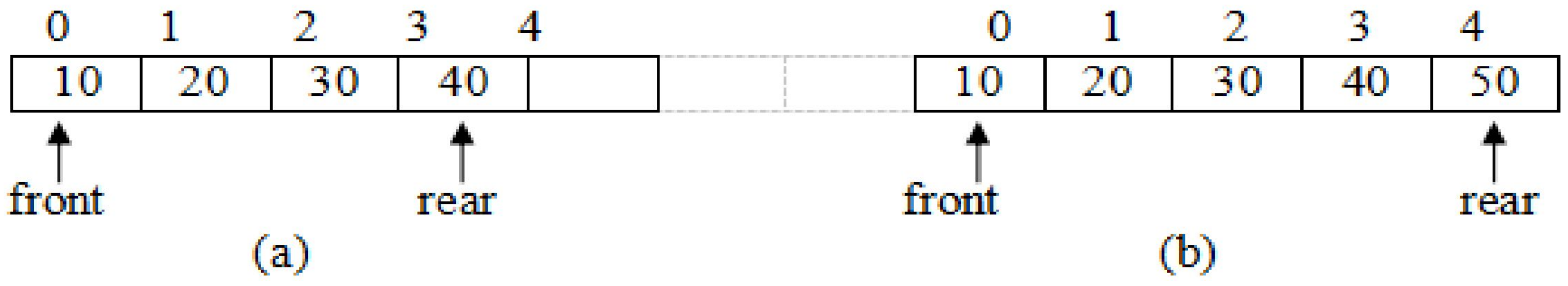
<u>Queues and their operations:</u>

An ordinary queue operates on first come first serve basis. Items will be inserted from one end and they are deleted at the other end in the same order in which they are inserted. Here first element inserted is the first element to go out of the queue. The operations that can be performed on these queues are

- An element at the rear end
- Delete an element at the front end
- Display the contents of the queue

Insert an element at the rear end

- Consider a queue, with Q_SIZE as 5 and assume 4 items are present as shown in fig (a). Here, two variable f and r are used to access the elements located at the front end and rear end respectively.
- It is clear from this figure that at most 5 elements can be inserted into the queue. Any new item should be inserted from the rear end of queue. So, if an item 50 has to be inserted, it has to be inserted to the right of item 40, i.e., at q[4].



- It is possible if increment r by 1 so as to point to next location and insert the item 50.
- The queue after inserting an item to is shown in fig (b). When queue is full, it is not possible to insert any element into queue and this condition is called overflow i.e.,

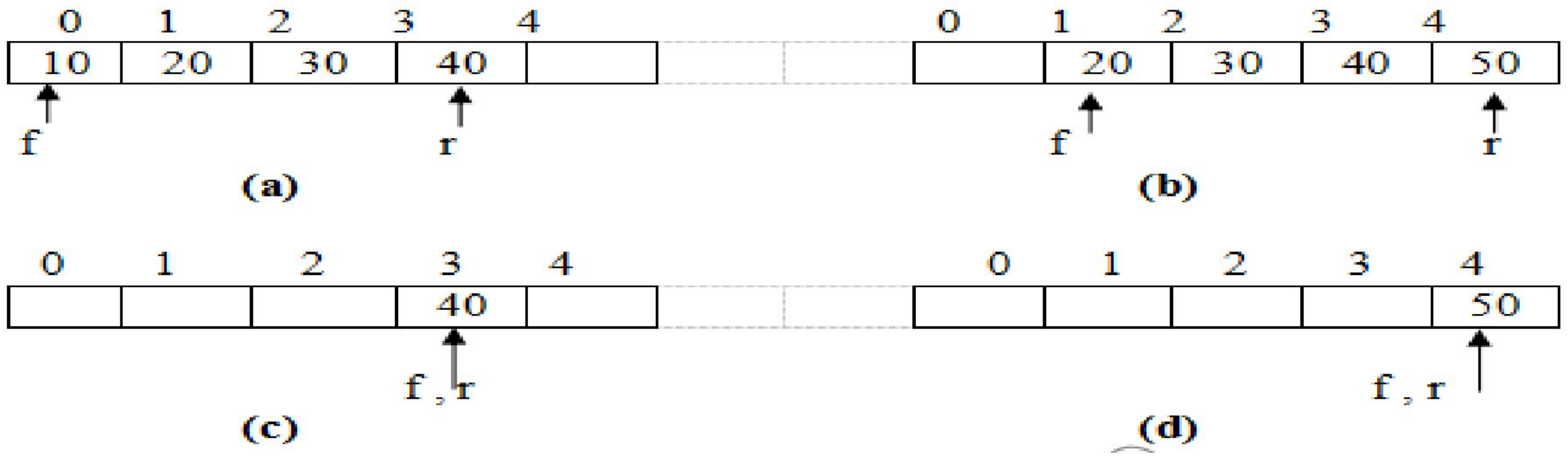
  when r = = Q_SIZE – 1, queue becomes full.
- It can be written as:

```
void insert_rear (int item, int q[], int r)
{
      if(r= =Q_SIZE-1)
     {
        cout<<“Queue overflow\n”;
          return;
      }
     q[++r] = item;
}
```

Delete an element at the front end

The first item to be deleted from the queue is the item, which is at the front end of the queue. It is clear from the queue shown in fig (a) that the first item to be deleted is 10.

Once this item is deleted, next item i.e., 20 in the queue will be the first element and the resulting queue is of the form shown in the fig (b).



- So, the variable f should point to 20 indicating that 20 is at the front of the queue. This can be achieved by accessing the item 10 first and then incrementing the variable f.
- But, as we keep on deleting one after the other, finally queue will be empty. Consider the queue shown in (c). Once the item 40 is deleted, f points to next location and queue is empty. This condition is called underflow of queue. It can be written as:

```
void delete_front (int q[], int f, int r)
{
        if(f > r)
        {
        cout<<“Queue underflow\n”;
        return;
        }
        cout<<“The element deleted is ”<< q[f++] ;
        if(f > r)
        {
        f = 0, r = – 1 ;
        }
}
```

- Consider the situation shown in fig (d)(prevoius slide). Deleting an item 50 from queue results in an empty queue.
- At this stage, suppose an item has to be inserted. While inserting when queue rear pointer r reaches Q_SIZE – 1 it displays the message “Queue overflow”.
- It is clear from the queue that queue is not full and even then, an item cannot be inserted.
- Hence, whenever queue is empty, we reset the front end identified by f to 0 and rear end identified by r to –1. So, the initial values of front pointer f and rear pointer r should be 0 and –1 respectively.

Display Queue contents

The contents of queue can be displayed only if queue is not empty. If queue is empty an appropriate message is displayed.

```
void display (int q[], int f, int r)
      {
           int i;
            if(f > r)
            {
              cout<<“Queue is empty\n”;
               return;
            }
          cout<<“Contents of the queue is\n”;
          for(i = f; i<=r; i++)
           cout<<q[i];
            }
```