UNIT -II

Stacks and Queues: Templates in C++, Template Functions- Using Templates to Represent Container Classes, The Stack Abstract Data Type, The Queue Abstract Data Type, Subtyping and Inheritance in C++, Evaluation of Expressions, Expression- Postfix Notation- Infix to Postfix.

***

TEMPLATES

Template is a new feature added in C++ that allows us to define generic classes and functions and thus provides support for generic programming. Generic programming is an approach where generic data types are used as parameters in algorithm so that they work for a variety of suitable data types and data structures. A template can be used to create a family of classes or functions.

CLASS TEMPLATES

The general format of defining a class template is:

Syntax:

```
template <class T>
class ClassName
{
        - - - - - -
        - - - - - -
        - - - - - -
        - - - - - -
};
```

A class created from a class template is called **template class**. The process of creating a specific class from a class template is called **instantiation.**

The general format of defining an object of a template class is:

Syntax: **ClassName<Type> ObjectName;**

// Example program for Class Template

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

template<class T>
class Array
{
        T K[10];
        int N,i;
        public:void ReadData()
            {
                cout<<"Enter How Many Elements = ";
                cin>>N;
                cout<<"Enter "<<N<<" Elements = ";
                for(i=0;i<N;i++)
                cin>>K[i];
            }
            void Display()
            {
                cout<<"Array Elements Are = ";
                for(i=0;i<N;i++)
                cout<<"    "<<K[i];
            }
};


void main()
{
        Array<int> obj;
        clrscr();
        obj.ReadData();
        obj.Display();
}
```

CLASS TEMPLATES WITH MULTIPLE PARAMETERS

It is also possible to use more than one generic data type in a class template. In such cases, those are separated with comma operator. The general form of defining class templates with multiple parameters is:

Syntax:

```
template<class T1 , class T2 , - - - - - - >
class ClassName
{
        - - - - - -
        - - - - - -
        - - - - - -
        - - - - - -
};
```

// Example Program for Class template with Multiple parameters

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

template<class T1,class T2>
class Test
{
        T1 A;
        T2 B;
        public:Test(T1 x,T2 y)
            {
                A=x;
                B=y;
            }
            void Display()
            {
                cout<<endl<<"Value 1 = "<<A;
                cout<<endl<<"Value 2 = "<<B;
            }
};
void main()
{
        Test<int,float> obj(12,45.67);
        clrscr();
        obj.Display();
}
```

FUNCTION TEMPLATES

Like class templates, it is also possible to define function templates that could be used to create a family of functions with different argument types. The general format of defining a function template is:

Syntax:

```
template<class T>
ReturnType  FunctionName (Arguments of Type T)
{
        - - - - - -
        - - - - - -
        - - - - - -
        - - - - - -
}
```

The function template syntax is similar to that of the class template except that we are defining function instead of classes. A function generated from a function template is called a template function.

// Example program for Function Template

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

template<class T>
void swap(T &a, T &b)
{
        T Temp;
        Temp=a;
        a=b;
        b=Temp;
}
void main()
{
        int x,y;
        clrscr();
        cout<<endl<<"Enter Two Values = ";
        cin>>x>>y;
        cout<<endl<<"Original Values Are ="<<x<<"    "<<y;
        swap(x,y);
        cout<<endl<<"Interchange Values Are ="<<x<<"    "<<y;
}
```

FUNCTION TEMPLATES WITH MULTIPLE PARAMETERS

It is also possible to use more than one generic data type in the template statement. In such cases, those are separated with comma operator. The general form of defining funcion templates with multiple parameters is:

Syntax:

```
template<class T1 , class T2 , - - - - - - >
ReturnType  FunctionName (Arguments of Type T1, T2 ,  - - - )
{
        - - - - - -
        - - - - - -
        - - - - - -
        - - - - - -
}
```

// Example Program for Function template with Multiple parameters

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

template<class T1,class T2>
void show(T1 x,T2 y)
{
        cout<<endl<<"X Value ="<<x;
        cout<<endl<<"Y Value ="<<y;
}

void main()
{
        int p=74;
        double q=9.078;
        clrscr();
        show(p,q);
}
```

NON-TYPE TEMPLATE ARGUMENTS

In case of class template with multiple parameters, it is also possible to use non-type arguments. That is, in addition to type argument T, use other arguments types such as int, char, float, double, function names, constant expression.

// Example program for Non-Type Template Arguments

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

template<class T,int size>
class Array
{
        T K[size];
        int N,i;
        public:void ReadData()
            {
                cout<<"Enter How Many Elements =";
                cin>>N;
                cout<<"Enter "<<N<<" Values =";
                for(i=0;i<N;i++)
                cin>>K[i];
            }
            void Display()
            {
                cout<<endl<<"Array Elements Are =";
                for(i=0;i<N;i++)
                cout<<"     "<<K[i];
            }
};
void main()
{
        Array<int,25> obj;
        clrscr();
        obj.ReadData();
        obj.Display();
}
```

MEMBER FUNCTION TEMPLATES

In a class template, all the member functions were defined as inline was not necessary. It is possible to define their definition out side the class declaration. The member functions of the template classes themselves are parameterized by the type argument and there fore these functions must be defined by the function templates.

The general form of defining member function template is:

Syntax:

```
template<class T>
ReturnType ClassName < T > :: FunctionName (Argument List)
{
        - - - - - - -
        - - - - - - -
}
```

// Example program for Member Function Template

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

template<class T>
class Array
{
        T K[25];
        int N;
        public:void ReadData();
            void Display();
};

template<class T>
void Array<T>::ReadData()
{
  cout<<"Enter How Many Elements =";
  cin>>N;
  cout<<"Enter "<<N<<" Values =";
  for(int i=0;i<N;i++)
  cin>>K[i];
}

template<class T>
void Array<T>::Display()
{
  cout<<endl<<"Array Elements Are =";
  for(int i=0;i<N;i++)
  cout<<"      "<<K[i];
}



void main()
{
        Array<int> obj;
        clrscr();
        obj.ReadData();
        obj.Display();
}
```

CONTAINER CLASS

It is possible to create an object of one class into another and that object will be a member of the class. This type of relationship between classes is known as **containership** or **has_a** relationship as one class contain the object of another class.

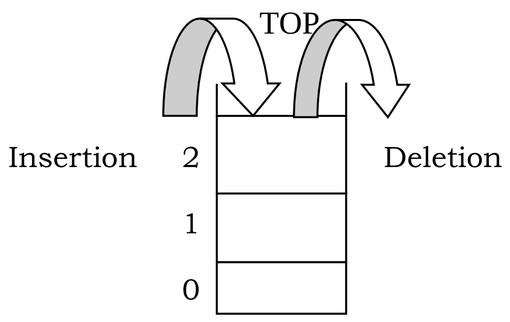
The class which contains the object and members of another class in this kind of relationship is called a **container class.** The object that is a part of another object is called **contained object**, whereas object that contains another object as its part of attribute is called **container object.**

// Example program for Container Class

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

class Date
{
   int Day,Month,Year;
   public:Date()
            {
            }
            Date(int d,int m ,int y)
            {
                   Day = d;
                   Month = m;
                   Year = y;
            }
            void display()
            {
                   cout<<"JOIN DATE = "<<Day<<" - "<<Month<<" - "<<Year;
            }
};
class Employee                          // Container class
{
   int Id;
   int BasicSal;
   Date Bd;                             // Contained Object
   public:Employee(int i, int sal, int d, int m, int y)
            {
                   Id = i;
                   BasicSal = sal;
                   Bd = Date(d,m,y);
            }
            void display()
            {
                   cout <<"Employee ID  : " <<Id << endl;
                   cout <<"Basic Salary : " <<BasicSal << endl;
                   Bd.display();
            }
};

int main()
{
         Employee St(2,20000,7,11,1999);              // Container Object
         clrscr();
         St.display();
         return 0;
}
```

STACK

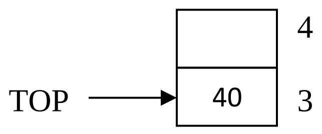
A stack is a linear data structure in which an element may be inserted or deleted only at one end, called the **TOP** of the list. The diagrammatic representation of a stack is as follows:



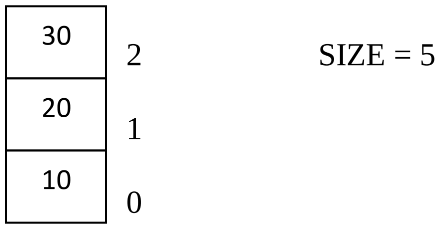
STACK

Consider the elements as: 10 20 30 40

When these elements are inserted into a stack of size 5, then
Insertion order is: 10 20 30 40



Status of Stack:



Now, same elements are deleted from the stack, then

Deletion order is: 40 30 20 10

Here, the order of deleted elements from the stack is reverse order in which they were inserted into the stack. Hence, a stack is also known as a **LIFO** (Last-In-First-Out) list. Since, the last inserted element is the first out coming element of the list.

Examples: Stack of trays, Stack of plates, Letter basket etc.,

REPRESENTATIONS OF STACK

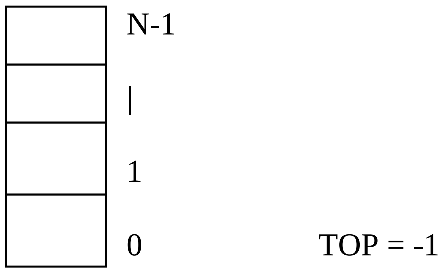
Stacks may be represented in the memory in two ways. Those are:

a) Array representation (Static representation)
b) Linked representation (Dynamic representation)

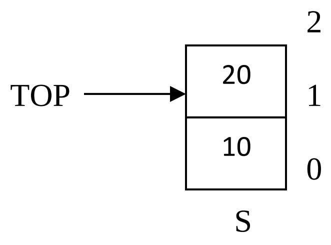
a) Array Representation

Static representation of a stack uses **array** concept. In this case, assume S is an array used to represent a stack with a maximum size as 'N'. Initially no elements are available in the stack S. As per C language array index starts from 0th location. At this stage, a variable TOP is set at -1 position. Then, the status of the stack is as follows:

Status of Stack:



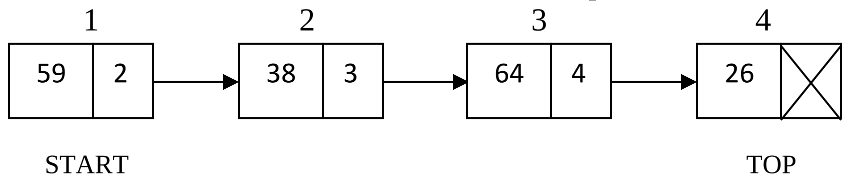
Example: Assume N = 3 and two elements 10 and 20 are inserted into the stack, and then the status of stack is:



b) Linked Representation

Dynamic representation of a stack uses **linked list** concept. In general, single linked list structure is sufficient to represent any stack.

In linked representation of a stack, each node consists of two parts namely **DATA** field and **LINK** field. **DATA** field contains information part of the element and **LINK** field contains address of the next node of the list. Then the linked representation of a stack is as follows:



OPERATIONS ON STACKS

Basic operations on the stack are: push, pop, peek, empty, full, display etc.

Stack Abstrat Data Type: The specification of a Stack ADT can be shown as:

AbstractDataType Stack
{

Instances: Linear collection of elements

Operations:

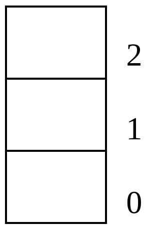
push(Item):	Inserts a new element Item at TOP position of the Stack.
pop():	Remove the TOP position element from the Stack.
peek():	Return the TOP position element of the Stack.
empty():	Return **true** if the Stack is empty, otherwise return **false**.
full():	Return **true** if the Stack is full, otherwise return **false**.
display():	Print elements of the Stack.

}

For implementing these operations, consider ***array representation*** of the stack. Assume S is an array used to perform stack operations with a maximum size as ‘N’. Initially elements are not available in the stack S. At this stage, set the TOP variable at -1 position of the stack S.

Example: Let N = 3

Status of Stack:



TOP = -1

Push Operation:

The process of inserting an element into the stack is known as push operation.

At every push operation, first the variable TOP is incremented by 1 and then the new element is inserted into top position of the stack.

While performing push operation, if empty locations are not available to insert the element then the status of the stack is called as "**STACK OVERFLOW**" or "**STACK FULL**".

Algorithm push(x): This procedure inserts a new element x into top position of the stack S.

```
Step 1:     IF TOP = N-1 THEN
                WRITE 'STACK OVERFLOW'
                RETURN
            ENDIF
Step 2:     TOP ← TOP + 1
Step 3:     S[TOP] ← x
Step 4:     RETURN
```

Pop Operation:

The process of deleting an element from the stack is known as pop operation.

At every pop operation, first delete the element from top position of the stack and then the variable TOP is decremented 1.

While performing pop operation, if elements are not available to delete from the stack then the status of the stack is called as "**STACK UNDERFLOW**" or "**STACK EMPTY**".

Algorithm pop(): Function is used to delete the topmost element from the stack S.

```
Step 1:     IF TOP = -1 THEN
```

```
                    WRITE ‘STACK UNDERFLOW’
                    RETURN -1
                ENDIF
Step 2:         K ← S[TOP]
Step 3:         TOP ← TOP - 1
Step 4:         RETURN   K
```

Peek Operation:

The process of printing the topmost element of the stack is known as peek operation.

Algorithm peek(): Function returns the top element from the stack S if exists; otherwise, it returns -1.

```
Step 1:         IF   TOP = -1   THEN
                    RETURN   -1
                ELSE
                    RETURN    S[TOP]
                ENDIF
```

Empty Operation:

Empty operation is used to check whether the stack is empty or not.

Algorithm empty(): Function returns 1 if the stack is empty; otherwise, it returns 0.

```
Step 1:         IF   TOP = -1   THEN
                    RETURN   1
                ELSE
                    RETURN   0
                ENDIF
```

Full Operation:

Full operation is used to check whether the stack is full with elements or not.

Algorithm full(): Function returns 1 if the stack is full; otherwise, it returns 0.

```
Step 1:         IF   TOP = N-1   THEN
```

```
                RETURN   1
        ELSE
                RETURN   0
        ENDIF
```

Display Operation:

Display operation is used to print elements of the stack S.

Algorithm display(): Function prints elements of the stack S.

```
Step 1:     IF TOP = -1 THEN
                WRITE 'STACK EMPTY'
            ELSE
                REPEAT FOR i ← 0 TO TOP DO STEPS BY 1
                    WRITE S[i]
                ENDREPEAT
            ENDIF
Step 2:     RETURN
```

APPLICATIONS OF STACK

Stack operations are utilized by the computer system to do various tasks. Some of the important application areas are:

1) Evaluation of arithmetic expressions
2) Code generation for stack machines
3) Implementation of recursion
4) Record management etc.

Evaluation of Arithmetic Expressions

An arithmetic expression is formed with the combination of arithmetic operators and operands.

Example: $2x + 3y = 74$

Any arithmetic expression can be represented in three ways. Those are:

- Infix notation
- Prefix notation

- Postfix notation.

Infix Notation: In infix notation, operators are placed in between the operands of the mathematical expression. In general ordinary mathematical expressions are represented in terms of infix expressions.

Example: A+B
2-7+4/3
(14+25) / (9+2-4) etc.

Prefix Notation: In prefix notation, operators are placed before the operands of the mathematical expression. Prefix notation is also known as **polish** notation.

Example: +AB
- + 2 14 9 etc.

Postfix Notation: In postfix notation, operators are placed after the operands of the mathematical expression. Postfix notation is also known as **reverse polish** notation.

Example: AB+
5 6 2 + * 12 4 / - etc.

Computer system evaluates the given arithmetic expression in two steps. First it converts the given infix expression into postfix expression and then evaluates the postfix expression. In both stages, stack is the important tool used to perform the task.

A) Conversion of Infix Expression into Postfix Expression

Let Q is an arithmetic expression written in infix notation consists of the operators + (Addition), - (Subtraction), * (Multiplication), / (Division), ^ (Power), left and right parentheses. For this assume the priority order as:

OPERATOR	PRIORITY
^	3
* /	2
+ -	1

Then the following procedure converts the given infix expression Q into its equivalent postfix expression P.

Step 1: Push "(" onto the STACK, and add ")" to the end of Q.

Step 2: Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:

Step 3: If an **<u>operand</u>** is encountered, add it to P.

Step 4: If a **<u>left parenthesis</u>** is encountered, push it onto the STACK.

Step 5: If an **<u>operator</u>** © is encountered, then:

a) Repeatedly pop elements from the STACK and add to P each operator which has the same precedence as or higher precedence than ©.

b) Add © to the STACK.

Step 6: If a **<u>right parenthesis</u>** is encountered, then:

a) Repeatedly pop elements from the STACK and add to P each operator until a left parenthesis is encountered.

b) Remove the left parenthesis.

Step 7: EXIT

The above procedure transforms the infix expression Q into its equivalent postfix expression P. It uses a stack to temporarily hold operators and left parenthesis.

The postfix expression P will be constructed from left to right using the operands from Q and the operators which are removed from the stack. Procedure starts by pushing a left parenthesis onto stack and adding a right parenthesis at the end of Q. The procedure is completed when the stack is empty.

<u>Example:</u> ***Convert the expression A + B into postfix expression.***

<u>Solution:</u> Given infix expression Q = A + B)

Scanned Element	STACK	Postfix Notation P	Remarks
-	(	-	-
A	(	A	-
+	(+	A	+ = 1 (= 0 0>=1 FALSE
B	(+	A B	-
)	EMPTY	A B +	-

Resultant Postfix Expression = A B +

B) Evaluation of Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following procedure evaluates the expression P and uses a STACK to hold operands.

Procedure:

Step 1: Add a right parenthesis “)” at the end of P.

Step 2: Scan P from left to right and repeat Steps 3 and 4 for each element of P until the right parenthesis is encountered.

Step 3: If an **operand** is encountered, put it on the STACK.

Step 4: If an **operator** © is encountered, then:

a) Remove the two top elements of the STACK, where A is the top element and B is the next-to-top element.
b) Evaluate B © A.
c) Place the result of (b) back on the STACK.

Step 5: Set result VALUE equal to the top element of the STACK.

Step 6: EXIT.

Example: ***Evaluate the postfix expression 5 6 2 + * 12 4 / -***

Solution: Given Postfix Expression P = 5 6 2 + * 12 4 / -)

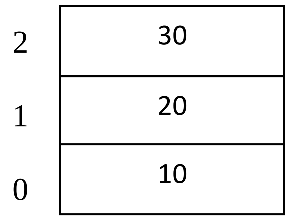
Scanned Element	STACK	Operation
5	5	-
6	5 6	-
2	5 6 2	-
+	5 8	A = 2 , B = 6 6+2 = 8
*	40	A = 8 , B = 5 5*8 = 40
12	40 12	-
4	40 12 4	-

/	40 3	A = 4 , B = 12 12/4 = 3
-	37	A = 3 , B = 40 40-3 = 37
)	**37 = Result Value**	-

Resultant Postfix Expression Value = 37

Stack Disadvantages:

Consider the status of the stack is as:



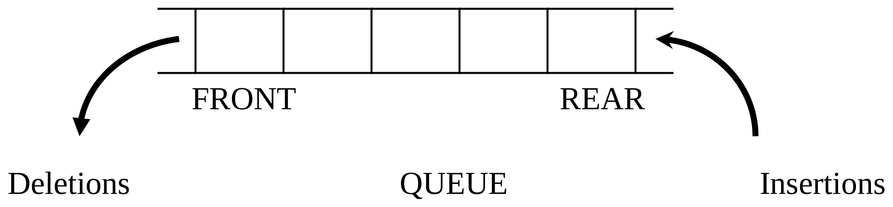
STACK

Now, if we want to delete the first inserted element 10 direct deletion is not possible. First delete the element 30, then 20 and then the actual required element 10. After deletion the actual element 10 again inserts the previous deleted elements 20 and 30 into the stack. It's a time consuming process.

***

QUEUES

A queue is a linear data structure in which elements can be inserted only at one end, called **REAR** end and elements can be deleted from the other end, called **FRONT** end of the list. The diagrammatic representation of a queue is as follows:

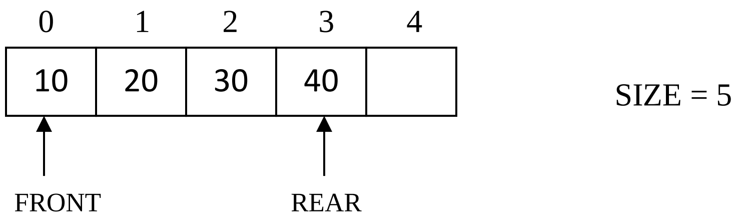


Consider the elements as: 10 20 30 40

When these elements are inserted into a queue of size 5, then

Insertion order is: 10 20 30 40

Status of Queue:



Now, same elements are deleted from the queue, then

Deletion order is: 10 20 30 40

Here, the order of deleted elements from the queue is the same order in which they were inserted into the queue. Hence, a queue is also known as a **FIFO** (First-In-First-Out) list. Since, the first inserted element is the first out coming element of the list.

Examples: Ticket issuing counter process
Printing multiple programs on a printer etc.

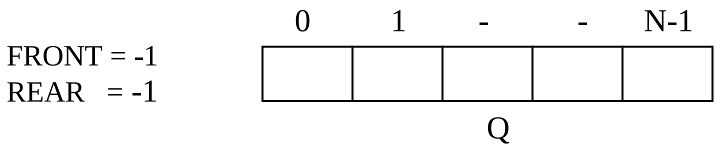
REPRESENTATIONS OF QUEUES

Queues may be represented in the memory in two ways. Those are:

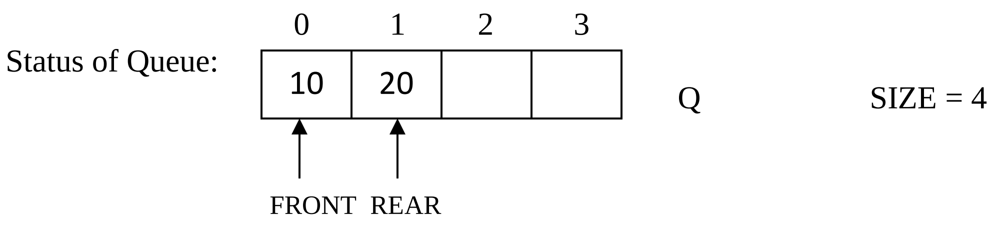
a) Array representation (Static representation)
b) Linked representation (Dynamic representation)

a) Array Representation

Static representation of a queue uses **array** concept. In this case, assume Q is an array used to represent a queue with a maximum size as 'N'. Initially no elements are available in the queue Q. At this stage, two variables FRONT and REAR set at -1 position. Then, the status of the queue is as follows:



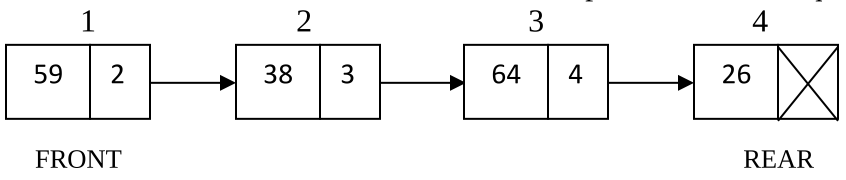
Example: Assume N = 4 and two elements 10 and 20 are inserted into the queue, and then the status of queue is:



b) Linked Representation

Dynamic representation of a queue uses **linked list** concept. In general, single linked list structure is sufficient to represent any queue.

In linked representation of a queue, each node consists of two parts namely DATA field and LINK field. DATA field contains information part of the element and LINK field contains address of the next node of the list. Then the linked representation of a queue is as follows:



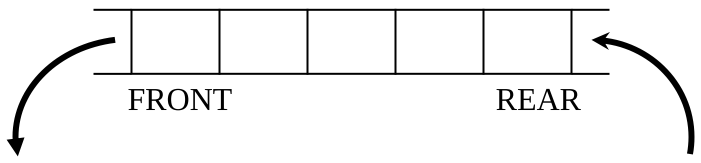
TYPES OF QUEUES

Depending on the way of performing operations on the queue data structure, queues can be classified into four types as:

a) Linear Queue
b) Circular Queue
c) Deque
d) Priority Queue

a) LINEAR QUEUE

A linear queue is a linear data structure in which elements can be inserted only at one end, called **REAR** end and elements can be deleted from the other end, called **FRONT** end of the list. The diagrammatic representation of a linear queue is as follows:



Deletions QUEUE Insertions

OPERATIONS ON LINEAR QUEUE

Basic operations on the queue are: enqueue, dequeue, front element, rear element, empty, full, display etc.

AbstractDataType Queue
{

Instances: Linear collection of elements

Operations:

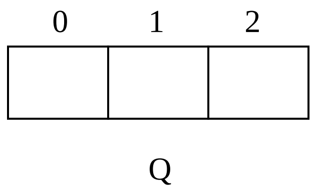
enqueue(Item):	Inserts a new element Item at REAR position of the Queue.
dequeue():	Remove the FRONT position element from the Queue.
frontelement():	Return the FRONT position element of the Queue.
rearelement():	Return the REAR position element of the Queue.
empty():	Return **true** if the Queue is empty, otherwise return **false**.
full():	Return **true** if the Queue is full, otherwise return **false**.
display():	Print elements of the Queue.

}

For implementing these operations, consider ***array representation*** of the queue. Assume Q is an array used to perform queue operations with a maximum size as 'N'. Initially elements are not available in the queue Q. At this stage, set two variables F and R at -1 that represents FRONT and REAR ends of the list.

Example: Let N = 3

Status of Queue:



F = -1
R = -1

Enqueue Operation:

The process of inserting an element into the queue is known as enqueue operation.

For every enqueue operation, first the variable REAR is incremented by 1 and then the new element is inserted into REAR position of the queue.

While performing enqueue operation, if empty locations are not available to insert the element then the status of the queue is called as "**QUEUE OVERFLOW**" or "**QUEUE FULL**".

Algorithm enqueue(x): This procedure inserts a new element x into REAR position of the queue Q.

```
Step 1:     IF R = N-1 THEN
                WRITE  'LINEAR QUEUE OVERFLOW'
                RETURN
            ENDIF
Step 2:     R ← R + 1
            Q[R] ← x
Step 3:     IF  F = -1  THEN
                F ← 0
            ENDIF
Step 4:     RETURN
```

Dequeue Operation:

The process of deleting an element from the queue is known as dequeue operation.

For every dequeue operation, first delete an element from FRONT position of the queue and then the variable FRONT is incremented 1.

While performing dequeue operation, if elements are not available to delete from the queue then the status of the queue is called as "**QUEUE UNDERFLOW**" or "**QUEUE EMPTY**".

Algorithm dequeue(): Function is used to delete the FRONT position element from the queue Q.

```
Step 1:     IF  F = -1  THEN
                WRITE  'LINEAR QUEUE UNDERFLOW'
                RETURN  -1
            ENDIF
Step 2:     K ← Q[F]
Step 3:     IF  F = R  THEN
                F ← R ← - 1
            ELSE
                F ← F + 1
            ENDIF
Step 4:     RETURN  K
```

Front Element Operation:

This operation is used to print FRONT position element of the Queue Q.

Algorithm felement(): Function returns the FRONT position element of the queue Q if exists; otherwise, it returns -1.

```
Step 1:     IF  F = -1  THEN
                RETURN  -1
```

```
                    ELSE
                            RETURN   Q[F]
                    ENDIF
```

Rear Element Operation: This operation is used to print REAR position element of the Queue Q.

Algorithm relement(): Function returns the REAR position element of the queue Q if exists; otherwise, it returns -1.

```
        Step 1:     IF   R = -1   THEN
                            RETURN   -1
                    ELSE
                            RETURN   Q[R]
                    ENDIF
```

Empty Operation: Empty operation is used to check whether the queue is empty or not.

Algorithm empty(): Function returns 1 if the queue is empty; otherwise, it returns 0.

```
        Step 1:     IF   F = -1   THEN
                            RETURN   1
                    ELSE
                            RETURN   0
                    ENDIF
```

Full Operation: Full operation is used to check whether the queue is full with elements or not.

Algorithm full(): Function returns 1 if the queue is full; otherwise, it returns 0.

```
        Step 1:     IF   R = N-1   THEN
                            RETURN   1
                    ELSE
                            RETURN   0
                    ENDIF
```

Display Operation: Display operation is used to print elements of the Queue Q.

Algorithm display(): Function prints elements of the queue Q.

```
        Step 1:     IF   F = -1    THEN
                            WRITE    'LINEAR QUEUE EMPTY'
                    ELSE
                            REPEAT    FOR i ← F TO R DO STEPS BY 1
                                      WRITE Q[i]
                            ENDREPEAT
                    ENDIF
        Step 2:   RETURN
```

Linear Queue Disadvantages:

Consider the following status of the linear queue as:

Example: Let N = 5

Status of Queue Q:

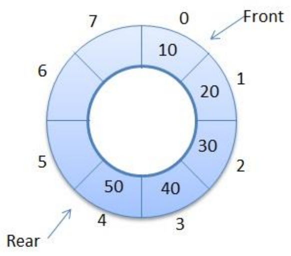
0	1	2	3	4
		30	40	50
		F		R

At this stage, if we tried to insert an element into the queue, it prints a message as "Queue Overflow" that refers to empty locations is not available. But still empty locations are available at the FRONT end of the list.

From this, Even though empty locations are available at FRONT end and REAR reached to N-1th locations then further insertion is not possible in linear queues.

b) CIRCULAR QUEUE

In circular queue, elements are arranged in circular fashion. Here, when REAR end reaches to N-1th location and still empty locations are available at the FRONT end of the list then REAR variable is set back to starting location. The logical view of a circular queue can be represented as:



Consider the basic operations on circular queue as Insertion, Deletion and Display operations.

Algorithm Insertion(x): This procedure inserts a new element x into REAR position of the circular queue CQ.

```
Step 1:     IF   (F = 0  AND  R = N-1)  OR  (F = R+1)   THEN
                 WRITE   'CIRCULAR QUEUE OVERFLOW'
                 RETURN
            ENDIF
Step 2:     IF  R = N-1   THEN
                 R ← 0
            ELSE
                 R ← R + 1
            ENDIF
Step 3:     CQ[R] ← x
Step 4:     IF   F = -1  THEN
                 F ← 0
            ENDIF
Step 5:     RETURN
```

Algorithm Deletion(): Function is used to delete the FRONT position element from the circular queue CQ.

```
Step 1:     IF   F = -1   THEN
                 WRITE    'CIRCULAR QUEUE UNDERFLOW'
                 RETURN   -1
            ENDIF
Step 2:     K ← CQ[F]
Step 3:     IF   F = R   THEN
                 F ← R ← - 1
            ELSEIF   F = N-1   THEN
                 F ← 0
            ELSE
                 F ← F + 1
            ENDIF
Step 4:     RETURN   K
```

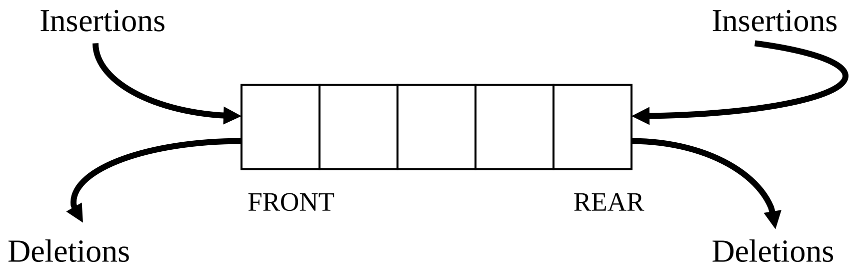
Algorithm display(): Function prints elements of the circular queue CQ.

```
Step 1:     IF   F = -1    THEN
                 WRITE    'CIRCULAR QUEUE EMPTY'
            ELSE
                 IF   F ≤ R  THEN
                      REPEAT    FOR i ← F TO R DO STEPS BY 1
                           WRITE   CQ[i]
                      ENDREPEAT
```

```
                ELSE
                    REPEAT    FOR i ← F TO N-1 DO STEPS BY 1
                        WRITE   CQ[i]
                    ENDREPEAT
                    REPEAT    FOR i ← 0 TO R DO STEPS BY 1
                        WRITE   CQ[i]
                    ENDREPEAT
                ENDIF
            ENDIF

Step 2:     RETURN
```

c) DEQUE

A deque (Double Ended Queue) is a linear data structure in which elements may be inserted and deleted at either ends of the list. The diagrammatic representation of a deque is as follows:

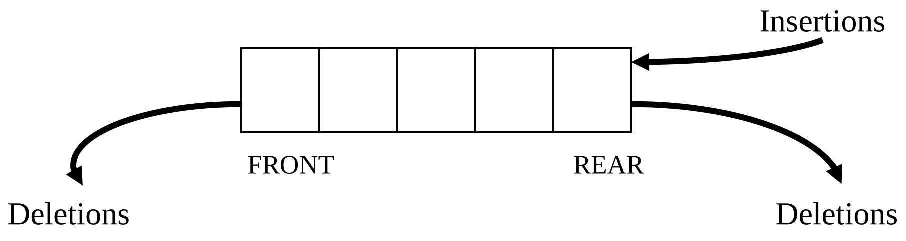


Deque can be classified into two types as:

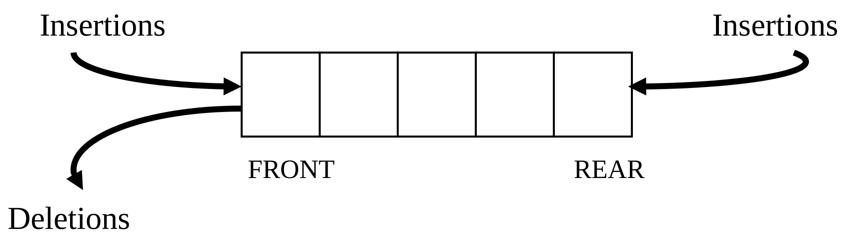
i) Input-Restricted Deque

ii) Output-Restricted Deque

***i) Input-Restricted Deque*:** An input-restricted deque is a deque which allows insertions only at one end of the list but allows deletions at both ends of the list. The diagrammatic representation of an input-restricted deque is as follows:



***ii) Output-Restricted Deque*:** An output-restricted deque is a deque which allows deletions only at one end of the list but allows insertions at both ends of the list. The diagrammatic representation of an output-restricted deque is as follows:



d) PRIORITY QUEUE

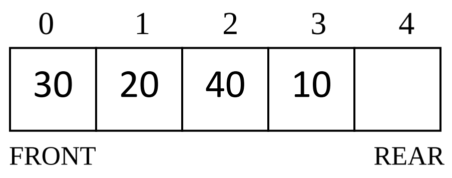
A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

- An element of higher priority is processed before any element of lower priority.
- Two elements with the same priority are processed according to the order in which they were added to the queue.

Example: Consider the following jobs/elements and their properties as:

JOB/ELEMENT	PRIORITY
10	3
20	2
30	1
40	2

Then the status of the priority queue after processing these elements is:



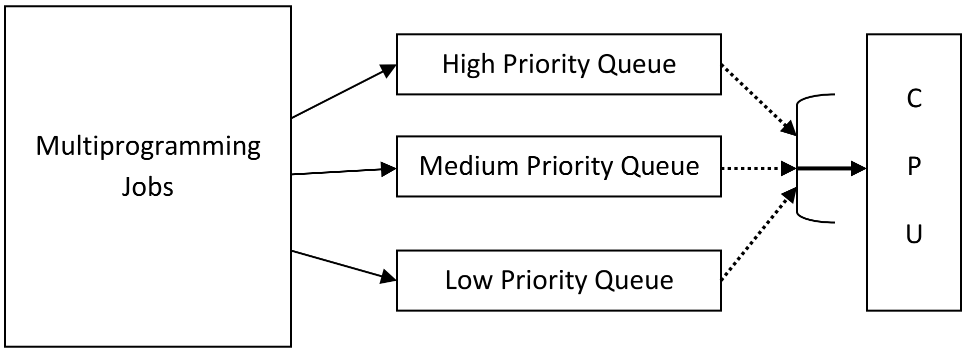
APPLICATION OF QUEUES

Queues are used in different application areas such as: simulation, multiprogramming environments, job scheduling applications etc.

CPU Scheduling in Multiprogramming Environments

In multiprogramming environment a single CPU has to serve more than one program simultaneously. In this case, scheduling is to classify the work load according to its characteristics and to maintain separate process queues. Process will assign to their respective queues. Then CPU will service the processes as per the priority of the queues.

Example:



Here, high priority programs are processed first before the medium and low priority queue jobs. After completing high priority queue program, then CPU serve its service to medium priority queue programs. Finally it serves to lowest priority programs.

***

INHERITANCE

Reusability is an important feature of object-oriented programming. C++ strongly supports the concept of reusability with inheritance. Here, a new class is derived from the exising class.

The mechanism of deriving a new class from the existing class is called **inheritance.** The old class is referred to as the **base class** and the new one is called the **derived class** or **subclass.**

Defining Derived Classes

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

Syntax:
```
class DerivedClassName : VisibilityMode BaseClassName
{
            - - - - - - - - - -
            - - - - - - - - - -
            - - - - - - - - - -
};
```
Here,

- The colon indicates that the DerivedClassName is derived from the BaseClassName.
- VisibilityMode can be either private, protected or public. The default VisibilityMode is **private.**
- When the VisibilityMode is **private**, 'publc members' of the base class become 'private members' of the derived class therefore the public members of the base class can only be accessed by the member functions of the derived class.
- When the VisibilityMode is **public**, 'publc members' of the base class become 'public members' of the derived class therefore they are accessible to the objects of the derived class.

TYPES OF INHERITANCE

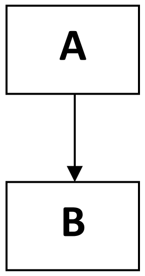
In C++, inheritance can be classified into five types such as:

a) Single Inheritance
b) Multilevel Inheritance
c) Multiple Inheritance
d) Hierarchical Inheritance
e) Hybrid Inheritance

a) SINGLE INHERITANCE

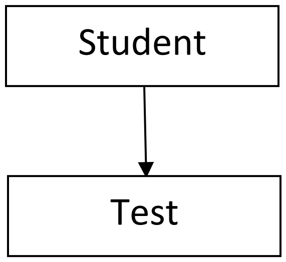
The process of deriving a sub class from only one base class is known as single inheritance.

Example:



Here, A is a base class and B is a derived class.

// Example program for Single Inheritance

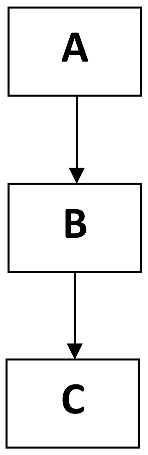


```
#include<iostream.h>
#include<conio.h>

class Student
{
        int RNO;
        public:void GetNumber()
             {
                  cout<<"Enter Roll Number =";
                  cin>>RNO;
             }
             void PutNumber()
             {
                  cout<<endl<<"Student Roll Number ="<<RNO;
             }
};

class Test:public Student
{
        int S1,S2;
        public:void GetMarks()
             {
                  cout<<endl<<"Enter Subject 1 Marks =";
                  cin>>S1;
                  cout<<endl<<"Enter Subject 2 Marks =";
                  cin>>S2;
             }
             void PutMarks()
             {
                  cout<<endl<<"Subject 1 Marks ="<<S1;
                  cout<<endl<<"Subject 2 Marks ="<<S2;
             }
};

void main()
{
        Test obj;
        clrscr();
        obj.GetNumber();
        obj.GetMarks();
        obj.PutNumber();
        obj.PutMarks();
}
```

b) MULTILEVEL INHERITANCE

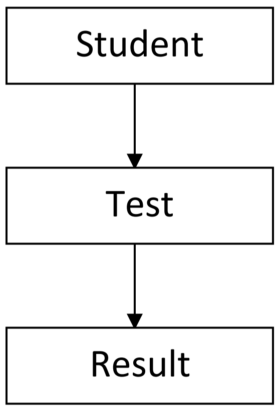
The process of deriving a sub class from an intermediate base class, which is derived from a base class, is known as multilevel inheritance.

Example:



Here,
A is a base class, B is intermediate base class and C is a derived class.

// Example program for Multilevel Inheritance

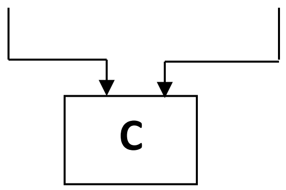


```
#include<iostream.h>
#include<conio.h>

class Student
{
        int RNO;
        public:void GetNumber()
            {
                cout<<"Enter Roll Number =";
                cin>>RNO;
            }
            void PutNumber()
            {
                cout<<endl<<"Student Roll Number ="<<RNO;
            }
};
class Test:public Student
{
        public:int S1,S2;
            void GetMarks()
            {
                cout<<endl<<"Enter Subject 1 Marks =";
                cin>>S1;
                cout<<endl<<"Enter Subject 2 Marks =";
                cin>>S2;
            }
            void PutMarks()
            {
                cout<<endl<<"Subject 1 Marks ="<<S1;
                cout<<endl<<"Subject 2 Marks ="<<S2;
            }
};
class Result:public Test
{
        int Total;
        float Avg;
        public:void ShowResult()
            {
                Total=S1+S2;
                cout<<endl<<"Total Marks ="<<Total;
                Avg=(S1+S2)/2;
                cout<<endl<<"Average Marks ="<<Avg;
            }
};
void main()
{
        Result obj;
        clrscr();
        obj.GetNumber();
        obj.GetMarks();
        obj.PutNumber();
        obj.PutMarks();
        obj.ShowResult();

}
```

c) MULTIPLE INHERITANCE

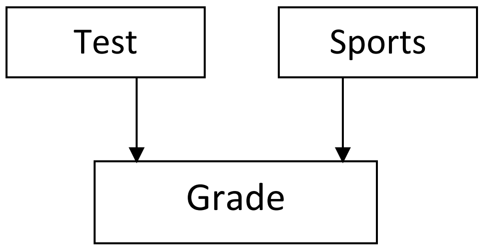
The process of deriving a sub class from more than one base class is known as multiple inheritance.

Example:



Here,
A and B are base classes and C is a derived class.

// Example program for Multiple Inheritance

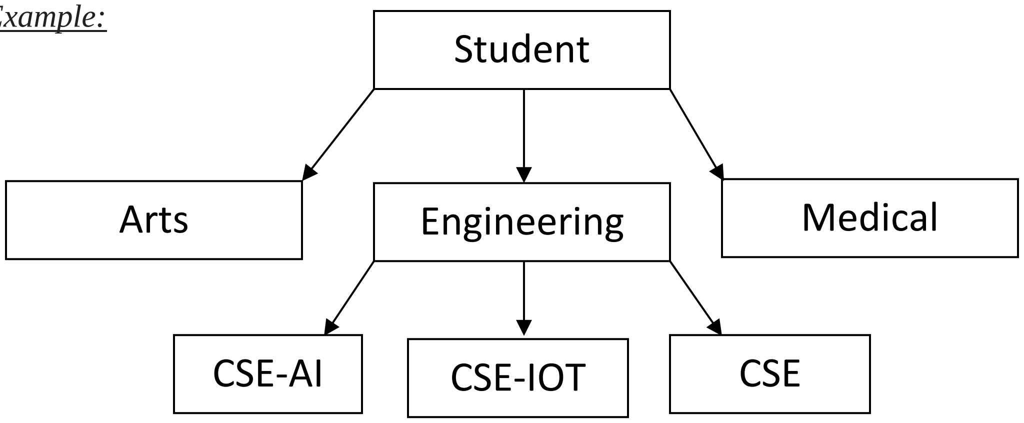


```
#include<iostream.h>
#include<conio.h>

class Test
{
        public:int RNO,S1,S2;
            void ReadData()
            {
                cout<<"Enter Roll Number =";
                cin>>RNO;
                cout<<endl<<"Enter Subject 1 Marks =";
                cin>>S1;
                cout<<endl<<"Enter Subject 2 Marks =";
                cin>>S2;
            }
            void PrintData()
            {
                cout<<endl<<"Student Roll Number ="<<RNO;
                cout<<endl<<"Subject 1 Marks ="<<S1;
                cout<<endl<<"Subject 2 Marks ="<<S2;
            }
};
class Sports
{
        public:int SMarks;
            void SportsMarks()
            {
                cout<<endl<<"Enter Sports Marks =";
                cin>>SMarks;
            }
            void PrintMarks()
            {
               cout<<endl<<"Sports Points ="<<SMarks;
            }
};
class Grade:public Test,public Sports
{
        public:int Final;
            void ResultStatus()
            {
                Final=S1+S2+SMarks;
                if(Final>=180)
                  cout<<endl<<"PASS";
                else
                  cout<<endl<<"FAIL";
            }
};
void main()
{
        Grade obj;
        clrscr();
        obj.ReadData();
        obj.SportsMarks();
        obj.PrintData();
        obj.PrintMarks();
        obj.ResultStatus();

}
```

d) HIERARCHICAL INHERITANCE

Hierarchical Inheritance in C++ refers to the type of inheritance that has a hierarchical structure of classes. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level. i.e., A **sub class** can be constructed by inheriting the properties of **base class**. A **sub class** can serve as a base class for the lower level classes and so on.

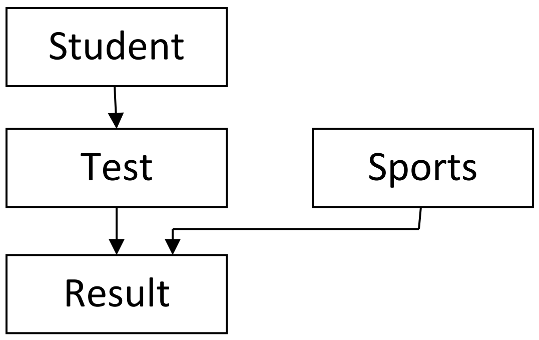
Example:



e) HYBRID INHERITANCE

Hybrid Inheritance forms with the combination of two or more types of inheritances.

Example:



VIRTUAL FUNCTION

A virtual function is a member function of a base class which is redefined by a derived class. For this, use the keyword **“virtual”.**

One very useful feature of virtual functions is creating pure virtual functions. When a virtural function is equated to zero, then the virtual function is known as **pure virtual function**.

// Example program for virtual function

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

class Company
{
        public:virtual void display( ) = 0;      // Pure Virtual Function
};

class Tata:public Company
{
        public:void display()
                {
                        cout<<endl<<"TATA SHOW ROOM";
                }
};

void main()
{
        Tata obj;
        clrscr();
        obj.display();
}
```

ABSTRACT CLASS

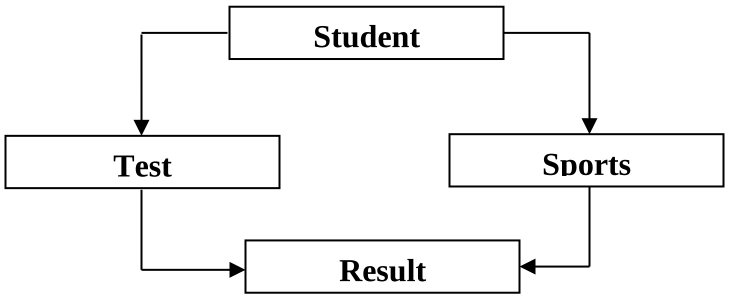
An abstract class is a class that is specifically used as a base class and consists atleast one pure virtual function.

// Example program for Abstract Class

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

class Company                                    // Abstract Class
{
        public:virtual void display( ) = 0;      // Pure Virtual Function
};

class Tata:public Company
{
        public:void display()
                {
                        cout<<endl<<"TATA SHOW ROOM";
                }
};
void main()
{
        Tata obj;
        clrscr();
        obj.display();
}
```

VIRTUAL BASE CLASS

Consider an application that uses both multiple and multilevel inheritance as:



Here, Result is a derived class derived from the classes Test and Sports. Test class is derived from Student and Sports class is derived from Student class. When we apply this format, Result class inherits the properties of Student twice, first via ‘Test’ and again via ‘Sports’. This introduces ambuiguity and should be avoided.

The duplication of inherited members due to those multiple paths can be avoided by making the base class as ‘Virtual base class’. When a class is made as virtual base class, C++ takes necessary steps and allowed to inherit the same functionalities through only one path.

// Example program for Virtual Base Class

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

class Student
{
        int RNO;
        public:void ReadNumber()
                {
                        cout<<"Enter Roll Number =";
                        cin>>RNO;
                }
                void PrintNumber()
                {
                        cout<<endl<<"ROLL NUMBER ="<<RNO;
                }
};

class Test:virtual public Student
{
        protected: int s1,s2;
        public:void ReadMarks()
                {
                        cout<<"Enter Two Subject Marks =";
                        cin>>s1>>s2;
                }
};

class Sports:virtual public Student
{
        protected:int sm;
        public:void SportMarks()
                {
                        cout<<"Enter Sports Marks =";
                        cin>>sm;
                }
};

class Result:public Test,public Sports
```

```
{
        int total;
        public:void ShowResult()
                {
                        total=s1+s2+sm;
                        cout<<endl<<"Total Marks ="<<total;
                }
};

int main()
{
        Result obj;
        clrscr();
        obj.ReadNumber();
        obj.ReadMarks();
        obj.SportMarks();
        obj.PrintNumber();
        obj.ShowResult();
        return 1;
}
```

END