Department of MCA

LECTURE NOTE

ON

OBJECT-ORIENTED PROGRAMMING USING C++

(MCA 2nd Sem)

COURSE CODE: MCA-108

Prepared by :

Mrs. Sasmita Acharya

Assistant Professor

Department of MCA

VSSUT, Burla.

MCA-108 Object Oriented Programming using C++ L-T-P: 3-1-0

Prerequisite: Basic knowledge of object oriented programming.

UNIT I: (10 Hours)
Introduction to object oriented programming, user defined types, polymorphism, and encapsulation. Getting started with C++ - syntax, data-type, variables, strings, functions, exceptions and statements, namespaces and exceptions operators. Flow control, functions, recursion. Arrays and pointers, structures.

UNIT II: (10 Hours)
Abstraction Mechanisms: Classes, private, public, constructors, destructors, member functions, static members, references etc. class hierarchy, derived classes.
Inheritance: simple inheritance, polymorphism, object slicing, base initialization, virtual functions.

UNIT III: (12 Hours)
Prototypes, linkages, operator overloading, ambiguity, friends, member operators, operator function, I/o operators etc. Memory management: new, delete, object copying, copy constructors, assignment operator, this Input/Output.
Exception handling: Exceptions and derived classes function exception declarations, Unexpected exceptions.
Exceptions when handling exceptions, resource capture and release etc.

UNIT – IV: (8 Hours)
Templates and Standard Template library: template classes declaration, template functions, namespaces, string, iterators, hashes, iostreams and other type.
Design using C++ design and development, design and programming, role of classes.

Text Book:
1. E. Balaguruswamy, Object-Oriented Programming in C++. 4 ed, Tata McGraw-Hill.
2. Ashok N. Kamthane, Object-Oriented Programming with ANSI & Turbo C++, Pearson Education.

Reference Books:
1. Herbert Schildt, The Complete Reference C++. 4 ed, Tata McGraw-Hill.
2. Bjarne Stroustrup, Programming: Principles and Practice Using C++. 4 ed, AddisonWesley.

Course Outcomes:
1. Familiar to map real world problems into the Programming language using objects.
2. Can solve the problems in systematic way using class and method paradigms.
3. Efficiently implement linear, nonlinear data structures and various searching and sorting techniques.
4. Efficiently implement Exception handling techniques.

CONTENTS

INTRODUCTION

- C++ is an object oriented programming language & is an extension of c.
- Bjarne Stroustrup at AT & Bell Lab,USA developed it.
- He called the language C with classes.

1.1 Feature

1) Data Abstraction
   Abstraction directs to the procedure of representing essential features without including the background details.
2) Encapsulation
   The packing of data and functions in to a single component is known as encapsulation.
3) Inheritance
   It is the method by which objects of one class get the properties of objects of another class.
4) Polymorphism
   It allows the same function to act differently in different classes.
5) Dynamic Binding
   Binding means connecting one program to another program that is to be executed in reply to the call. It is also known as late binding.
6) Reusability
   Object oriented technology allows reusability of the classes by executing them to other classes using inheritance.
7) Delegation
   When an object of one class is used as a data member in another class such composition of object is known as delegation.
8) Genericity
   This feature allows declaration of variables without specifying exact data type. The compiler identifies the data type at run time.

1.2 Advantages

- OOPS can be comfortably upgraded.
- Using inheritance redundant program, codes can be eliminated & use of previously defined classes may be continued.
- OOP languages have standard class library.

1.3 Uses of OOP

- Object-Oriented DBMS
- Office automation software
- AI and expert systems
- CAD/CAM software
- Network programming
- System software

INPUT & OUTPUT IN C++

2.1 pre-defined streams

cin	• Standard input, usually keyboards, corresponding to stdin in C. • It handles input from input devices.
Cout	• Standard output, usually screen, corresponding to stdout in C. • It passes data to output devices such as monitor and printer.
clog	• It control error messages that are passed from buffer to the standard error device
cerr	• Standard error output, usually screen, corresponding to stderr in C. • it controls the unbuffered output data. • It catches the error & passes to standard error device monitor.

13.1 stream classes

Input stream

- It does read operation through keyboard.
- It uses cin as object.
- The cin statement is used to read data through the input device & uses the extraction operator " >> ".

Syntax: cin>> varname;

```
int a;    float b;
cin>>a>>b;
```

Output stream

- It display contents of variables on the screen.
- It uses insertion operation " << " before variable name.

Syntax :

```
cout << variable;
 int a; float b;
 cout <<a << b;
```

2.3 Typecasting

- It refers to the conversion of data from one basic type to another by applying external use of data type keywords

Program:

Write a program to display from A-Z using typecasting.

```
# include<iostream.h>
#include<conio.h>
void main()
{
int i;
for(i=65;i<=91;i++)
cout<<(char)i;
}
```

get() – it is used to read a single character from the keyboard.

Puts() – it is used to display a single character on the screen.

```
 cin.get(ch);
 cin.put(ch);
```

getline() – it is used to read the string including whitespace. The object cin calls the function by
syntax cin.getline(variable,size);
cin.getline(name,30);
read() – it is used to read text through the keyboard.
Syntax cin.read(variable,size);
write() – it is used to display the string on screen.
Syntax cout.write(variable,size);

C++ DECLARATION

3.1 Parts of C++ program

Include files
Class definition or declaration
Class function definition
main() function(compulsory)

Include files It controls the header files for function declaration.Each header file has an extension.
e.g # include " iostream.h"

Class declaration A class is defined in this section.It contains data variable, data member, function prototype, function definition.

Class function definition This contains definition of function.

main() C++ programs always start execution from main().

3.2 Token

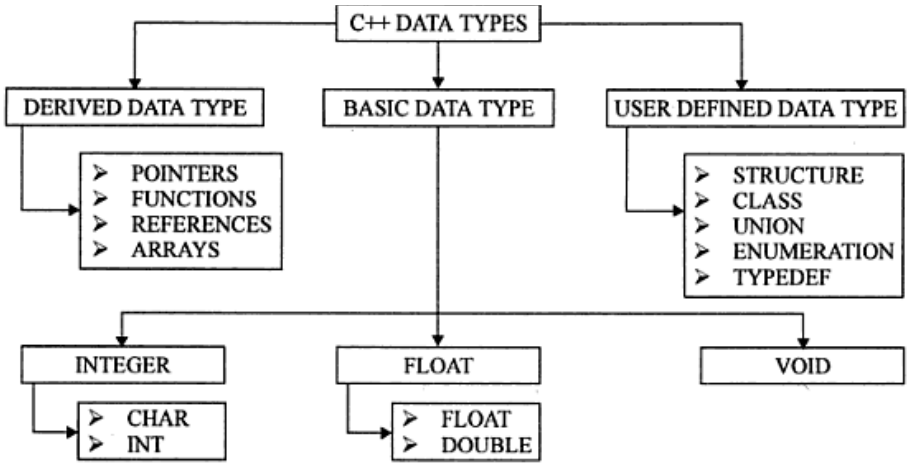
The smallest individual units in a program is known as token.

- Keywords
- Identifiers
- Constants
- Strings
- Operators

3.3 Data types in C++

- Basic data types
- Derived data types
- User defined data type
- Void data type

3.3.1 Basic data types



C++ data types

Basic data types supported by C++ with size and range

DATA TYPE	SIZE IN BYTES	RANGE
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E –308 to 1.7E+308
long double	10	3.4E –4932 to 1.1E +4932
enum	2	-32768 to 32767
bool	1	true / false

3.3.2 Derived data type

a) Pointer

- A pointer is a memory variable that stores a memory address.
- Pinter can have any name i.e legal for other variable and is declared in the same fashion like other variable but it is always denoted by '*' operator.

```
int *a;
float *f;
```

b) Function

- A function is a self-contained block or subprogram of one or more statement that perform a specific task when call

```
main( )
{
fun A( );
fun B( );
}
```

c) Array

- Array is a collection of elements of homogenous data type.

  Program

  Write a program to print array element using pointer.

```
# include<iostream.h>
#include<conio.h>
void main()
{
int a[5],*p;
int i;
clrscr();
cout<<"enter the array element ";
```

```
for(i=0;i<5;i++)
cin>>a[i];
p=a;
cout<<"elements are";
for(i=0;i<5;i++)
cout<<*(p+i);
getch();
}
```

d) Reference

- It is used to define a reference variable.
- This variable prepares an alias for a previously defined variable.

```
int qty=10;
int &qt=qty;
```

syntax : data type & reference variable name=variable name;

3.3.3 User defined data type

a) Structure & classes

```
struct student
{
char nm[30];
char sex[10];
int roll;
}
```

Classes

```
class student
{
 char nm[30];
 char sex[10];
 int roll;
 void input(void);
};
```

b) Union

```
union number
{
char c;
int I;
}
union number num;
```

Anonymous union

- it doesn't contain tag name.
- Elements of such union can be accessed without using tag name.

```
void main()
 {
union
{
```

```
int r;
float f;
};
F=3.1;r=2;
Cout<<r<<f;
}
```

c) Enumerated data type

Keyword: enum

enum logical {true,false}

3.3.4 Void data type

It also known as empty data type..

```
void display()
{
Cout<<"Hello";
}
```

3.4 Operators in C++

1) Insertion operator(<<)
2) Extraction operator(>>)
3) Scope access /resolution operator(::)
4) New(Memory allocation operator)
5) Delete(Memory release operator)
6) Comma operator(,)
7) Reference operator(&)
8) Dereferencing operator(*)

FUNCTIONS IN C++

4.1 Passing Arguments

- The main objective of passing arguments to function is message passing.
- It is also known as communication between two function i.e coller & collee function.

4.1.1 Call by value(pass by value)

- Value of actual argument is passed to the formal argument & operation is done on the formal argument.
- Any change in formal argument doesn't effect the actual argument because formal argument are photocopy of actual arguments .

Program - Write a program to swap two value

```
#include< iostream.h>
void main()
{
int x,y;
void swap(int,int);
cout<<"enter value of x and y";
cin>>x>>y;
swap(x,y);
}
void swap(int a, int b)
{
int k;
k=a; a=b; b=k;
cout<<"a="<<a<<"b="<<b;
}
```

4.1.2 Call by address(pass by address)

- Instead of passing value address are passed. Function operates on addresses rather than values.
- The formal arguments are pointers to the actual argument.hence,changes made in the arguments are permanents.

Program - Write a program to swap two number

```
#include<iostream.h>
void main()
{
int x,y;
void swap(int * ,int *);
cout<<"enter value of x and y";
cin>>x>>y;
swap(&x, &y);
cout<<"x="<<x<<"y="<<y;
}
void swap(int *p, int *q)
```

```
{
int k;
k=*p; *p=*q; *q=k;
}
```

4.1.3 Call by reference(pass by reference)

Program - Write a program to swap two value

```
#include<iostream.h>
void main()
{
int x,y;
void swap(int & , int &);
cout<<"enter value of x and y";
cin>>x>>y;
swap(x, y);
cout<<"x="<<x<<"y="<<y;
}
void swap(int &p, int &q)
{
int k;
k=p;
 p=q;
q=k;
}
```

4.2 Default Arguments

- C++ compiler allows the programmer to assign default values in function prototype.
- When the function is called with less parameter or without parameter the default values are used for operation.

```
#include <iostream.h>
void main()
{
int sum(int a,int b=10;int c=15,int d=20);
int a=2;int b=3; int c=4; int d=5;
cout<<sum(a,b,c,d);
cout<<sum(a,b,c);
cout<<sum(a,b);
cout<<sum(a);
cout<<cum(b,c,d);
}
int sum(int j,int k, int l, int m)
{
 return(j+k+l+m);
}
```

4.3 Inline functions

- C++ provides a mechanism called inline function . When a function is declared as inline the compiler copies the code of the function in calling function i.e function body is inserted in place of function call during compilation.
- Passing of control between coller and collee function is avoided.

Program - Write a program to find square of a number.

```
#include <iostream.h>
inline float square(float j)
{     return(j*j);   }
void main()
{
Int p,q;
cout<<"enter a number:" ;
cin>>p;
q=square(p);
cout<<q;
}
```

4.4 Function overloading

- Defining multiple function with same name is known as function overloading or function polymorphism.
- The overloading function must be different in its argument list & with different data type

Program - Write a program to addition of int & float number using function overloading.

```
#include<iostream.h>
int add(int,int);
float add(float,float,float);
int  main()
{
Clrscr();
float fa,fb,fc,fsum;
int ia,ib,ic,isum;
cout<<"enter integer value:";
cin>>ia>>ib>>ic;
cout<<"enter float value";
cin>>fa>>fb>>fc;
isum=add(ia,ib,ic);
cout<<isum;
fsum=add(fa,fb,fc);
cout<<fsum;
return 0;
}
add(int j, int k, int l)
{
```

```
return(j+k+l);
}
float add(float a, float  b,float c)
{
return(a+b+c);
}
```

4.5 Library function

a) ceil, ceill and floor, floor1

- The function **ceil, ceill** round up given float number.
- The function **floor,floor1** round down float number.
- They are defined in **math.h** header file

```
#include<iostream.h>
#include<math.h>
void main()
{
float num=3.2;
float d,u;
d=float(num);
u=ceil(num);
cout<<num<<u<<d;
}
```

b) Modf and modf1

- The function **modf** breaks double into integer and fraction elements
- The function **modf1** breaks long double into integer and fraction elements.

```
void main()
{
double f,I;
double num=211.57;
f=modf(num,&i);
cout<<num<<i<<f;
}
```

c) abs, fabs and labs

- The function **abs()** returns the absolute value of integer.
- The **fabs()** returns the absolute value of a floating point number.
- The **labs()** returns the absolute value of a long number.

Declaration:

```
Int abs(int n);
double fabs(double n);
long int labs(long int n);
```

d) norm

- The function is defined in complex.h header file and it is used to calculate the square of the absolute value.

Program

Write a program to calculate the square of the complex number using norm() function.

```
#include<iostream.h>
#include<complex.h>
#include<conio.h>
int main(0
{
 double x=-12.5;
cout<<norm(x);
return 0;
}
```

e) complex(), real(), imag() and conj()

complex(): - is defined in complex.h header file & it create complex numbers.

real() :- it returns real part of the complex number .

imag() ;- it returns imaginary part of the complex number.

Conj() :- it returns complex conjugate of a complex number.

Program

Write a program to addition, subtraction and multiplication of two complex number.

```
#include<iostream.h>
#include<conio.h>
class complex
{
public:
        int real,imag;
        void in()
        {
        cout<<"enter real part :";
        cin>>real;
        cout<<"enter imaginary part:";
        cin>>imag;
        }
};
void main()
{
clrscr();
complex a,b,c;
a.in();
b.in();
cout<<"\n addition of two complex number :";
c.real=a.real+b.real;
c.imag=a.imag+b.imag;
cout<<c.real<<"+"<<c.imag<<"i";
cout<<"\n subtraction of two complex number :";
c.real=a.real-b.real;
c.imag=a.imag-b.imag;
```

```
cout<<c.real<<"+"<<c.imag<<"i";
cout<<"\n multiplicaton of two complex number :";
c.real=a.real*b.real-a.imag*b.imag;
c.imag=a.imag*b.real+a.real*b.imag;
cout<<c.real<<"+"<<c.imag<<"i";
getch();
}
```

4.6 Const Argument

- The const variable can be declared using “const” keyword.
- This keyword makes the value of the variable stable.
- It assign an initial value to a variable that can’t be changed later on by the program.

CLASSES & OBJECTS

5.1 Introduction

- A class is a grouping of variables of different data types with function.
- Each variable of a class is called as member variable and function are called as member function.

Example

```
class item
{
private:
    int codeno;
    float price;
    int qty;
    void values;
public:
    void show();
};
void values()
{
cout<<"enter codeno,price,qty";
cin>>codeno>>price>>qty;
}
void show()
{
values();
cout<<codeno<<price<<qty;
}
void main()
{
item a;
a.show();
}
```

- An object can't directly access the member variables & functions declared in private section but it can access those declared in public section.
- The private member of a class can be accessed by public member function of same class

5.2 Declaring objects

Public keyword – it is used to allow an object to access the member variable of a class directly like structure.

Private keyword – it is used to prevent direct access to member variable or member function by the object. By default the class member are private.

Protected keyword – it is same as private. It is frequently used in inheritance

5.3 Defining member function

Program – Write a program to find simple interest using class concept & data should be declared in private section

```
#include<iostream.h>
#include<conio.h>
class interest
{
private:
    float p,i,r;
    float s.i;
    voi in();
public:
    void calc();
    void display();
    void in()
    {
    cout<<"enter p,i,r";
    cin>>p>>i>>r;
    }
    void calc()
    {
    in();
    s.i=p*i*r/100;
    amt= p+s.i;
    }
    void display()
    {
    cout<<"simple interest is";
    }
};
void main()
{
interest i;
i.calc();
i.display();
}
```

5.4 Declaring member function outside the class

```
#include<iostream.h>
#include<conio.h>
class item
{
private:
    int codeno;
    float price;
    int qty;
public:
```

```
        void show(void);
    };

    void item::show()
    {
    cout<<"enter codeno,price,qty";
    cin>>codeno>>price>>qty;
    }
    void main()
    {
    item one;
    one.show();
    }
```

5.5 Static member variable

- Once a data member variable is declared as static only one copy of that member is created for the whole class.

```
#include<iostream.h>
#include<conio.h>
class number
{
static int c;
int k;
public:
        void zero()
        {
        k=0;
        }
        void count()
        {
        ++c; ++k;
        cout<<c<<k;
        }
};
int number::c=0;
void main()
{
number A,B,C;
A.zero;
B.zero;
C.zero;
A.count;
B.count;
C.count;
}
```

5.6 Static member function

- When a function is defined as static it can access only static member variables & functions of same class.
- The non-static member are not available to these function.
- It can also declared in private section but it must invoked using a static public function.

```
#include<iostream.h>
#include<conio.h>
class bita
{
private:
        static int c;
public:
        static void count()
        {
        c++;
        }
        static void display()
        {
        cout<<c;
        }
};
int bita::c=0;
void main()
{
bita::display();
bita::count();
bita::count();
bita::display();
}
```

5.7 Static object

- When an object is declared as static ,all it's data member is initialized to 0;
- The declaration of static object removes garbage of it's data members & initialize them to 0.

```
#include<iostream.h>
#include<conio.h>
class bita
{
private:
    static int c,k;
public:
    void plus()
    {
    c+=2;
```

```
        k+=2;
        }
        void show()
        {
        cout<<c<<k;
        }
};
void main()
{
static bita A;
A.plus();
A.show();
}
```

5.8 Object as function arguments

5.8.1 pass-by value

- A copy of an actual object is send to function & assign to a formal argument.
- Both actual and formal copies of objects are stored at different memory location.
- So, changes made to formal objects are not reflected in actual argument.

```
#include<iostream.h>
#include<conio.h>
class life
{
int mfgyr;
float expyr;
int yr;
public:
        void getyr()
        {
        cout<<"enter manufacterer and exp date";
        cin>>mfgyr>>expyr;
        }
        void period(life);
};
void life::period(life y1)
{
yr=y1.expyr-y1.mfgyr;
cout<<"product life="<<yr<<"years";
}
void main()
{
clrscr();
life a1;
a1.getyrs();
```

```
a1.period(a1);
}
```

5.8.2 pass-by reference

- Here, address of actual object is implicitly send to the called function using a reference object.
- As actual object & reference object share the same memory space. So, any change made to reference object will also reflected in the actual object.

```
#include<iostream.h>
#include<conio.h>
class life
{
int mfgyr;
float expyr;
int yr;
public:
    void getyr()
    {
    cout<<"enter manufacterer and exp date";
    cin>>mfgyr>>expyr;
    }
    void period(life &);
};
void life::period(life &y1)
{
yr=y1.expyr-y1.mfgyr;
cout<<"product life="<<yr<<"years";
}
void main()
{
clrscr();
life a1;
a1.getyrs();
a1.period(& a1);
}
```

5.8.3 pass by address

- Here, address of actual object is explicitly send to the called function using a pointer object.
- So, any change made to pointer object will also reflected in the actual object as the pointer object holds the address of actual object.

```
#include<iostream.h>
#include<conio.h>
class life
{
```

```
int mfgyr;
float expyr;
int yr;
public:
    void getyr()
    {
    cout<<"enter manufacterer and exp date";
    cin>>mfgyr>>expyr;
    }
    void period(life *);
};
void life::period(life *y1)
{
yr=y1->expyr-y1->mfgyr;
cout<<"product life="<<yr<<"years";
}
void main()
{
clrscr();
life a1;
a1.getyrs();
a1.period(& a1);
}
```

5.9 Friend Function

- C++ allows a mechanism in which a non-member function has access permission to the private data member of the class
- This can be done by declaring a non-member function as friend to the class whose private data is to be accessed.

Program- Write a program to addition of 3 number using friend function.

```
#include<iostream.h>
#include<conio.h>
class B;
class C;
class A
{
int a[5];
public:
        void in();
        friend C sum(A,B,C);
};
void A::in()
{
int k;
cout<<"\n enter five integer:";
for(k=0;k<5;k++)
```

```
cin>>a[k];
}
class B
{
int b[5];
public:
        void in();
        friend C sum(A,B,C);
};
void B::in()
{
int k;
cout<<"\n enter five integer:";
for(k=0;k<5;k++)
cin>>b[k];
}
class C
{
int c[5];
public:
        void out();
        friend C sum(A,B,C);
};
void C::out()
{
cout<<"\n addition:";
for(int k=0;k<5;k++)
cout<<" "<<c[k];
}
C sum(A a1,B b1,C c1)
{
for(int k=0;k<5;k++)
c1.c[k]=a1.a[k]+b1.b[k];
return c1;
}
void main()
{
clrscr();
A a;
B b;
C c;
a.in();
b.in();
c=sum(a,b,c);
c.out();
```

```
getch();
}
```

5.10 Friend classes

- When all the functions need to access another class in such a situation , we can declare an entire class as friend class
- Friend is not inheritable from one class to another.
- Declaring class A to be a friend of class B doesn't mean that class B is also friend of class A. therefore, friendship is not exchangeable.

```
#include<iostream.h>
#include<conio.h>
class B;
class A
{
private:
        int a;
public:
        void aset()
        {a=30;}
        void show(B);
};
class B
{
private:
        int b;
public:
        void bset()
        { b=40}
        friend void A::show(B b);
};
void Ashow(B b)
{
cout<<a<<b.b;
}
void main()
{
clrscr();
A a1;
a1.aset();
B b1;
b1.bset();
a1.show(b1);
getch();
}
```

5.11 Constant member function

- Member function of a class can also be declared as constant using ‘const keyword’.
- The const function can’t modify any data in the class.
- It is added as suffix.

5.12 Recursive member function

- When function call itself repeatedly, called recursive function .
  1) Direct recursion
  2) Indirect recursion

5.13 Member function & non-member function

program – Write a program to call a member function using non-member function.

```
#include<iostream.h>
#include<conio.h>
void moon();
class mem
{
public:
    void earth()
    {
    cout<<"on earth:";
    }
};
void main()
{
clrscr();
moon();
}
void moon()
{
mem j;
j.earth();
cout<<"on moon";
}
```

5.14 Overloading member function

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class absv
{
public:
        int num(int);
        double num(double);
};
int absv::num(int x)
{
```

```
int ans=abs(x);
return(ans);
}
double absv::num(double d)
{
double ans=fabs(d);
return(ans);
}
void main()
{
clrscr();
absv n;
int i;
double j;
cout<<"enter value of i & j";
cin>>i>>j;
cout<<n.num(i)<<n.num(j);
}
```

5.15 Overloading main() function

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
        void main(int t)
        {
        cout<<t;
        }
        void main(double f)
        {
        cout<<f;
        }
        void main(char *s)
        {
        cout<<s;
        }
};
void main()
{
clrscr();
A *a;
int i;
double j;
char k[10];
```

```
cout<<"enter value of i,j,k";
cin>>i>>j>>k;
a->main(i);
a->main(j);
a->amin(k);
}
```

5.16 Indirect recursion

- When a function call itself repeatedly, known as direct recursion.
- When two function call each other repeatedly, known as indirect recursion.

program-write a program to find the factorial of number using indirect recursion.

```
#include<iostream.h>
#include<conio.h>
int n=5,f=1,j;
main(int x)
{
void pass();
if(x==0)
{
cout<<"factorial ="<<f;
}
f=f*x;
pass();
return x;
}
void pass()
{
main(m--);
}
```

CONSTRUCTOR & DESTRUCTOR

1. C++ provides a pair of inbuilt special member function called constructor & destructor.
2. The constructor constructs the object allocates memory for data members & also initializes them.
3. The destructor destroys the object when it is of no use or goes out of scope & deallocates the memory.
4. The compiler automatically executes these functions.
5. When an object is created ,compiler invokes the constructor function.
6. Destructor is executed at the end of the function when objects are of no use & goes out of scope.

Example

```
Class num
{
Private:
    int a,b,c;
public:
    num();
~num();
};
num:;num()
{
a=0;b=0;c=0;
}
num::~num()
{
cout<<"destructor invoked";
}
void main()
{
num x;
}
```

6.1 Constructor with Arguments

- The constructors are also called parameterized constructor.
- It is necessary to pass values to the constructor when an object is created

Program

```
#include<iostream.h>
#include<conio.h>
class num
{
Private:
        int a,b,c;
public:
        num(int m,int n,int k)
        void show()
        {
        cout<<a<<b<<c;
```

```
        }
};
num::num(int m,int j,int k)
{
a=m; b=j; c=k;
}
void main()
{
clrscr();
num x=num(4,5,7);
num y(1,2,8);
x.show();
y.show();
}
```

6.2 Constructor overloading

- When a class contains more then one constructor all defined with the same name as the class but with different number of arguments, is called constructor overloading
- Depending upon number of arguments the compiler executes the appropriate constructor.

Program – Write a program to find the simple interest using constructor overloading.

```
#include<iostream.h>
#include<conio.h>
class si
{
private:
        float p,i,r;
public:
        float SI  ;
        si(float a,float b,float c);
        si(float x,float y);
        si(float z);
        si();
        void out();
};
si::si(float a,float b,float c)
{
p=a;
i=b;
r=c;
}
si::si(float x,float y)
{
p=x;
i=y;
r=5;
}
```

```
si::si(float z)
{
p=z;
i=3;
r=2;
}
si::si()
{
p=20;
i=4;
r=2;
}
void si::out()
{
SI=(p*i*r)/100;
cout<<"simple interest:"<<SI<<endl;
}
void main()
{
clrscr();
si a(5.3,2.1,6.3) ;
a.out();
si b(4.5,3.2) ;
b.out();
si c(5.3) ;
c.out();
si d ;
d.out();
getch();
}
```

6.3Constructor with default Argument

Program – Write a program to find the power of a number using default arguments.

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class power
{
        Private:
                int num;
                int pow,ans;
        public:
                power(int n=9,int p=3)
                void show()
                {
                Cout<<ans;
```

```
        }
};
power::power(int n,int p)
{
num=n;
pow=p;
ans=pow(n,p);
}
void main()
{
clrscr();
power(p1,p2(5));
p1.show();
p2.show();
getch();
}
```

6.5 Copy Constructor

- When we pass the reference of an object to a constructor function, declaration is known as copy constructor.
- All copy constructor required one argument with reference to an object of that class.
- Using copy constructor ,it is possible for the programmer to declare and initialize ne object using reference of another objects

Program – Write a program to show copy constructor

```
#include<iostream.h>
#include<conio.h>
class num
{
        int n;
        public:
                num(){ }
                num(int k)
                {
                n=k;
                }
                num(num &j)  //copy constructor
                {
                n=j.n;
                }
                void show()
                {
                cout<<n;
                }
};
void main()
{
```

```
clrscr();
num J(50);
num K(J);
num L=J; //copy constructor invoked
num M;
M=J;
J.show();
K.show();
L.show();
M.show();
}
```

6.6Const Object

- An object can be declared constant using const keyword
- A constructor function can initialize the data member of a constant object.
- A const object can access only constant function.

Program – Write a program to show

```
#include<iostream.h>
#include<conio.h>
class ABC
{
int a;
public:
        ABC(int m)
        {
        a=m;
        }
        void show() const
        {
        cout<<a;
        }
};
int main()
{
clrscr();
const ABC x(5);
x.show();
return 0;
}
```

6.7 Destructor

- For real 7 non-static object the destructor is executed ,when object goes out of scope.
- It is not possible to define more than one destructor .
- The destructor is only way to destroy the object. so, they can't be overloaded.
- Destructor neither required any argument nor return any value.

Program – Write a program to destroy an object

```
#include<iostream.h>
#include<conio.h>
int c=0;
class A
{
public:
    A();
    ~A();
};
A::A()
{
c++;
cout<<"\n object created:"<<c;
}
A::~A()
{
c--;
cout<<"\n object destroyed:"<<c;
}
void main()
{
clrscr();
A a1,a2,a3;
getch();
}
```

6.8 Qualifier & Nested classes

- The class declaration can be done in another class. While declaring object of such class ,it is necessary to precede the name of outer class.
- The name of outer class is called qualifier name & class defined inside class is called as nested class .

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
        int x;
        A()
        {
        x=5;
        cout<<x;
        }
        class B
        {
        public:
                int y;
                B()
```

```
        {
        y=10;
        cout<<y;
        }
        class C
        {
        public:
                int z;
                C()
                {
                z=15;
                cout<<z;
                }
        };
    };
};
void main()
{
clrscr();
A a;
A:: B b;
A::B::C c;
}
```

6.9 Anonymous object

- Objects are created with names. it is possible to create object without name & such objects are known as anonymous objects.
- We can gate the address of anonymous object using ‘this pointer’.

```
#include<iostream.h>
#include<conio.h>
class noname
{
private:
        int x;
public:
        noname(int j)
        {
        x=j;
        cout<<x;
        }
        noname()
        {
        x=15;
        cout<<x;
        cout<<this;
        }
```

```
        ~noname()
        {
        cout<<"in destructor";
        }
};
void main()
{
clrscr();
noname();
noname(12);
}
```

6.10 Private Constructor and Destructor

- When a function is declared as private section ,it can be invoked by public member function of the same class, but when constructor & destructor are declared as private they can't executed implicitly & it is a must to execute them explicitly.

```
#include<iostream.h>
#include<conio.h>
class A
{
private:
        int x;
        A()
        {
        x=7;
        cout<<"in constructor";
        }
        ~A()
        {
        cout<<"in destructor";
        }
public:
        void show()
        {
        this ->A::A();
        cout<<x;
        this ->A::~A();
        }
};
void main()
{
clrscr();
A *a;
a->show();
}
```

6.11 main() as constructor & destructor

```
#include<iostream.h>
#include<conio.h>
class main()
{
public:
        main()
        {
        cout<<"in constructor main";
        }
        ~main()
        {
        cout<<"in destructor main";
        }
};
void main()
{
clrscr();
class main a;
}
```

6.12 Local versus Global Object

- The object declared outside function bodies is known as global object.
- All function can access the global object.
- The object declared inside function bodies is known as local object.
- Scope is local to its current block.

INHERITANCE

- The procedure of creating a new class from one or more existing classes is called inheritance.
- The program can defined new member variables and function in the derived class .
- The base class remain unchanged.
- An object of derived class can access members of base as well as derived class but reverse is not possible.
- The term reusability means reuse of properties of base class in the derived class.

7.1Types of Inheritance

7.1.1 Single Inheritance: It is the inheritance hierarchy wherein one derived class inherits from one base class.

Program- Write a program to show single inheritance.

```
#include<iostream.h>
#include<conio.h>
class student
{
    public:
    int rno;
    //float per;
    char name[20];
    void getdata()
    {
            cout<<"Enter RollNo :- \t";
            cin>>rno;
            cout<<"Enter Name :- \t";
            cin>>name;

    }
};
class marks : public student
{
public:
    int m1,m2,m3,tot;
    float per;
    void getmarks()
    {
            getdata();
            cout<<"Enter Marks 1 :- \t";
            cin>>m1;
            cout<<"Enter Marks 2 :- \t";
            cin>>m2;
            cout<<"Enter Marks 2 :- \t";
            cin>>m3;
    }
    void display()
```

```
    {
            getmarks();
            cout<<"Roll Not \t Name \t Marks1 \t marks2 \t Marks3 \t Total \t Percentage";
            cout<<rno<<"\t"<<name<<"\t"<<m1<<"\t"<<m2<<"\t"<<m3<<"\t"<<tot<<"\t"<<per;
    }
};
void main()
{
    student std;
    clrscr();
    std.getmarks();
    std.display();
    getch();
}
```

7.1.2 Multiple Inheritance: It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es)

Program - Write a program calculates the area and perimeter of an rectangle but, to perform this program, multiple inheritance is used.

```
#include <iostream>
using namespace std;
class Area
{
 public:
   float area_calc(float l,float b)
   {
      return l*b;
   }
};

class Perimeter
{
 public:
   float peri_calc(float l,float b)
   {
      return 2*(l+b);
   }
};

/* Rectangle class is derived from classes Area and Perimeter. */
class Rectangle : private Area, private Perimeter
{
   private:
      float length, breadth;
   public:
     Rectangle() : length(0.0), breadth(0.0) { }
```

```
    void get_data( )
    {
       cout<<"Enter length: ";
       cin>>length;
       cout<<"Enter breadth: ";
       cin>>breadth;
    }

    float area_calc()
    {
    /* Calls area_calc() of class Area and returns it. */

       return Area::area_calc(length,breadth);
    }

    float peri_calc()
    {
    /* Calls peri_calc() function of class Perimeter and returns it. */

       return Perimeter::peri_calc(length,breadth);
    }
};

int main()
{
   Rectangle r;
   r.get_data();
   cout<<"Area = "<<r.area_calc();
   cout<<"\nPerimeter = "<<r.peri_calc();
   return 0;
}
```

7.1.3 Hierarchical Inheritance: It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

Program- Write a program to implement hierarchical inheritance.

```
#include<iostream.h>
#include<conio.h>
class A
{
   public:
   int a,b;
   void getnumber()
   {
   cout<<"\n\nEnter Number :::\t";
   cin>>a;
   }
```

```
};

class B : public A
{
   public:
   void square()
   {
   getnumber();
   cout<<"\n\n\tSquare of the number :::\t"<<(a*a);

   }
};

class C :public A
{
   public:
   void cube()
   {
   getnumber(); //Call Base class property
   cout<<"\n\n\tCube of the number :::\t"<<(a*a*a);

   }
};

int main()
{
clrscr();

B b1;
b1.square();
C c1;
c1.cube();

getch();
}
```

7.1.4 Multilevel Inheritance: It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

Program – Write program to implement multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student    // base class
{
private:
```

```
  int rl;
  char nm[20];
public:
   void read();
   void display();
};
class marks : public student // dervivec from student
{
protected:
  int s1;
  int s2;
  int s3;
public:
   void getmarks();
   void putmarks();
};
class result : public marks // derived  from marks
{
private:
  int t;
  float p;
  char div[10];
public:
   void process();
   void printresult();
};
   void student::read()
   {
    cout<<"enter Roll no and Name "<<endl;
    cin>>rl>>nm;
   }
   void student:: display()
   {
   cout <<"Roll NO:"<<rl<<endl;
   cout<<"name : "<<nm<<endl;
   }
   void marks ::getmarks()
   {
   cout<<"enter three subject marks  "<<endl;
    cin>>s1>>s2>>s3;
   }
   void marks ::putmarks()
   {
    cout <<"subject 1:"<<s1<<endl;
    cout<<" subject 2 : "<<s2<<endl;
```

```
 cout <<"subject 3:"<<s3<<endl;
}

void result::process()
{
 t= s1+s2+s3;
 p = t/3.0;
 p>=60?strcpy(div,"first"):p>=50?strcpy(div, "second"): strcpy(div,"third");
}

void result::printresult()
{
 cout<<"total = "<<t<<endl;
 cout<<"per  = "<<p<<endl;
 cout<<"div = "<<div<<endl;
}
void main()
{
 result x;
 clrscr();
 x.read();
 x.getmarks();
 x.process();
 x.display();
 x.putmarks();
 x.printresult();
 getch();
}
```

7.1.5 Hybrid Inheritance: The inheritance hierarchy that reflects any legal combination of other four types of inheritance.

Program- Write a program to implement hybrid inheritance

```
#include<iostream.h>
#include<conio.h>
class arithmetic
{
protected:
int num1, num2;
public:
void getdata()
{
cout<<"For Addition:";
cout<<"\nEnter the first number: ";
cin>>num1;
cout<<"\nEnter the second number: ";
```

```
cin>>num2;
}
};
class plus:public arithmetic
{
protected:
int sum;
public:
void add()
{
sum=num1+num2;
}
};
class minus
{
protected:
int n1,n2,diff;
public:
void sub()
{
cout<<"\nFor Subtraction:";
cout<<"\nEnter the first number: ";
cin>>n1;
cout<<"\nEnter the second number: ";
cin>>n2;
diff=n1-n2;
}
};
class result:public plus, public minus
{
public:
void display()
{
cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
}
};
void main()
{
clrscr();
result z;
z.getdata();
z.add();
z.sub();
z.display();
```

```
getch();
}
```

7.1.6 Multipath Inheritance :

When a class is derived from two or more classes that are derived from same base class . such type of inheritance is known as multipath inheritance.

Program- Write a program to implement multipath inheritance

```
#include<iostream.h>
#include<conio.h>
class student
{
protected:
    int roll;
    char nm[30],branch[20],sem[10];
};
class exam:virtual public student
{
protected:
    int m1,m2,m3,m4,m5,m6;
};
class sport:virtual public student
{
protected:
    char gd[10],sp[10];
};
class result:public exam,sport
{
protected:
    int total;
public:
    void in()
    {
    cout<<"enter name,roll,branch,semester & 6 subject mark:";
    cin>>nm>>roll>>branch>>sem>>m1>>m2>>m3>>m4>>m5>>m6;
    total=(m1+m2+m3+m4+m5+m6);
    cout<<"enter sports name & grade:";
    cin>>gd>>sp;
    }
    void out()
    {
    cout<<"\n name="<<nm<<"\n roll="<<roll<<"\n branch="<<branch<<"\n semester="<<sem;
    cout<<"\n total="<<total;
    cout<<"\n sports="<<sp<<"\n grade="<<gd;
    }
};
void main()
{
```

```
clrscr();
result r;
r.in();
r.out();
getch();
}
```

7.2 Container class

- Declaring object of one class as data member in another class is known as deligation.
- A class that has objects of another class as its data member is known as container class
- This kind of relationship is known as has-a-relationship or containership.

Program- Write a program to implement container class

```
#include<iostream.h>
#include<conio.h>
class door
{
public:
        int dc,dn;
        void in()
        {
        cout<<"enter no.of doors:";
        cin>>dn;
        dc=2000*dn;
        cout<<dc;
        }
};
class windo
{
public:
        int wc,wn;
        void in()
        {
        cout<<"enter no.of windows:";
        cin>>wn;
        wc=1500*wn;
        cout<<wc;
        }
};
class house
{
public:
        door d;
        windo w;
        long int hc,rn,rc;
        house()
```

```
        {
        d.in();
        w.in();
        cout<<"enter the no. of rooms and price of room:";
        cin>>rn>>rc;
        hc=d.dc+w.wc+rn*rc;
        cout<<"price="<<hc;
        }
        ~house()
        {
        }
};
void main()
{
clrscr();
house h;
getch();
}
```

7.3 Abstract class

- When a class is not used for creating objects, it is called as abstract class.
- Abstract class can act only as base class.

7.4 Common constructor

- When constructor of derived class is used to initialize the data members of base class as well as derived class, known as common constructor.

7.5 Pointer & Inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{
Protected:
int x,y;
public:
    A()
    {
    x=1; y=2;
    }
};
class B:private A
{
Public:
    Int z;
    B()
    {
    Z=3;
```

```
    }
};
void main()
{
clrscr();
B b;
int *p;
obj.in();
p=&b.z;
cout<<(unsigned)p<<*p;
p--;
cout<<(unsigned)p<<*p;
p--;
cout<<(unsigned)p<<*p;
}
```

7.6 Overloading Member function

```
#include<iostream.h>
#include<conio.h>
class B
{
public:
    void show()
    {
    cout<<"in base class";
    }
};
class D:public B
{
public:
    void show()
    {
    cout<<"in derived class";
    }
};
void main()
{
clrscr();
B b;
D d;
b.show();
d.show();
d.B::show();
getch();
}
```

OPERATOR OVERLOADING

8.1 Introduction

- The capability to relate the existing operator with a member function and use the resulting operator with objects of its class as its operands is called operator overloading.

  Program – Write a program to perform addition of two objects

```
#include<iostream.h>
#include<conio.h>
class number
{
public:
int x,y;
number()
{        }
number(int j,int k)
{
x=j;
y=k;
}
numberoperator +(number D)
{
number T;
T.x=x+D.x;
T.y=y+D.y;
return T;
}
void show()
{
cout<<"x="<<x<<"y="<<y;
}
};
void main()
{
clrscr();
number A(2,3),B(y,5),c;
A.show();
B.show();
C=A+B;
C.show();
getch();
}
```

8.2 Overloading Unary operator

/* Write a program to overload unary operator */

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

class complex
{
    int a,b,c;
   public:
      complex(){}
      void getvalue()
     {
            cout<<"Enter the Two Numbers:";
            cin>>a>>b;
     }



  void operator++()
     {
            a=++a;
            b=++b;
     }

     void operator--()
     {
            a=--a;
            b=--b;
      }

      void display()
      {
            cout<<a<<"+\t"<<b<<"i"<<endl;
       }
};

void main()
{
    clrscr();
    complex obj;
    obj.getvalue();
    obj++;
    cout<<"Increment Complex Number\n";
    obj.display();
    obj--;
    cout<<"Decrement Complex Number\n";
    obj.display();
    getch();
}
```

8.3 Overloading binary operator using friend function

```
#include<iostream.h>
#include<conio.h>
class num
{
private:
    int a,b,c,d;
public:
    void in()
    {
    cout<<"enter a value of a,b,c,d:";
    cin>>a>>b>>c>>d;
    }
    void show();
    friend num operator *(int,num);
};
void num::show()
{
cout<<a<<b<<c<<d;
}
num operator *(int a,num t)
{
num tmp;
tmp.a=a*t.a;
tmp.b=a*t.b;
tmp.c=a*t.c;
tmp.d=a*t.d;
return(tmp);
}
void main()
{
clrscr();
num x,z;
x.in();
x.show();
z=3*x;
z.show();
}
```

8.4 Overloading increment & decrement operator

```
#include<iostream.h>
#include<conio.h>
class number
{
float x;
public:
```

```
    number(float k)
    {
    x=k;
    }
    void operator ++(int)
    {
    x++;
    }void operator--()
    {
    --x;
    }
    void show()
    {
    cout<<x;
    }
};
void main()
{
clrscr();
number N(2,3);
N.show();
N++;
N.show();
--N;
N.show();
getch();
}
```

8.5 Overloading special operator

```
#include<iostream.h>
#include<conio.h>
class num
{
int a[3];
public:
    num (int i,int j, int k)
    {
    a[0]=i;
    a[1]=j;
    a[2]=k;
    }
    int operator [ ](int i)
    {
    return a[i];
    }
```

```
};
void main()
{
clrscr();
num ob(1,2,3);
cout<<ob[1];
cout<<ob[2];
cout<<ob[3];
getch();
}
```

8.6 Overloading function call operator

```
#include<iostream.h>
#include<conio.h>
class loc
{
int longitude,latitude;
public:
    loc() { }
    loc(int lg,int lt)
    {
    longitude=lg;
    latitude= lt;
    }
    void show()
    {
    cout<<longitude<<latitude;
    }
    loc operator() (int i,int j);
};
loc loc::operator()(int i,int j)
{
longitude= i;
latitude=j;
return *this;
}
void main()
{
clrscr();
loc ob1(10,20);
ob1.show();
ob1(7,8);
ob1.show();
}
```

8.7 Overloading class member access operator

```
#include<iostream.h>
#include<conio.h>
class num
{
public:
    int i;
    num *operator->()
    {
    return this;
    }
};
void main()
{
num ob;
ob->i=10;
cout<<ob.i<<  <<ob->i;
}
```

8.8 Overloading comma operator

```
/* Write a program to overload comma operator*/
#include<iostream.h>
#include<conio.h>
class loc
{
int longitude,latitude;
public:
    loc() { };
    loc(int lg,int lt)
    {
    longitude=lg;
    latitude=lt;
    }
    void show()
    {
    cout<<"\n longitude="<<longitude<<"latitude="<<latitude;
    }
    loc operator +(loc op2);
    loc operator,(loc op2);
};
loc loc :: operator,(loc op2)
{
loc temp;
temp.longitude=op2.longitude;
temp.latitude=op2.latitude;
return temp;
```

```
}
loc loc::operator +(loc op2)
{
loc temp;
temp.longitude=op2.longitude+longitude;
temp.latitude=op2.latitude+latitude;
return temp;
}
void main()
{
clrscr();
loc ob1(10,20),ob2(5,30),ob3(1,1);
ob1.show();
ob2.show();
ob3.show();
ob1=(ob1,ob2+ob2,ob3);
ob1.show();
getch();
}
```

8.9 Overloading stream operator

```
#include<iostream.h>
#include<conio.h>
class mat
{
public:
    int r,c,a[30][30],i,j;
    void in()
    {
    cout<<"\nEnter row & column size";
    cin>>r>>c;
    cout<<"\nEnter"<<r*c<<" elements of matrix:";
    for(i=0;i<r;i++)
    {
    for(j=0;j<c;j++)
    cin>>a[i][j];
    }
    }

    friend ostream & operator << (ostream &,mat &);
    friend istream & operator >> (istream &,mat &);

};
istream & operator>>(istream &in, mat &m)
{
cout<<"\nEnter row & column size:";
```

```
cin>>m.r>>m.c;
cout<<"\nEnter "<<m.r*m.c<<" elements :";
   for (int i = 0; i < m.r; ++i)
   {
     for (int j = 0; j < m.c; ++j)
        in >> m.a[i][j];
   }
   return in;
}
ostream & operator<<(ostream &out, mat &m)
{
   for (int i = 0; i < m.r; ++i)
   {
     for (int j = 0; j < m.c; ++j)
      out << m.a[i][j] << "  ";
     out << endl;
   }
   return out;
}
void main()
{
clrscr();
mat p,q;
p.in();
q.in();
mat M1;
cin>>M1;
cout<< M1;
getch();
}
```

POINTERS AND ARRAYS

9.1 Void Pointer:

- Pointers can also be declared as void type. When declared void 2 bytes are allocated to it. Later using type casting actual number of bytes can be allocated or deallocated.
  Ex:

```
#include<iostream.h>
#include<conio.h>
int p; float d; char c;
void *pt=&p;
void main()
{
        clrscr();
        *(int *)pt=12;
        cout<<p;
        pt=&d;
        *(float *)pt=5.4;
        cout<<d;
        pt=&c;
        *(char *)pt='s';
        cout<<c;

}
```

9.2 Wild pointer:

- When a pointer points to an unallocated memory location or to data value whose memory is deallocated such a pointer is called as wild pointer.
- This pointer generates garbage memory location and pendent reference.
- A pointer can become a wild pointer due to :
  - ✓ Pointer is declared but not initialized.
  - ✓ Pointer alteration:
    It is the careless assignment of new memory location in a pointer. This happenes when other wild pointer access the location of a legal pointer. This wild pointer than converts a legal pointer to wild pointer.
  - ✓ Access destroyed data:
    This happens when the pointer attempts to access data that has no longer life.

9.3 Class Pointer:

- Class pointer are pointers that contains the starting address of member variables.

Ex:

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

```
#include<conio.h>
void main()
{
    class man
    {
        public:
            char nm[20];
            int age;
    };
    man m= {“Ravi”,15};
    man *ptr;
    ptr= &(man)m;
    clrscr();
    cout<<m.name<<m.age;
    cout<<ptr->name<<ptr->age;
}
```

9.4 Pointers to derived and base classes:

```
#include<iostream.h>
#include<conio.h>
class A
{
    public:
        int b;
        void display()
        {
            cout<<b;
        }
};
class B : public A
{
    public:
        int d;
        void display()
        {
            cout<<b<<d;
        }
};
void main()
{
    clrscr();
    B *cp;
    B b;
    cp=&b;
    cout<<”Enter values for b & d :”;
```

```
        cin>>cp->b>>cp->d;
        cp->display();
}
```

9.5 Array of classes:

- In this array every element is of class type.

Ex:

```
#include<iostream.h>
#include<conio.h>
class student
{
    public:
            char name[30];
            int rolln0;
            char branch[10];
};
class student st[10];
```

9.6 BINDING AND POLYMORPHISM

Binding means link between a function call and the real function that is executed when a function is called.

- There are two types of binding :-
  i) Compile-time or early or static binding
  ii) Run-time or late or dynamic binding
- In early binding, the information pertaining to various overloaded member function is to be given to the compiler while compiling.
- Deciding a function call at compile time is called static binding.
- Deciding a function call at run time is called dynamic binding.
- It permits suspension of the decision of choosing a suitable member function until run time.
- There are two types of polymorphism:
  i) Run-time polymorphism (Virtual Function)
  ii) Compile-time polymorphism (Function Overloading & Operator Overloading)

9.7 Compile time (Virtual Function):

```
class first
{
        int d;
        first() {d=5;}
        public:
                void display()
                {
                        cout<<d;
                }
};
```

```
class second : public first
{
        int k;
        public:
                second() {k=10;}
                void display()
                {
                        cout<<k;
                }
};
void main()
{
        second r;
        r.display();
}
```

- The virtual function of base classes must be redefined in the derive classes.
- The programmer can define a virtual function in a base class and can use the same function name in any derived class even if the number and type of arguments are matching.
- The matching function overwrites the base class function of the same name. This is called as function overriding.
- The base Class Version is available to derived class objects via scope overriding.

Rules For Virtual Functions:

- The virtual functions should not be static and must be member of a class.
- A virtual function may be declared as friend of another class.
- Constructors can't be declared as virtual but destructors can be virtual.
- The virtual function must be defined in the public section of the class.
- The prototype of virtual functions in base and derived classes should be exactly the same.
- In case of mismatch the compiler neglects the virtual function mechanism and treate them as overloaded function.
- If a base class contain virtual function and if the same function is not redefined in the derived classes in that case the base class function is invoked.
- The keyword virtual prevents the compiler to perform early binding. Binding is postponed until run time.
- The operator keyword used for operator overloading also supports virtual mechanism.

```
#include<iostream.h>
#include<conio.h>
class first
{
        int b;
        public:
                first()
```

```
        {
            b=10;
        }
        virtual void display()
        {
            cout<<"b="<<b;
        }
};
class second : public first
{
    int d;
    public:
        second()
        {
            d=20;
        }
        void display()
        {
            cout<<"d="<<d;
        }
};
void main()
{
    clrscr();
    first f, *p;
    second s;
    p=&f;
    p->display();
    p=&s;
    p->display();
}
```

9.8 Pure Virtual Function :

- In practical applications the member functions of base classes are rarely used for doing any operation such functions are called as do-nothing functions or dummy functions or pure virtual functions.
- They are defined with null body. So that the derived classes should be able to over write them.
- After declaration of a pure virtual function in a class the class becomes an abstract class. It can't be used to declare any object.

syntax:

```
virtual void functionname () =0;
```

Ex:

```
virtual void display () =0;
```

- Any attempt to declare an object will result in an error that is can't create an instance of abstract class.

- Here the assignment operator is used just to instruct the compiler that the function is a pure virtual function and it will not have a definition.
- The classes derived from pure abstract classes are required to redeclare the pure virtual function.
- All these derived classes which redefine the pure virtual function are called as concrete classes.
- These classes can be used to declare objects.

Ex:

```
#include<iostream.h>
#include<conio.h>
class first
{
        protected:
                int b;
        public:
                first()
                {
                b=10;
                }
        virtual void display()  = 0;
};
class second : public first
{
        int d;
        public:
                second()
                {
                        d=20;
                }
                void display()
                {
                        cout<<b<<d;
                }
};
void main()
{
        clrscr();
        first *p;
        second s;
        p=&s;
        p->display();
}
```

9.9 Object Slicing:

- A virtual function can be invoke using pointer or reference. If we do so object slicing takes place.

```
#include<iostream.h>
#include<conio.h>
class A
{
	public:
		int a;
	A()
	{
		a=10;
	}
};
class B : public A
{
	public:
		int b;
	B()
	{
		a=40;b=30;
	}
};
void main()
{
	clrscr();
	A x;
	B y;
	cout<<x.a;
	x=y;
	cout<<x.a;
}
```

9.10 VTABLE & VPTR:

- To perform late binding the compiler establishes a virtual table (VTABLE) for every class and its derived classes having virtual function.
- The VTABLE contains addresses of the virtual functions.
- The compiler puts each virtual function address in the VTABLE.
- If no function is redefined in the derived class ,i.e defined as virtual in the base class the compiler take address of the base class function.
- If it is redefined in the derived class, the compiler takes the address of derived class function.
- When objects of base or derived classes are created a void pointer is inserted in the VTABLE called VPTR (virtual pointer).
- This VPTR points to VTABLE.

9.11 new & delete operator:

- The new operator not only creates the object but also allocates memory.
- It allocates correct amount of memory from the heap that is also called as a free store.
- The delete operator not only destroys the object but also releases allocated memory.
- The object created and memory allocated by using new operator should be deleted by delete operator . Otherwise such mismatch operations may corrupt the heap or may crush the system.
- The compiler should have routines to handle such errors.
- The object created by new operator remains in memory until it is released by delete operator.
- Don't apply C functions such as malloc (), realloc () & free () with new and delete operators. These functions are unfit to object oriented techniques.
- Don't destroy the pointer repetitively. The statement delete x does not destroy the pointer x but destroys the object associated with it.
- If object is created but not deleted it occupies unnecessary memory. So it is a good habit to destroy the object & release memory.

Ex:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int *p;
        p=new int[3];
        cout<<"Enter 3 integers :";
        cin>>*p>>*(p+1)>>*(p+2);
        for(int i=0; i<3; i++)
        cout<<*(p+i)<< (unsigned) (p+i);
        delete [] p;
}
```

9.12 Dynamic Object:

- An object that is created at run time is called as a dynamic object.
- The construction & destruction of dynamic object is explicitly done by the programmer using new and delete operators.
- A dynamic object can be created using new operator.

  ptr = new classname;
- Where the variable ptr is a pointer object of the same class the new operator returns the object created and it is stored in the pointer ptr.
- delete object syntax is : delete ptr;

Ex:

```
#include<iostream.h>
#include<conio.h>
class data
{
        int x,y;
        public:
                data()
                {
                        cout<<"Enter the values for x & y :";
                        cin>>x>>y;
                }
                ~data()
                {
                        cout<<"Destructor";
                }
                void display()
                {
                        cout<<x<<y;
                }
};
void main()
{
        clrscr();
        data *d;
        d=new data;
        d->display();
        delete d;
}
```

/* Write a program to define virtual and non-virtual functions and determine the size of objects.*/

```
#include<iostream.h>
#include<conio.h>
class A
{
        private:
                int j;
        public:
                virtual void show()
                {
                        cout<<"In class A";
                }
```

```
};
class B
{
        private:
                int j;
        public:
                void show()
                {
                cout<<"In class B";
                }
};
class C
{
        public:
                void show()
                {
                        cout<<"In class C";
                }
};
void main()
{
        clrscr();
        A x;
        B y;
        C z;
        cout<<sizeof(x);
        cout<<sizeof(y);
        cout<<sizeof(z);
}
```

9.13 Heap:

- It is a huge section of memory in which large number of memory locations is placed sequentially.
- It is used to allocate memory during program execution or runtime.
- Local variables are stored in the stack and code in code space.
- Local variables are destroyed when a function returns.
- Global variables are stored in the data area. They are accessible by all functions.
- The heap is not declared until program execution completes. It is a user's task to free the memory.
- Memory allocated from heap remains available until the user explicitly deallocates it.

9.14 Virtual Destructors:

- The constructor can't be virtual since it requires information about the accurate type of object in order to construct it properly, but destructor can be declared as virtual and implemented like virtual functions.
- A derived class object is constructed using new operator.
- The base class pointer object force the address of the derived object.
- When this base class pointer is destroyed using delete operator the destructor of base and derived classes is executed.

Ex:

```
#include<iostream.h>
#include<conio.h>
class B
{
        public:
                B()
                {
                        cout<<"Class B constructor";
                }
                virtual ~B()
                {
                        cout<<"In class B destructor";
                }
};
class D : public B
{
        public:
                D()
                {
                        cout<<"Class D constructor ";
                }
                ~D()
                {
                        cout<<"In class D destructor";
                }
};
void main()
{
        clrscr();
        B *p;
        p=new D;
        delete p;
        getch();
}
```

Advantage:

- Virtual destructors are useful when a derived class object is pointed by the base class pointer object in order to invoke the base class destructor.

EXCEPTION HANDLING

10.1 Multiple catch statement

```
 #include<iostream.h>
 #include<conio.h>
 void num(int k)
 {
 try
 {
 if(k==0)throw k;
 else
 if(k>0)throw 'p';
 else
 if(k<0)throw .0;
 }
 catch(char g)
 {
        cout<<"\n caught a positive value";
 }
 catch(int j)
 {
       cout<<"\n caught a null value";
 }
 catch(double f)
 {
         cout<<"\n caught a negative value";
 }
}
int main()
{
 num(0);
 num(-5);
 num(-1);
 getch();
}
```

10.2 Catching Multiple Exception\ Generic catch block

```
#include<iostream.h>
#include<conio.h>
void num(int k)
{
 try
 {
 if(k==0)throw k;
```

```
    else
    if(k>0)throw 'p';
    else
    if(k<0)throw .0;
    cout<<"\n ---------TRY BLOCK-----------";
    }
    catch(...)
    {
            cout<<"\n caught an exception";
    }

}
int main()
{
   num(0);
   num(5);
   num(-1);
   getch();
}
```

10.3 Rethrowing an Exception

```
#include<iostream.h>
#include<conio.h>
void sub(int j,int k)
{
    try
    {
          if(j==0)
          throw j;
          else
          cout<<"\n subtraction:"<<j-k;
    }
    catch(int)
    {
           cout<<"caught a null value";
           throw;
    }
}
int main()
{
   try
   {
       sub(8,5);
       sub(0,8);
   }
   catch(int)
```

```
    {
            cout<<"caught a null value inside main";
    }
    getch();
}
```

10.4 Specifying Exception

```
#include<iostream.h>
#include<conio.h>
class num
{
     public:
          float a;
          void get()
          {
              cout<<"enter a number:";
              cin>>a;
          }
          friend void result(num &,num &);
};
void result(num &n,num &m)
{
     float r;
     r=n.a/m.a;
     cout<<"division is"<<r;
}
int main()
{
    num o1,o2;
    o1.get();
    o2.get();
    try
    {
          if(o2.a==0)
          throw (o2.a);
          else
          result(o1,o2);
    }
    catch(float k)
    {
             cout<<"exception caught:"<<k;
     }
     getch();
}
```

TEMPLATES

11.1 Introduction

- The templates provide generic programming by defining generic classes.
- A function that works for all C++ data types is called as a generic function.
- In templates generic data types are used as arguments and they can handle a verity of data types.
- Templates helps the programmer to declare a group of functions or classes. When used with functions they are called function templates.
  Ex:
  We can create a template for function square to find the square of an data type including int , float, long & double. The templates associated with classes are called as class template.

11.2 Need of Template:

- A template is a technique that allows using a single function or class to work with different data types.
- Using template we can create a single function that can process any type of data that is the formal arguments of template functions are of template type.
- So they can accept data of any type such as int, long, float, etc. Thus a single function can be used to accept values of different data type.
- Normally we overload functions when we need to handle different data type. But this approach increases the program size and also more local variables are created in memory.
- A template safely overcomes all this limitations and allows better flexibility to the program.
  Ex:

```
#include <iostream.h>
#include <conio.h>
template <class T>
class data
{
        Public:
                data (T c)
                {
                        cout<<c<<sizeof(c);
                }
};
void  main()
{
        clrscr();
        data <char> h('A');
        data <int > i(100);
        data <float> j(3.12);
        getch();
}
```

11.3 Normal Function Template:

- A normal function is not a member function of any class.
- The difference between a normal and member function is that :-
  - ✓ Normal functions are defined outside the class.
  - ✓ They are not members of any class and hence can be invoke directly without using object and dot operator.
  - ✓ But member functions are class members and hence need to be invoke using objects of the class to which they belong.

Ex:

/* Write a program to find square of a number using normal template function*/

```
#include <iostream.h>
#incude <conio.h>
template <class T>
void square (T x)
{
        cout<<"square="<<x*x;
}
void main()
{
        clrscr();
        int i; char j; double k;
        cout<<"Enter values for i, j & k :";
        cin>>i>>j>>k;
        square(i);
        square(j);
        square(k);
        getch();
}
```

11.4 Member Function Template:

```
#include <iostream.h>
#include <conio.h>
template <class T>
class sqr
{
        public:
                sqr (T c)
```

```
        {
                cout<<"Square="<<c*c;
        }
};
```

11.5 Working of Function Templates:

- After compilation the compiler can't case with which type of data the template function will work.
- When the template function called at that moment from the type of argument passed to the template function, the compiler identified the data type.
- Every argument of the template type is then replaced with the identified data type and this process is called as instantiating.
- So according to different data types respective versions of template functions are created.
- The programmer need not write separate functions for each data type.

```
/* Write a program to define data members of template types */
#include <iostream.h>
#include <conio.h>
template <calss T>
class data
{
        T x;
        public:
                data (T u)
                {
                        X=u;
                }
                void show (T y)
                {
                        cout<<x<<y;
                }
};
void main()
{
        Clrscr();
        data<char> C ('B');
        data<int> i (100);
        data<double>d (48.25);
        c.show ('A');
        i.show (65);
        d.show (68.25);
}
```

/* Write a program to define a constructor with multiple template variables */

```
#include <iostream.h>
#include <conio.h>
template <class T1,class T2>
class data
{
        Public:
                Data(T1 a,T2 b)
                {
                        cout<<a<<b;
                }
};
void main()
{
        clrscr();
        data <int, float> h(2,2.5);
        data <int, char> i(15,'c');
        data < float, int > j(3.12,50);
}
```

11.6 Overloading of Template Functions :

- Template functions can be overloaded by normal function or template function.
- While invoking these functions an error occurs if no accurate match is made.
- No implicit conversion is carried out in parameters of template function.

Ex:

```
#include <iostream.h>
#include <conio.h>
template <calss T>
void show (T c)
{
        cout<<"Template variable c="<<c;
}
void show(int f)
{
        cout<<"Integer variable f="<<f;
}
void main()
{
        clrscr ();
        show ('c');
        show (50);
        show (50.25);
```

```
}

/* Member function templates */

#include <iostream.h>
#include <conio.h>
template <calss T>
class data
{
        public:
                data (T c);
};
template <class T>
data <T> :: data (T c)
{
        cout<<"c="<<c;
}
void main()
{
        clrscr();
        data <char> h ('A');
        data <int> i (100);
        data <float> j (3.12);
}
```

11.7 Exception Handling With Class Template:

```
#include <iostream.h>
#include <conio.h>
class sq{};
template <calss T>
class square
{
        T s;
        public:
                square (T x)
                {
                        sizeof(x)==1 ? throw sq() : s=x*x;
                }
        void show()
        {
                cout<<"square="<<s;
        }
};
```

```
void main()
{
        try
        {
                square <int> i(2);
                i.show();
                square <char> c('c');
                c.show();
        }
        catch(sq)
        {
                cout<<"Square of character can't be calculated.";
        }
}
```

11.8 Class Templates with Overloaded Operators :

```
#include <iostream.h>
#include <conio.h>
template <calss T>
class num
{
        private:
                T number;
        public:
                num()
                {
                        number=0;
                }
                void input()
                {
                        cout<<"Enter a number :";
                        cin>>number;
                }
                num operator +(num);
                void show()
                {
                        cout<<number;
                }
};
template <class T>
num <T> num <T> :: operator +(num <T>c)
{
        num <T> tmp;
```

```
        tmp.number = number + c.number;
        return (tmp);
}
void main()
{
        clrscr();
        num <int> n1,n2,n3;
        n1.input();
        n2.input();
        n3=n1+n2;
        cout<<"n3=";
        n3.show();
        getch();
}
```

11.9 Class Template and Inheritance:

- The template class can also act as base class.
- There are 3 cases:-
  - ✓ Derive a template class and add new member to it, the base class must be of template type.
  - ✓ Derive a class from non-template class add new template type members to derive class.
  - ✓ Derive a class from a template base class and omit the template features in the derive class.
- This can be done by declaring the type while deriving the class.
- All the template based variables are substituted with basic data type.

How To Derive A Class Using A Template Base Class:

```
#include <iostream.h>
#include <conio.h>
template <calss T>
class one
{
        protected:
                T x,y;
        void display()
        {
                cout<<x<<y;
        }
};
template <class S>
class two : public one <S>
{
        S z;
        public:
                two (S a, S b, S c)
                {
```

```
                x=a; y=b; z=c;
        }
        void show ()
        {
                cout<<x<<y<<z;
        }
};
void main()
{
        clrscr();
        two <int> i (2,3,4);
        i.show();
        two <float> f (1.1,2.2,3.3);
        f.show();
}
```

11.10 Difference between Templates and Macros:

- Macros are not type safe. That is a macro defined for integer operation can't accept float data.
- They are expanded with no type checking.
- It is difficult to find errors in macros.
- In case a variable is post incremented (a++) or post decremented (a--) the operation is carried out twice for a macro.

11.11 Guidelines For Templates:

- Templates are applicable when we want to create type secure classes that can handle different data types with same member functions.
- The template classes can also be involved in inheritance.
- The template variables allow us to assign default values.

```
template <class T, int x=20>
class data
{
        T num[x];
}
```

- All template arguments declared in the template argument list should be used for definition of formal arguments. Otherwise it will give compilation error.
    - ✓
```
template <class T>
T show()
{
        return x;
}
```

    - ✓
```
template <class T>
void show(int y)
```

```
{
        T tmp;
}
```

NAMESPACES

12.1 Namespace Scope:

- C++ allows variables with different scopes such as local, global, etc with different blocks and classes. This can be done using keyword namespace.
- All classes, templates & functions are defined inside the name space std.
- The statement using namespace std tells the compiler that the members of this namespace are to be used in the current program.

12.2 Namespace Declaration:

- It is same as the class declaration, except that the name spaces are not terminated by (;) semicolons.

Ex:

```
namespace num
{
   int n;
   void show (int k)
   {
        cout<<k;
   }
}
num :: n=50;
using namespace num
n=10;
show (15) ;
```

12.3 Accessing elements of a name space:

1) Using Directive
2) Using Declaration

1) Using Directive:

- This method provides access to all variables declared with in the name space.
- Here we can directly access the variable without specifying the namespace name.

Ex:

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

```
namespace num
{
   int n;
   void show()
   {
            cout<<”n=”<<n;
   }
}
int main ()
{
   using namespace num;
   cout<<”Enter a number :”;
   cin>>n;
   show();
   getch();
}
```

2) **Using Declaratives**:

- Here the namespace members are accessed through scope resolution operator and name space name.

```
#include <iostream.h>
#include<conio.h>
namespace num
{
        int n;
        void show()
        {
                cout<<”n=”<<n;
        }
}
int main()
{
        cout<<”Enter value of n :”;
        cin>>num :: n;
        num :: show();
        getch();
}
```

12.4 Nested namespace:

- When one name space is declared inside another namespace it is known as nested namespace.

12.5 Anonymous namespaces:

- A namespace without name is known as anonymous namespaces.
- The members of anonymous namespaces can be accessed globally in all scopes.
- Each file posses separate anonymous namespaces.

```
#include<iostream.h>
#include<conio.h>
namespace    num
{
    int j=200;
    namespace num1 // nested namespace
    { int k=400;}
}
namespace //anonymous namespace
{int j=500;}
void main()
{
    cout<<" j= "<<num :: j;
    cout<<" k= "<<num :: num1 :: k;
    cout<<" j= "<<j;
}
```

12. 6 Function in namespace:

```
#include<iostream.h>
#include<conio.h>
namespace fun
{
        int add (int a,int b)
        {
                Return (a+b);
        }
        int mul (int a,int b)
}
int fun :: mul (int a, int b)
{ return (a*b);}
int main()
{
        using namespace fun;
        cout<<"Addition :"<<add(20,5);
        cout<<" Multiplication :"<<mul(20,5);
}
```

12.7 Classes in namespace:

```
#include<iostream.h>
#include<conio.h>
namespace A
{
	class num
	{
		private:
			int t;
		public:
			num (int m)
			{
				T=m;
			}
			void show()
			{
				Cout<<t;
			}
	};
}
void main()
{
//indirect access using scope access operator
	A :: num n1 (500);
	n1.show();
//direct access using directive
	using namespace A;
	num n2(800);
	n2.show();
}
```

12.8 Namespace Alias:

- It is designed to specify another name to existing namespace.
- It is useful if the previous name is long.
- We can specify a short name as alias and call the namespace members.

```
#include<iostream.h>
#include<conio.h>
namespace number
{
	Int n;
	Void show()
	{
		Cout<<" n= "<<n;
```

```
        }
}
Int main()
{
        namespace num=number;
        num :: n = 200;
        number :: show ();
}
```

- The standard namespace std contains all classes, templates and functions needed by a program.

12.9 Explicit Keyword :

- It is to declare class constructors to be explicit constructors.
- Any constructor called with one argument performs implicit conversion in which the type received by the constructor is converted to an object of the class in which the constructor is define. This conversion is automatic.
- If we don’t wants such automatic conversion to take place. We may da so by declaring the onr argument constructor as explicit.

Ex:

```
class ABC
{
        Int m;
        public:
                explicit ABC (int i)
                {
                        m=i;
                }
_____________________
    _________
        __
};
void main()
{
        ABC abc1 (100);
        ABC abc2=100;
}
```

12.10 Mutable keyword:

- If we want to create a constant object or function but would like to modify a particular data item only, we can make that particular data item modifiable by declaring it as mutable.

Ex:

```
Class ABC
{
        private:
                mutable int m;
        public:
                explicit ABC (int x=0)
                {
                        m=x;
                }
        void change () const
        {
                m=m+10;
        }
        int display () const
        {
                return m;
        }
}
void main()
{
        const ABC abc (100);
        abc.display();
        abc.change();
        abc.display();
}
```

12.11 Manipulating Strings:

- A string is a sequence of character we use null terminated character arrayes to store and manipulate strings. These strings are called C strings or C style string.
- ANSI standard C++ provides a new class called string. This class is very large and includes many constructors, member functions and operators.
- For using the string class we must include the string data type in the program.

Creating String Objects:

1) string s1; (null string) //using constructor with no argument
2) string s2(“xyz”); //using one argument constructor
3) s1=s2; //assigning string objects
4) cin>>s1; //reading through keyboard
5) getline (cin s1);

12. 12 Manipulating String Objects:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
        string  s1(“12345”);
        string  s2(“abcde”);
        cout<<s1<<s2;
        s1.insert(4,s2);
        cout<<s1;
        s1.erase(4,5);
        cout<<s1;
        s2.replace(1,3,s1);
        cout<<s2;
}
```

12.13 Relational Operator:

- These operators are overloaded and can be used to compare string objects.

```
void main()
{
        string  s1(“ABC”);
        string  s2(“XYZ”);
        string  s3=s1+s2;
        if(s1 != s2)
                cout<<”s1 is not equal to s2.”;
        if(s1 > s2)
                cout<<”s1 greater than s2.”;
        else
                cout<<”s2 greater than s1.”;
        int x=s1 . compare (s2);
        if(x==0)
                cout<<”s1 == s2”;
        else if (x>0)
                cout<<”s1 > s2”;
        else
                cout<<”s1 < s2”;
}
```

12.14 Accessing Characters In String:

- They are used to access sub strings and individual characters of a string.

Ex:

```
int main()
{
string  s(“one two three four”);
for(int i=0;i<s.length();i++)
cout<<s.at(i);
for(int j=0;j<s.length();j++)
cout<<s[j];
int x1=s.find(“two”);
cout<<x1;
int x2=s.find-first-of(‘t’);
cout<<x2;
int x3=s.find-last-of(‘r’);
cout<<x3;
cout<<s.substr(4,3);
}
string  s1(“Road”);
string  s2(“Read”);
s1.swap(s2);
```

STANDARD TEMPLATE LIBRARY(STL)

- In order to held the C++ user in generic programming Alexander Steponor & Meng Lee of P developed a set of general purpose templatised classes & function that could be used as standard approach for storing and processing of data.
- The collect of these generic classes & functions is called the STL.

13.1 Components of STL

13.1.1 Container

- A container is an object that actually stores data.
- It is a way data is organized in memory.
- The STL containers are implemented by template classes & can be easily customized to hold different types of data.

13.1.2 Algorithm

- It is a procedure i.e used to process the data contained in the containers.
- STL includes many different algorithm to provide support to take such as initializing, searching, popping, sorting, merging, copying.
- They are implemented by template functions.

13.1.3 Iterator

- It is an object like a pointer that points to an element in a container.
- We can use iterator to move through the contains of container.
- They are handle just like pointers we can increment or decrement them.

13.2 Types of containers

13.2.1 Sequence Containers

- They stored elements in a linear sequence like a line.
- Each element is related to other elements by its position along the line.
- They all expand themselves through allow insertion of elements & support a no. of operation.

Vector –

- It is a dynamic array.
- It allows insertion & deletion at back & permits direct access to any element.

List –

- It is a bidirectional linear list.
- It allows insertion & deletion any where.

Dequeue –

- It is a double ended queue.
- It allows insertion & deletion at both ends.

13.2.2 Associative Container

- They are design to support direct access to elements using keys.
- They are 4 types.

Set

- It is an associative container for storing unique sets.
- Here, is no duplicate are allowed.

Multisets

- Duplicate are allowed.

Map

- It is an associate container for storing unique key.
- Each key is associated with one value.

Multimap

- It is an associate container for storing key value pairs in which one key may be associated with more then one value.
- We can search for a desired student using his name as the key.
- The main difference between a map & multimap is that, a map allows only one key for a given value to be stored while multimap permits multiple key.

13.2.3 Derived Container

- STL provides 3 derived container, stack, queue, priority queue. They are also known as container adaptor.
- They can be created from different sequence container.

Stack – it is a LIFO list.
Queue – it is a FIFO list.
Priority queue – it is a queue where the 1st element out is always the highest priority queue.

13.3 Algorithms

- A large number of algorithms to perform activities such as searching and sorting are provided in the STL, each implemented to require a certain level of iterator (and therefore will work on any container that provides an interface by iterators).
- Searching algorithms like binary search and lower bound use binary search and like sorting algorithms require that the type of data must implement comparison operator < or custom comparator function must be specified;
- such comparison operator or comparator function must guarantee strict.
- Apart from these, algorithms are provided for making heap from a range of elements, generating lexicographically ordered permutations of a range of elements, merge sorted ranges and perform union, intersection, difference of sorted ranges.

13.4 Iterators

- The STL implements five different types of iterators.
- These are input iterators (that can only be used to read a sequence of values), output iterators (that can only be used to write a sequence of values), forward iterators (that can be read, written to, and move forward), bidirectional iterators (that are like forward iterators, but can also move backwards) and random access iterators (that can move freely any number of steps in one operation).
- It is possible to have bidirectional iterators act like random access iterators, as moving forward ten steps could be done by simply moving forward a step at a time a total of ten times.
- However, having distinct random access iterators offers efficiency advantages. For example, a vector would have a random access iterator, but a list only a bidirectional iterator.
- Iterators are the major feature that allow the generality of the STL.
  For example, an algorithm to reverse a sequence can be implemented using bidirectional iterators, and then the same implementation can be used on lists, vectors and deques.

- User-created containers only have to provide an iterator that implements one of the five standard iterator interfaces, and all the algorithms provided in the STL can be used on the container.
- This generality also comes at a price at times.
  For example, performing a search on an associative container such as a map or set can be much slower using iterators than by calling member functions offered by the container itself. This is because an associative container's methods can take advantage of knowledge of the internal structure, which is opaque to algorithms using iterators.