

CBCS SCHEME

USN

ICR13MCA11

18MCA11

First Semester MCA Degree Examination, Dec.2019/Jan.2020 Object Oriented Programming with C++

Time: 3 hrs.

Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

- a. Explain the salient features of object-oriented programming languages. (07 Marks)

b. Differentiate between procedural-oriented programming and object-oriented programming. (07 Marks)

c. What are inline functions? Discuss the advantages of inline functions with example. (06 Marks)

OR

- a. Write a C++ program to calculate the volume of different geometric shapes with the concept of function overloading. (06 Marks)

b. Define default arguments. Demonstrate with an example program. (07 Marks)

c. What are function templates? Explain with an example of Bubble sort program for integers and doubles. (07 Marks)

Module-2

- a. Define static data members and static member functions of a class. Explain with example program. (08 Marks)

b. What are constructors and destructors? Explain different types of constructors with example program. (08 Marks)

c. Explain the use of scope resolution operator. (04 Marks)

OR

- a. Explain dynamic memory allocation operators. Demonstrate with an example program. (07 Marks)

b. Illustrate how you can allocate objects dynamically with an example program. (08 Marks)

c. Write a C++ program to swap two numbers using pointers. (05 Marks)

Module-3

- a. Write a C++ program to create a class called MATRIX using two dimensional array of integers. Implement the following operations by overloading the operators:
i) \times operator to check the capability of two matrices
ii) $+$ operator to perform addition of two matrices
iii) $-$ operator to perform subtraction of two matrices.
Note: If $(m1 \times = m2)$ then $m3 = m1 + m2$ and $m4 = m1 - m2$ else display error message. (10 Marks)

b. Demonstrate overloading of $++$ and $--$ operators using friend function. (06 Marks)

c. What are the restrictions for operator overloading. (04 Marks)

OR

- a. Explain the base class access specifiers with example for each. (08 Marks)

b. Explain how to pass parameters to base class constructors with example program. (05 Marks)

c. Explain virtual base classes with example program. (06 Marks)

1 of 2

Important Note: 1. On completing your answers, compulsorily draw diagonal lines across the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and/or equivalent written up, (2+1+1+1), will be treated as malpractice.

Module-4

- 7 a. Define virtual functions. Demonstrate how to call virtual functions through base class. (10 Marks)
- b. What are abstract classes? Explain with example program. (07 Marks)
- c. Differentiate between early and late binding. (03 Marks)

OR

- 8 a. Explain formatting I/O using ios members. (07 Marks)
- b. Define extractors and inserters. Explain how to create. Your own extractors and inserters with example program. (10 Marks)
- c. Explain setting and clearing format tags. (03 Marks)

Module-5

- 9 a. Explain exception handling in C++. Write a C++ program to demonstrate multiple catch statements. (10 Marks)
- b. Explain how to restrict the exceptions and rethrow the exceptions with example program for each. (10 Marks)

OR

- 10 a. Define class templates. Explain with an example program of two generic data types. (07 Marks)
- b. Explain categories of containers in STL. (09 Marks)
- c. Write a short notes on vector class. (04 Marks)

.....

Class:

1. Class is user defined data type and behave like the built-in data type of a programming language.
2. Class is a blue print/model for creating objects.
3. Class specifies the properties and actions of an object.
4. Class does not physically exist.
5. Once a class has been defined, we create any number of objects belonging to that class. Thus, class is a collection of objects of similar type.

Object:

1. Object is the basic run time entities in an object oriented system.
2. Object is the basic unit that are associated with the data and methods of a class.
3. Object is an instance of a particular class.
4. Object physically exists.
5. Objects take up space in memory and have an associated address.
6. Objects communicate by sending messages to one another.

Data Abstraction and Encapsulation:

Abstraction refers to the act of representing essential features without including the background details. In programming languages, data abstraction will be specified by abstract data types and can be achieved through classes.

The wrapping up of data and functions into a single unit is known as encapsulation. It keeps them safe from external interface and misuse as the functions that are wrapped in class can access it. The insulation of the data from direct access by the program is called data hiding.

Inheritance:

1. It provides the concept of reusability.
2. It is a mechanism of creating new classes from the existing classes.
3. It supports the concept of hierarchical classification.
4. A class which provides the properties is called Parent/Super/Base class.
5. A class which acquires the properties is called Child/Sub/Derived class.
6. A sub class defines only those features that are unique to it.

Polymorphism:

1. Polymorphism is derived from two greek words Poly and Morphis where poly means many and morphis means forms.
2. Polymorphism means one thing existing in many forms.
3. Polymorphism plays an important role in allowing objects having different internal structures to share the same external interfaces.
4. Polymorphism is extensively used in implementing inheritance.
5. Function overloading and operator overloading can be used to achieve polymorphism.

Dynamic Binding:




1. Binding refers to the linking of a procedure call to be executed in response to the call.
2. If the binding occurs at runtime then it is called as dynamic binding.
3. It is also called as late binding as binding of a call to the procedure is not known until runtime.
4. Dynamic Binding is associated with polymorphism and inheritance.

Message Binding:

1. Objects communicate with each other by sending and receiving information using functions.
 2. The basic steps to perform message passing are
- * Creating classes that define objects and their behavior.

- * Creating objects from class definitions.
 - * Establishing communication among objects.
3. Message passing involves specifying the name of the object, name of the function and the information to be sent as
 objectname.functionname(message);
4. A message for an object is a request for execution of a procedure and therefore will invoke a function in receiving object that generates the desired result.
5. Communication with an object is feasible as long as it is alive.

1.b) Difference between POP and OOP (7)

	Procedure Oriented Programming	Object Oriented Programming  
Divided Into	In POP, program is divided into small parts called functions .	In OOP, program is divided into parts called objects .
Importance	In POP, importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, importance is given to the data rather than procedures or functions because it works as a real world .
Approach	POP follows Top Down approach .	OOP follows Bottom Up approach .
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

1.c) Inline function with example (6)

Ans:Inline Functions:

If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

Ex:

```
#include <iostream>

using namespace std;

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

// Main function for the program
int main( )
{
    cout << "Max (20,10): " << Max(20,10) << endl;
    cout << "Max (0,200): " << Max(0,200) << endl;
    cout << "Max (100,1010): " << Max(100,1010) << endl;
    return 0;
}
```

2a) c++ program to calculate the volume of different geometric shapes with the concept of function overloading (6)

```
#include<iostream>
using namespace std;

int volume(int n)
{
    return n*n*n;
}
double volume(double r, double h)
{
    Return3.14*r* r*h;
}
double volume(double ra)
{
    return 0.33*3.14*ra*ra*ra;
}
int main()
{
    int s,ra,rb;
    double r,h;
    cout<<" Enter the value of side in Integer to calculate the volume of cube:\n";
```

```

    cin>>s;
    cout<<" Volume of Cube is :"<<volume(s)<<endl;
    cout<<" Enter the value of radius and height in Double to calculate the volume of
cylinder:\n";
    cin>>r>>h;
    cout<<" Volume of Cylinder is :"<<volume(r,h)<<endl;
    cout<<" Enter the value of radius in Double to calculate the volume of Sphere:\n";
    cin>>ra>>rb;
    cout<<" Volume of Sphere is :"<<volume(ra)<<endl;
    return 0;
}

```

2.b) Default argument – example

(7)

A **default argument** is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the **argument** with a **default** value. Following is a simple C++ example to demonstrate the use of **default arguments**.

```

#include <iostream>
using namespace std;
void sum(int a,int b= 6);
int main( )
{
    int a,b;
    cout<<"enter any two numbers\n";
    cin>>a>>b;
    sum(a) ; // sum of default values
    sum(a,b);
    sum(b);
    return 0;
}
void sum (int a1, int a2)
{
    int temp;
    temp = a1 + a2;
    cout<<"a="<<a1<<endl;
    cout<<"b="<<a2<<endl;
    cout<<"sum="<<temp<<endl;
}

```

O/P

enter any two numbers

2 3

a=2

b=6

sum=8

a=2

b=3

sum=5

```
a=3
b=6
sum=9
```

2c) Function template – bubble sort

(7)

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. Many algorithms are logically the same no matter what type of data is being operated upon.

For example, the bubble sort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself. A generic function is created using the keyword template.

The general form of a template function definition is shown here:

```
template <class Ttype> ret-type func-name(parameter list)
{
// body of function
}
#include<conio.h>
#include<iostream.h>

template<class bubble>
void bubble(bubble a[], int n)
{
    int i, j;
    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(a[i]>a[j])
            {
                bubble element;
                element = a[i];
                a[i] = a[j];
                a[j] = element;
            }
        }
    }
}
```

```
void main()
{
    int a[6]={1,2,3,4,4,3};
    char b[4]={'s','b','d','e'};
```

```

clrscr();
bubble(a,6);
cout<<"\nSorted Order Integers: ";
for(int i=0;i<6;i++)
    cout<<a[i]<<"\t";
bubble(b,4);
cout<<"\nSorted Order Characters: ";
for(int j=0;j<4;j++)
    cout<<b[j]<<"\t";
getch();

```

3.a) Static data members and static member function of class – Example (6)

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable. All static variables are initialized to zero before the first object is created. When you declare a static data member within a class, you are not defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. (Remember, a class declaration is simply a logical construct that does not have physical reality.)

Example:

```

#include <iostream>
using namespace std;
class shared {
static int a;
int b;
public:
void set(int i, int j) {a=i; b=j;}
void show();
};
int shared::a; // define a
void shared::show()
{
cout << "This is static a: " << a;
cout << "\nThis is non-static b: " << b;
cout << "\n";
}
int main()
{
shared x, y;
x.set(1, 1); // set a to 1
x.show();
}

```



```

y.set(2, 2); // change a to 2
y.show();
x.show(); /* Here, a has been changed for both x and y
because a is shared by both objects. */
return 0;
}

```

This program displays the following output when run.

```

This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1

```

Static Member Function:

Member functions may also be declared as static. There are several restrictions placed on static member functions. They may only directly refer to other static members of the class. (Of course, global functions and data may be accessed by static member functions.)

A static member function does not have a this pointer. There cannot be a static and a non-static version of the same function. A static member function may not be virtual. Finally, they cannot be declared as const or volatile.

Example:

```

#include <iostream>
using namespace std;
class c1 {
static int resource;
public:
static int get_resource();
void free_resource() { resource = 0; }
};
int c1::resource; // define resource
int c1::get_resource()
{
if(resource) return 0; // resource already in use
else {
resource = 1;
return 1; // resource allocated to this object
}
}
int main()
{
c1 ob1, ob2;
/* get_resource() is static so may be called independent
of any object. */
if(c1::get_resource()) cout << "ob1 has resource\n";
if(!c1::get_resource()) cout << "ob2 denied resource\n";
ob1.free_resource();
}

```

```

if(ob2.get_resource()) // can still call using object syntax
cout << "ob2 can now use resource\n";
return 0;
}

```

3b) constructors and destructors. Different types of constructors with example

Ans: Constructor is a special member of a class which is used to initialise the objects. Constructors will be called whenever we create objects in the program.

Characteristics of a constructor:

- * Constructor will be having the same class name.
- * Constructor will not have any return type. It will not be specified even with void.
- * Constructor may or may not have parameters.
- * Constructor should be declared in public section of a class.
- * Constructor can also be overloaded.

Types of constructors:

There are two types of constructors. They are

- * Default Constructors
- * Parameterised Constructors

Default Constructor:

Constructor with no parameters is called default constructor.

Syntax:

```

classname()
{
    body of the constructor
}

```

Parameterised Constructors:

Constructor with parameters is called parameterised constructor.

Syntax:

```

classname(parameter list)
{
    body of the constructor
}

```

```

#include<iostream>
using namespace std;

```

```

class Human{
int Head,Legs,Hands;

```

```

public: Human()
{
    Head=1;
    Legs=2;
    Hands=2;
}
Human(int n1,int n2,int n3=5)

```

```

    {
        Head=n1;
        Legs=n2;
        Hands=n3;
    }
    Human(Human&);
    void Display()
    {
        cout<<"Head="<<Head<<endl;
        cout<<"Legs="<<Legs<<endl;
        cout<<"Hands="<<Hands<<endl;
    }
    ~Human()
    {
        cout<<"Destructor Executing"<<endl;
        cout<<"Human Died"<<endl;
    }
};
Human :: Human(Human &h1)
{
    Head=h1.Head;
    Legs= h1.Legs;
    Hands=h1.Hands;
}

int main()
{
    Human Human1 ;
    Human Human2(2,4,2);
    Human Human3(3,6);
    Human Human4(Human1);
    cout<<"Object with Default Constructor"<<endl;
    Human1.Display();
    cout<<"Object with Parameterized Constructor"<<endl;
    Human2.Display();
    cout<<"Object with Parameterized Constructor with default argument "<<endl;
    Human3.Display();
    cout<<"Copy Constructor"<<endl;
    Human4.Display();
}

```

3c)use of scope resolution operator

- The :: (scope resolution) operator is used to get hidden names due to variable scopes so that you can still use them.

- The scope resolution operator can be used as both unary and binary.
- For example, if you have a global variable of name my_var and a local variable of name my_var, to access global my_var, you'll need to use the scope resolution operator.

```
#include <iostream>
using namespace std;

int my_var = 0;
int main(void) {
    int my_var = 0;
    ::my_var = 1; // set global my_var to 1
    my_var = 2; // set local my_var to 2
    cout << ::my_var << ", " << my_var;
    return 0;
}
```

If a class member name is hidden, you can use it by prefixing it with its class name and the class scope operator. For example,

Example

```
#include <iostream>
using namespace std;
class X {
public:
    static int count;
};
int X::count = 10; // define static data member

int main () {
    int X = 0; // hides class type X
    cout << X::count << endl; // use static member of class X
}
```

4a)dynamic memory allocation operator. Example

```
#include<iostream>
using namespace std;

void swap(int *x, int *y)
{
    int t;
    cout<<" Value of x and y in swap before exchange:" <<x<<" "<<y<<endl;
    t=*x;
    *x=*y;
    *y=t;
    cout<<" Value of x and y in swap after exchange:" <<x<<" "<<y<<endl;
}

int main()
```

```

{
    int a, b;
    int *p,*q;
    cout<<" Enter two integers:";
    cin>>a>>b;
try{
    p= new int(a);
    q= new int(b);
}
    swap(p,q);
    cout<<" Value of a and b in swap after exchange in function main():" <<*p<<"
"<<*q<<endl;
    delete p;
    delete q;
    return 0;
}

```

4b)allocate objects dynamically with example prg

Constructor is a member function of a class which is called whenever a new object is created of that class. It is used to initialize that object. Destructor is also a class member function which is called whenever the object goes out of scope.

Destructor is used to release the memory assigned to the object. It is called in these conditions.

When a local object goes out of scope

For a global object, operator is applied to a pointer to the object of the class

We again use pointers while dynamically allocating memory to objects.

```
#include <iostream>
```

```
using namespace std;
```

```

class A
{
    public:
                                A() {
        cout << "Constructor" << endl;
    }
                                ~A() {
        cout << "Destructor" << endl;
    }
};

```

```

int main()
{
    A* a = new A[4];
    delete [] a; // Delete array
    return 0;
}

```

```
}
```

4c) swap two numbers using points

```
#include<iostream>
using namespace std;

void swap(int *x, int *y)
{
    int t;
    cout<<" Value of x and y in swap before exchange:" <<x<<" "<<y<<endl;
    t=*x;
    *x=*y;
    *y=t;
    cout<<" Value of x and y in swap after exchange:" <<x<<" "<<y<<endl;
}

int main()
{
    int a, b;
    cout<<" Enter two integers:";
    cin>>a>>b;
    swap(&a,&b);
    cout<<" Value of a and b in swap after exchange in function main():" <<a<<"
"<<b<<endl;
    return 0;
}
```

5a)

```
Ans: #include<iostream>
#define Max 20
using namespace std;
class Matrix
{
    public:
    int a[Max][Max];
    int r,c;
    void getorder();
    void getdata();
    Matrix operator -(Matrix);
    friend ostream& operator <<(ostream &, Matrix);
    int operator==(Matrix);
};
void Matrix::getorder()
{
```

```

        cout<<"enter the number of rows\n";
        cin>>r;
        cout<<"enter the number of columns\n";
        cin>>c;
    }
    void Matrix::getdata()
    {
        int i,j;
        for(i=0;i<r;i++)
        {
            for(j=0;j<c;j++)
            {
                cin>>a[i][j];
            }
        }
    }
    Matrix Matrix::operator -(Matrix m2)
    {
        Matrix m4;
        int i,j;
        for(i=0;i<r;i++)
        {
            for(j=0;j<c;j++)
            {
                m4.a[i][j] = a[i][j] - m2.a[i][j];
            }
        }
        m4.r = r;
        m4.c = c;
        return m4;
    }
    ostream & operator <<(ostream & out, Matrix m)
    {
        int i,j;
        for(i=0;i<m.r;i++)
        {
            for(j=0;j<m.c;j++)
            {
                out<<m.a[i][j]<<"\t";
            }
            out<<endl;
        }
        return out;
    }
    int Matrix::operator==(Matrix m2)
    {
        if((r==m2.r) && (c==m2.c))
            return 1;
    }

```

```

        else
            return 0;
    }
int main()
{
    Matrix m1,m2,m4;
    cout<<"enter the order of the first matrix\n";
    m1.getorder();
    cout<<"enter the order of the second matrix\n";
    m2.getorder();
    if(m1 == m2)
    {
        cout<<"enter the elements of the first matrix\n";
        m1.getdata();
        cout<<"enter the elements of the second matrix\n";
        m2.getdata();
        m4 = m1 - m2;
        cout<<"Difference of matrices is \n";
        cout<<m4<<endl;
    }
    else
    {
        cout<<"Order of the matrices is not same";
    }
    return 0;
}

```

5b)

```

#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
// Overload assignment for loc.

```



```

loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{ op.longitude++;
op.latitude++;
return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
op.longitude--;
op.latitude--;
return op;
}
int main()
{
loc ob1(10, 20), ob2;
ob1.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob2.show(); // displays 12 22
--ob2;
ob2.show(); // displays 11 21
return 0;
}

```

5.c) Operator Overloading restrictions

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. **Operator that are not overloaded** are follows

- scope operator - ::
- **sizeof**
- member selector - .
- member pointer selector - *
- ternary operator - ?:

6a) Base class access specifier – example for each (9)

The access restriction to the class members is specified by the labeled public, private, and protected sections within the class body.

The keywords public, private, and protected are called access specifiers.

- A class can have multiple public, protected, or private labeled sections.

- When the access specifier for a base class is **public**, **all public members of the base become public members of the derived class**, and **all protected members of the base become protected members of the derived class**.
- When the base class is inherited by using the **private** access specifier, **all public and protected members of the base class become private members of the derived class**.
- When the base class is inherited by using the **protected** access specifier, **all public and protected members of the base class become protected members of the derived class**.

```

• #include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
// derived may access base's i and j
void setk() { k=i*j; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob;
ob.set(2, 3); // OK, known to derived
ob.show(); // OK, known to derived
ob.setk();
ob.showk();
return 0;
}

```

In this example, because base is inherited by derived as public and because i and j are declared as protected, derived's function setk() may access them. If i and j had been declared as private by base, then derived would not have access to them, and the program would not compile.

6b) pass parameters to base class constructors with example program

(5)

The general form

of this expanded derived-class constructor declaration is shown here:

```

derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// body of derived constructor
}

```

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes. Consider this program:

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) { i=x; cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
int j;
public:
// derived uses x; y is passed along to base.
derived(int x, int y): base(y)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**. However, **derived()** uses only **x**; **y** is passed along to **base()**. In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. As the example illustrates, any parameters required by the base class are passed to it in the base class' argument list specified after the colon.

6c) Virtual base classes with example program

(6)

- A *virtual function* is a member function that is declared within a base class and redefined by a derived class
- Precede the function's declaration in the base class with the keyword **virtual**.
- **When a class containing a virtual function is inherited**, the derived class redefines the virtual function to fit its own needs.
- Virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism.
- The virtual function within the base class defines the *form of the interface* to that function.
- Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.

```
include <iostream>
```

```

using namespace std;
class base {
public:
virtual void vfunc()
{
    cout << "This is base's vfunc().\n";
}
};
class derived1 : public base
{
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base
{
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};

```

```

int main()
{
base *p, b;
derived1 d1;
derived2 d2;
    // point to base
p = &b;
p->vfunc();    // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc();    // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc();    // access derived2's vfunc()
return 0;
}

```

```

7a)
#include <iostream>
using namespace std;
class base
{
    public:

```

```

        virtual void vfunc() {
            cout << "This is base's vfunc().\n";
        }
};

class derived1 : public base
{
    public:
    void vfunc()
    {

        cout << "This is derived1's vfunc().\n";
    }
};

class derived2 : public base
{
    public:
    void vfunc()
    {
        cout << "This is derived2's vfunc().\n";
    }
};

// Use a base class reference parameter.

void f(base &r)
{
    r.vfunc();
}

int main()
{
    base b;
    derived1 d1;
    derived2 d2;
    f(b); // pass a base object to f()
    f(d1); // pass a derived1 object to f()
    f(d2); // pass a derived2 object to f()
    return 0;
}

```

In this example, the function `f()` defines a reference parameter of type `base`. Inside `main()`, the function is called using objects of type `base`, `derived1`, and `derived2`. Inside `f()`, the specific version of `vfunc()` that is called is determined by the type of object being referenced when the function is called.

7b)

A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows –

```
class Box {
public:
    // pure virtual function
    virtual double getVolume() = 0;

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

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error. Classes that can be used to instantiate objects are called **concrete classes**.

Example

```
#include <iostream>

using namespace std;

// Base class
class Shape {
public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
        width = w;
    }

    void setHeight(int h) {
        height = h;
    }

protected:
    int width;
    int height;
};

// Derived classes
class Rectangle: public Shape {
public:
    int getArea() {
```

```

        return (width * height);
    }
};

class Triangle: public Shape {
public:
    int getArea() {
        return (width * height)/2;
    }
};

int main(void) {
    Rectangle Rect;
    Triangle Tri;

    Rect.setWidth(5);
    Rect.setHeight(7);

    // Print the area of the object.
    cout << "Total Rectangle area: " << Rect.getArea() << endl;

    Tri.setWidth(5);
    Tri.setHeight(7);

    // Print the area of the object.
    cout << "Total Triangle area: " << Tri.getArea() << endl;

    return 0;
}

```

7c)

Early Binding	Late Binding
1. Early binding refers to events that occur at compile time	1. Late binding refers to events that occur at run time
2. The information needed to call a function is known at compile time	2. The information needed to call a function is not known until run time
3. It is more efficient	3. It is more flexible
4. It is fast	4. It is slow
5. Example: function overloading	5. Example: virtual functions

8a)

- C++ I/O system allows you to format I/O operations.
 - set a field width,

- specify a number base
- determine how many digits after the decimal point will be displayed.
- There are two ways you can format data
- Directly access members of the **ios class**. Set various format status flags defined inside the ios class or call various ios member functions.
- Use special functions called *manipulators that can* be included as part of an I/O expression.
- Each stream has associated with it a set of format flags that control the way information is formatted.
- The **ios class declares a bitmask enumeration called fmtflags in which the** following values are defined

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

- **skipws flag is set, leading white-space characters (spaces, tabs, and newlines)** are discarded – i/p
- **skipws is cleared**, white-space characters are not discarded.
- **left flag is set, output is left justified.**
- **right is set, output is right justified.**
- **internal flag is set, a numeric value is padded to fill a field by inserting spaces** between any sign or base character.
- If none of these flags are set, output is right justified by default.
- **oct flag** causes output to be displayed in octal.
- **hex flag** causes output to be displayed in hexadecimal.
- To return output to decimal, set the **dec flag**.
- Setting **showbase causes the base of numeric values to be shown.**
- **For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.**
- By default, when scientific notation is displayed, the **e is in lowercase.**
- Hexadecimal value is displayed, the **x is in lowercase.**
- **Uppercase is set, these** characters are displayed in uppercase.
- Setting **showpos causes a leading plus sign to be displayed before positive values.**
- Setting **showpoint causes a decimal point and trailing zeros to be displayed for all** floating-point output—whether needed or not.
- Setting the **scientific flag, floating-point numeric values are displayed using** scientific notation.
- **Fixed is set**, floating-point values are displayed using normal notation.
- When **unitbuf is set**, the buffer is flushed after each insertion operation.
- When **boolalpha is set**, Booleans can be input or output using the keywords true and false.

- Since it is common to refer to the **oct, dec, and hex fields, they can be collectively** referred to as **basefield**.
- Similarly, the left, right, and internal fields can be referred to as **adjustfield**.
- Finally, the scientific and fixed fields can be referenced as floatfield.

8b)

: In the language of C++, the << output operator is referred to as the *insertion operator* because it inserts characters into a stream. Likewise, the >> input operator is called the *extraction operator* because it extracts characters from a stream.

Creating Your Own Inserters

It is quite simple to create an inserter for a class that you create. All inserter functions have this general form:

```
ostream &operator<<(ostream &stream, class_type obj)
{
    // body of inserter
    return stream;
}
```

the function returns a reference to a stream of type ostream. Further, the first parameter to the function is a reference to the output stream. The second parameter is the object being inserted.

Creating Your Own Extractors

Extractors are the complement of inserters. The general form of an extractor function is

```
istream &operator>>(istream &stream, class_type &obj)
{
    // body of extractor
    return stream;
}
```

Extractors return a reference to a stream of type istream, which is an input stream. The first parameter must also be a reference to a stream of type istream. Notice that the second parameter must be a reference to an object of the class for which the extractor is overloaded. This is so the object can be modified by the input (extraction) operation.

```
#include <iostream>
#include <cstring>
using namespace std;
class phonebook
{
    char name[80];
    int areacode;
    int prefix;
    int num;
```

```

public:
phonebook() { };
phonebook(char *n, int a, int p, int nm)
{
strcpy(name, n);
areacode = a;
prefix = p;
num = nm;
}
friend ostream &operator<<(ostream &stream, phonebook o);
friend istream &operator>>(istream &stream, phonebook &o);
};
// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
stream << o.name << " ";
stream << "(" << o.areacode << ") ";
stream << o.prefix << "- " << o.num << "\n";
return stream; // must return stream
}
// Input name and telephone number.
istream &operator>>(istream &stream, phonebook &o)
{
cout << "Enter name: ";
stream >> o.name;
cout << "Enter area code: ";
stream >> o.areacode;
cout << "Enter prefix: ";
stream >> o.prefix;
cout << "Enter number: ";
stream >> o.num;
cout << "\n";
return stream;
}
int main()
{
phonebook a;
cin >> a;
cout << a;
return 0;
}

```

8c)

- To set a flag, use the **setf() function**.
- **This function is a member of ios.**
 - `fmtflags setf(fmtflags flags);`

- **unsetf():** The unsetf method is used To remove the flag setting

This function returns the previous settings of the format flags and turns on those flags specified by *flags*

```
// using setw(), setiosflags(), showpos and internal
#include <iostream>
#include <iomanip>
using namespace std;
void main(void)
{
    cout<<setiosflags(ios::internal | ios::showpos)<<setw(12)<<12345<<endl;
}
```

9a)

Exception means run time errors. Exception handling allows you to manage run-time errors in an orderly fashion. Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.

C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The following discussion elaborates upon this general description.

Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.) Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown. The general form of **try** and **catch** are shown here.

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}...
catch (typeN arg) {
// catch block
}
```

The **try** can be as short as a few statements within one function or as all-encompassing as enclosing the **main()** function code within a **try** block (which effectively causes the entire program to be monitored).

When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement associated

with a **try**. Which **catch** statement is used is determined by the type of the exception. That is, if the data type specified by a **catch** matches that of the exception, then that **catch** statement is executed (and all others are bypassed). When an exception is caught, *arg* will receive its value. Any type of data may be caught, including classes that you create. If no exception is thrown (that is, no error occurs within the **try** block), then no **catch** statement is executed.

The general form of the **throw** statement is shown here:

```
throw exception;
```

Example

```
#include <iostream>
using namespace std;
void divide(double a, double b);
int main()
{
    double i, j;
    do {
        cout << "Enter numerator (0 to stop): ";
        cin >> i;
        cout << "Enter denominator: ";
        cin >> j;
        divide(i, j);
    } while(i != 0);
    return 0;
}
void divide(double a, double b)
{
    try {
        if(!b) throw b; // check for divide-by-zero
        cout << "Result: " << a/b << endl;
    }
    catch (double b) {
        cout << "Can't divide by zero.\n";
    }
}
```

9b)

If you wish to rethrow an expression from within an exception handler, you may do so by calling **throw**, by itself, with no exception. This causes the current exception to be passed on to an outer **try/catch** sequence. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a **catch** block (or from any function called from within that block). When you rethrow an exception, it will not be recaptured by the same **catch** statement. It will propagate outward to the next **catch** statement. The following program illustrates rethrowing an exception, in this case a **char*** exception.

```

// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
void Xhandler()
{
try {
throw "hello"; // throw a char *
}
catch(const char *) { // catch a char *
cout << "Caught char * inside Xhandler\n";
throw ; // rethrow char * out of function
}
}
int main()
{
cout << "Start\n";
try{
Xhandler();
}
catch(const char *) {
cout << "Caught char * inside main\n";
}
cout << "End";
return 0;
}

```

This program displays this output:

```

Start
Caught char * inside Xhandler
Caught char * inside main
End

```

Restricting Exceptions

You can restrict the type of exceptions that a function can throw outside of itself. In fact, you can also prevent a function from throwing any exceptions whatsoever. To accomplish these restrictions, you must add a **throw** clause to a function definition. The general form of this is shown here:

```

ret-type func-name(arg-list) throw(type-list)
{
// ...
}

```

Here, only those data types contained in the comma-separated *type-list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don't want a function to be able to throw *any* exceptions, then use an empty list.

Attempting to throw an exception that is not supported by a function will cause the standard library function **unexpected()** to be called. By default, this causes **abort()** to be called, which causes abnormal program termination.

The following program shows how to restrict the types of exceptions that can be thrown from a function.

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
    if(test==0) throw test; // throw int
    if(test==1) throw 'a'; // throw char
    if(test==2) throw 123.23; // throw double
}
int main()
{
    cout << "start\n";
    try{
        Xhandler(0); // also, try passing 1 and 2 to Xhandler()
    }
    catch(int i) {
        cout << "Caught an integer\n";
    }
    catch(char c) {
        cout << "Caught char\n";
    }
    catch(double d) {
        cout << "Caught double\n";
    }
    cout << "end";
    return 0;
}
```

10a) class Template

A **class template** provides a specification for generating **classes** based on parameters. **Class templates** are generally used to implement containers. A **class template** is instantiated by passing a given set of types to it as **template** arguments.

```
#include <iostream>
using namespace std;

// One function works for all data types. This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}
```

```

}

int main()
{
    cout << myMax<int>(3, 7) << endl; // Call myMax for int
    cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
    cout << myMax<char>('g', 'e') << endl; // call myMax for char

    return 0;
}

```

10b) containers categories

At the core of the standard template library are three foundational items: *containers*, *algorithms*, and *iterators*. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers

Containers are objects that hold other objects, and there are several different types. For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called *sequence containers* because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines *associative containers*, which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key. Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

Algorithms act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a *range* of elements within a container.

Iterators

Iterators are objects that act, more or less, like pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. There are five types of iterators:

Iterator	Access	Allowed
Random Access	Store and retrieve values.	Elements may be accessed randomly.
Bidirectional	Store and retrieve values.	Forward and backward moving.
Forward	Store and retrieve values.	Forward moving only.
Input	Retrieve, but not store values.	Forward moving only.
Output	Store, but not retrieve values.	Forward moving only.

Container Classes:

Vectors

Perhaps the most general-purpose of the containers is **vector**. The **vector** class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be adjusted at run time to accommodate changing program conditions. A vector solves

this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements. The template specification for **vector** is shown here:

```
template <class T, class Allocator = allocator<T> > class vector
```

Some of the most commonly used member functions are **size()**, **begin()**, **end()**, **push_back()**, **insert()**, and **erase()**. The **size()** function returns the current size of the vector. This function is quite useful because it allows you to determine the size of a vector at run time. Remember, vectors will increase in size as needed, so the size of a vector must be determined during execution, not during compilation. The **begin()** function returns an iterator to the start of the vector. The **end()** function returns an iterator to the end of the vector. As explained, iterators are similar to pointers, and it is through the use of the **begin()** and **end()** functions that you obtain an iterator to the beginning and end of a vector. The **push_back()** function puts a value onto the end of the vector. If necessary, the vector is increased in length to accommodate the new element. You can also add elements to the middle using **insert()**. A vector can also be initialized. In any event, once a vector contains elements, you can use array subscripting to access or modify those elements. You can remove elements from a vector using **erase()**

List

The list class supports a bidirectional, linear list. Unlike a vector, which supports random access, a list can be accessed sequentially only. Since lists are bidirectional, they may be accessed front to back or back to front.

A list has this template specification:

```
template <class T, class Allocator = allocator<T> > class list
```

Here, T is the type of data stored in the list. The allocator is specified by Allocator, which defaults to the standard allocator. It has the following constructors:

```
explicit list(const Allocator &a = Allocator());
```

```
explicit list(size_type num, const T &val = T(),
```

```
const Allocator &a = Allocator());
```

```
list(const list<T, Allocator> &ob);
```

```
template <class InIter>list(InIter start, InIter end, const Allocator &a = Allocator());
```

The first form constructs an empty list. The second form constructs a list that has num elements with the value val, which can be allowed to default. The third form constructs a list that contains the same elements as ob. The fourth form constructs a list that contains the elements in the range specified by the iterators start and end.

Maps

The map class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus, in its most general sense, a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone number as its value. Associative containers are becoming more popular in programming.

As mentioned, a map can hold only unique keys. Duplicate keys are not allowed. To create a map that allows nonunique keys, use multimap. The map container has the following template specification:

```
template <class Key, class T, class Comp = less<Key>, class Allocator = allocator<pair<const key, T>> > class map
```

10c) Vector Class

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections.

- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
- It extends **AbstractList** and implements **List** interfaces.

int capacityIncrement: Contains the increment value.

int elementCount: Number of elements currently in vector stored in it.

Object elementData[]: Array that holds the vector is stored in it.

```
#include <iostream>
```

```
#include <vector>
using namespace std;
```

```
int main() {
    // create a vector to store int
    vector<int> vec;
    int i;

    // display the original size of vec
    cout << "vector size = " << vec.size() << endl;

    // push 5 values into the vector
    for(i = 0; i < 5; i++){
        vec.push_back(i);
    }

    // display extended size of vec
    cout << "extended vector size = " << vec.size() << endl;

    // access 5 values from the vector
    for(i = 0; i < 5; i++){
        cout << "value of vec [" << i << "] = " << vec[i] << endl;
    }
    return 0;
}
```