**1.What are the advantage and disadvantages of object oriented programming over procedural programming? Describe the characteristics of OOP.**

Ans: The advantage of object oriented programming over procedural programming are as listed below:

 It is easy to model a real system as real objects are represented by programming objects in OOP. The objects are processed by their member data and functions. It is easy to analyze the user requirements.

 With the help of inheritance, we can reuse the existing class to derive a new class such that the redundant code is eliminated and the use of existing class is extended. This saves time and cost of program.

 In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that cannot be invaded by code in other part of the program.

 With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.

 Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.

 It is possible to have multiple instances of an object to co-exist without any interference i.e. each object has its own separate member data and function.

The main characteristics of OOP are described below:

**1. Objects**

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle. The fundamental idea behind object oriented approach is to combine both data and function into a single unit and these units are called objects. The term objects means a combination of data and program that represent some real word entity. When a program executed, the object interact by sending messages to one another. Each object contain data, and code to manipulate the data.

**2. Class**

A group of objects that share common properties for data part and some program part are collectively called as class. In C ++ a class is a new data type that contains member variables and member functions that operate on the variables.The entire set of data and code of an object can be made a user-defined data type with the help of a class. Objects are variable of the type class. Once a class has been defined, we can create any number of objects belonging to that class.When class is defined, objects are created as:

class student {

char name[30];

int reg-no;

int marks[7];

tot\_marks();

percentage\_marks();

}

student ram;

This will creat object ram belonging to the class student.

**3. Encapsulation**

Wrapping of data and functions together as a single unit is known as encapsulation. By default data is not accessible to outside world and they are only accessible through the functions which are wrapped in a class. Prevention of data from direct access by the program is called data hiding or information hiding.

**4. Data Abstraction**

Abstraction refers to the act of representing essential features without including the back ground details or explanation. Classes use the concept of abstraction and are defined as a list of attributes such as size, weight, cost and functions to operate on these attributes. They encapsulate all essential properties of the object that are to be created. The attributes are called as data members as they hold data and the functions which operate on these data are called as member functions.

**5. Inheritance**

Inheritance is the mechanism by which one class can inherit the properties of another. It allows a hierarchy of classes to be build, moving from the most general to the most specific. When one class is inherited by another, the class that is inherited is called the base class. The inheriting class is called the derived class. In general, the process of inheritance begins with the definition of a base class. The base class defines all qualities that will be common to any derived class. . In OOPs, the concept of inheritance provides the idea of reusability. In essence, the base class represent the most general description of a set of traits. The derived class inherits those general traits and adds properties that are specific to that class.

**6. Polymorphism**

Polymorphism comes from the Greek words “poly” and “morphism”. “poly” means many and “morphism” means form i.e.. many forms. Polymorphism means the ability to take more than one form. For example, an operation have different behavior in different instances. The behavior depends upon the type of the data used in the operation.

7. **Data Hiding**

The data and function in OOP are always together within an object that provides data hiding. Thus the insulation of data from direct access by the program is called data hiding or information hiding.

**Qns.2: Explain how the use of defult argument supports the function overloading with suitable example. Define namespace with its significance**.

Ans:

In C++, there is a provision of supplying less number of arguments than the actual number of parameters. This mechanism is supported by the default argument. If we do not supply any argument the default value is used for the argument that is absent in the function call. The default value are specified when function is declared. The default value is specified similar to the variable initialization. The default values can be constants, global variables, or even a function call. All of the default arguments must be in the right part of parameter list. Suppose a function has more than one default arguments. In a function call, if a value of a default argument is not specified, then the arguments from right are omitted.

Example:

//an example of default argument

//imagine 40 is given if absent in exam.

#include<iostream>

**using** **namespace** std;

**void** **marks\_tot** (**int** ml=**40**, **int** m2=**40**, **int** m3=**40** );

**int** **main**()

}

marks\_tot ();

marks\_tot (**55**);

marks\_tot (**66**,**77**);

marks\_tot (**75**, **85**, **92**);

**return** **0**;

**void** marks\_tot (**int** ml, **int** m2, **int** m3)

(

cout<<"Total marks: "<< (m1+m2+m3) <<endl;

}

Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names. Namespaces provide the space where we can define or declare identifiers i.e. names of variables, methods, classes, etc.

Significance:

* Namespace is a component included C++ and not present in C.
* A namespace is a revelatory district that gives a degree to the identifiers (names of the kinds, work, factors and so forth) inside it.
* Various namespace blocks with a similar name are permitted. All presentations inside those squares are proclaimed in the named scope.
* Namespaces in C++ are utilized to coordinate such a large number of classes with the goal that it tends to be not difficult to deal with the application.
* For getting to the class of a namespace, we need to utilize namespacename::classname. We can utilize catchphrases with the goal that we don’t need to utilize total names constantly.
* In C++, worldwide namespace is the root namespace. The global::std will consistently allude to the namespace “sexually transmitted disease” of C++ Framework.

**Qns3: Explain the relation between constant object and constant function with example. When do we use static data member and static function in a class? Exemplify**.

### Constant Objects

An object of a class may be declared to be const, just like any other C++ variable. For example:

const Date birthday(7, 3, 1969);

declares a constant object of the Date class named birthday.

The const property of an object goes into effect after the constructor finishes executing and ends before the class's destructor executes. So the constructor and destructor can modify the data members of the object, but other methods of the class can't.

Only constant member functions can be called for a constant object:

birthday.set\_month(12); // Syntax error.

### Constant Member Functions

You can declare a member function of a class to be const. This must be done both in the function's prototype and in its definition by coding the keyword const after the method's parameter list. For example:

class Date

{

private:

int month;

int day;

int year;

public:

Date();

Date(int, int, int);

void set\_month(int);

int get\_month() const; // This function does not change modify data members of the object that calls it.

...

};

int Date::get\_month() const

{

return month;

}

The this pointer passed to a const member function is a pointer to a const object. That means the pointer can not be used to modify the object's data members. Any attempt to change a data member of the object that called a constant method will result in a syntax error, as will attempting to call a non-const member function for that object.

Constant objects can **only** call constant member functions. Non-constant objects can call both constant and non-constant member functions.

A constant member function can be overloaded with a non-constant version. The choice of which version to use is made by the compiler based on whether or not the object used to call the member function is constant.

Constructors and destructors can **never** be declared to be const. They are always allowed to modify an object even if the object itself is constant.

A static variable is normally used to maintain value common to the entire class. For e.g, to hold the count of objects created. Note that the type and scope of each static member variable must be declared outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object.

Example:

#include <iostream>

**using** **namespace** std;

**class** **Demo**

{

**public:**

**static** **int** ABC;

};

//defining

**int** Demo :: ABC =**10**;

**int** **main**()

{

cout<<"**\n**Value of ABC: "<<Demo::ABC;

**return** **0**;

}

Output

Value of ABC: **10**

# Static Member Function

like a static member variable, we can also have static member functions. A member function that is declared static has the following properties:-

* A static function can have access to only other static members (function or variable) declared in the same class.
* A static member function can be called using the class name (instead of its object) as follows-

*Class\_name::Function\_name();*

Example:

#include <iostream>

**using** **namespace** std;

**class** **Demo**

{

**private:**

**static** **int** X;

**public:**

**static** **void** **fun**()

{

cout <<"Value of X: " << X << endl;

}

};

//defining

**int** Demo :: X =**10**;

**int** **main**()

{

Demo X;

X.fun();

**return** **0**;

}

Output

Value of X: **10**

**Qns4: How do you convert user-defined data type to basic data type? Write a program to overload the relational operators to compare the length (in meter and centimeter) of two objects.**

Ans: To convert a user-defined type to basic type, we can define an overloaded casting operator. The operator function must having following characteristics:

i) It must be a member of a class.

ii) It must not specify a return type.

iii) It must not have any arguments.

Syntax:

Operator type\_name()

{

//body…

}

Example:

#include<iostream.h>

**using** **namespace** std;

**class** **Distance**

{

**int** feet;

**float** inch;

**public:**

Distance()

{

feet=**0**;

inch=**0.0**;

}

Distance(**int** ft,**float** in)

{

feet=ft;

inch=in;

}

**operator** **float**()

{

**float** f= inch/**2**;

f=f+feet;

**return** (f/MTF);

}

};

**void** **main**()

{

Distance dist1(**3**,**3.37**);

**float** m;

m=dist1;

cout<<endl<<"The value of distance dist1 in meter:"<<m;

}

Comparing program:

#include<iostream>

**using** **namespace** std;

**class** **Distance**{

**private:**

**int** m,cm;

**public:**

Distance(){

m = **0**;

cm = **0**;

}

Distance(**int** a, **int** b){

m = a;

cm = b;

}

**friend** **int** **operator** < (Distance d1, Distance d2);

**friend** **int** **operator** > (Distance d3, Distance d4);

};

**int** **operator** <(Distance d1, Distance d2){

**int** t1,t2;

t1 = d1.m\***100** + d1.cm;

t2 = d2.m\***100** + d2.cm;

**if**(t1<t2){

**return** **1**;

}**else**{

**return** **0**;

}

}

**int** **operator** >(Distance d3, Distance d4){

**int** t1,t2;

t1 = d3.m\***100** + d3.cm;

t2 = d4.m\***100** + d4.cm;

**if**(t1>t2){

**return** **1**;

}**else**{

**return** **0**;

}

}

**int** main(){

Distance d1(**2**,**3**),d2(**4**,**5**);

cout << (d1>d2) << endl;;

cout << (d1<d2) ;

}

**Qns5: How the function over-riding differ from function overloading? Explain. Write a program to show the order of constructor invocation in multilevel inheritance.**

**Ans:**

### Difference between Function Overloading and Function Overriding in C++

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Function Overloading** | **Function Overriding** |
| 1. | In function overloading, two or more functions can own the same name, but the parameters will be different. | Function overriding permit us to redefine a method with the same name and signature |
| 2. | There is no requirement of the inheritance concept here. | In function overriding, we need an inheritance concept. |
| 3. | In the case of function overloading, the signatures should be different. | In the case of function overriding, the signatures should be the same. |
| 4. | We can use it as an example of compile time polymorphism. | We can use it as an example of run time polymorphism. |
| 5. | It happens during compile time. | It occurs during the run time. |
| 6. | A function has the ability to load multiple times. | A function can be overridden only a single time. |

Program:

// C++ program to implement

// constructor in multilevel

// Inheritance

#include<iostream>

**using** **namespace** std;

// Base class

**class** **A**

{

**public:**

A()

{

cout << "Base class A constructor **\n**";

}

};

// Derived class B

**class** **B**: **public** A

{

**public:**

B()

{

cout << "Class B constructor **\n**";

}

};

// Derived class C

**class** **C**: **public** B

{

**public:**

C()

{

cout << "Class C constructor **\n**";

}

};

// Driver code

**int** **main**()

{

C obj;

**return** **0**;

}

OUTPUT:

Base class A constructor

Class B constructor

Class C constructor

**Qns6: Explain abstract class with example. Explain how dynamic cast and typeid operators are used to achieve RTTI.**

Ans: By definition, an **abstract class in C++** is a class that has at least one pure virtual function (i.e., a function that has no definition). The classes inheriting the abstract class must provide a definition for the pure virtual function; otherwise, the subclass would become an abstract class itself.

Example:

#include <iostream>

**using** **namespace** std;

**class** **Shape** {

**public:**

**virtual** **int** Area() = **0**; // Pure virtual function is declared as follows.

// Function to set width.

**void** **setWidth**(**int** w) {

width = w;

}

// Function to set height.

**void** **setHeight**(**int** h) {

height = h;

}

**protected:**

**int** width;

**int** height;

};

// A rectangle is a shape; it inherits shape.

**class** **Rectangle**: **public** Shape {

**public:**

// The implementation for Area is specific to a rectangle.

**int** Area() {

**return** (width \* height);

}

};

// A triangle is a shape too; it inherits shape.

**class** **Triangle**: **public** Shape {

**public:**

// Triangle uses the same Area function but implements it to

// return the area of a triangle.

**int** Area() {

**return** (width \* height)/**2**;

}

};

**int** **main**() {

Rectangle R;

Triangle T;

R.setWidth(**5**);

R.setHeight(**10**);

T.setWidth(**20**);

T.setHeight(**8**);

cout << "The area of the rectangle is: " << R.Area() << endl;

cout << "The area of the triangle is: " << T.Area() << endl;

}

The syntax for dynamic\_cast operator is given below:

dynamic\_cast<target-type>(expr);

The target-type specifies the target type into which the cast converts the type to and expr is the expression which is being cast into the new type. So, the operator performs a runtime cast along with verifying the validity of the cast. When a pointer to object of a polymorphic class (class with virtual function) holds the address of the object of the derived class of the polymorphic base class then dynamic\_cast changes the base class pointer to derived class pointer.

The syntax for typeid operator is given below:

typeid(expr); or typeid(type\_name);

The typeid operator is like a function as that of sizeof operator. When an expression is passed as its operand, typeid returns a reference to a type\_info class type that represents the type of the object denoted by the expression. When a type name is passed as its operand, typeid returns a reference to a ‘type \_info’ class that represents the type\_name. If value of the pointer is null(0), then it throws a bad\_typeid exception. If value of pointer is not null then typeid returns the exact type of object during runtime.

**Qns7. What are different ios function used in stream I/O? How they are different from manipulators? Write a program to store and retrieve the information of patient ( patient\_ID, name, address,age and type) in hospital management system.**

#include<iostream>

#include<fstream>

#include<stdlib.h>

**using** **namespace** std;

**class** **hospital**

{

**char** name[**20**];

**int** id;

**char** addr[**20**];

**int** age;

**char** type[**20**];

**public:**

**void** **input**()

{

cout<<"**\n**Enter the patient id:";

cin>>id;

cout<<"**\n**Enter Patient name:";

cin>>name;

//cout<<"\nAddress:";

//cin>>addr;

//cout<<"\nAge:";

//cin>>age;

//cout<<"\nEnter type of visiting department:";

//cin>>type;

}

**void** **display**()

{

cout<<"**\n**Patient ID:"<<id;

cout<<"**\n**Patient Name:"<<name;

//cout<<"\nAddress:"<<addr;

//cout<<"\nAge:"<<age;

//cout<<"\nType:"<<type;

}

**void** **add**()

{

fstream fout;

hospital s;

fout.open("hospital.txt",ios::app | ios::out | ios::binary);

cout<<"**\n**The Hospital record:";

s.input();

fout.write((**char** \*)&s,**sizeof**(s));

fout.close();

}

**void** **displayall**()

{

fstream fin;

hospital s;

fin.open("hospital.txt", ios::in | ios::binary);

fin.seekg(**0**);

fin.read((**char** \*)&s,**sizeof**(s));

**while**(!fin.eof())

{

s.display();

fin.read((**char** \*)&s,**sizeof**(s));

}

fin.close();

}

};

/\*int main()

{

cout<<"Enter the detail of the patient:"<<endl;

hospital s1;

s1.add();

s1.displayall();

return 0;

}

\*/

//in case of 10 patients/employee/students

**int** **main**()

{

cout<<"Enter the detail of the patient:"<<endl;

hospital s1[**5**];

**for**(**int** i=**0**;i<**5**;i++)

{

s1[i].add();

}

s1[**0**].displayall();

**return** **0**;

}

**Qns8. How do you use class template with multiple template type? How the exception is re-thrown during exception handling?**

**Ans:**

While creating templates, it is possible to specify more than one type. We can use more than one generic data type in a class template. They are declared as a comma-separated list within the template as below:

Syntax:

template<class T1, class T2, ...>

class classname

{

...

...

};

Example:

#include<iostream>

**using** **namespace** std;

// Class template with two parameters

**template**<**class** **T1**, **class** **T2**>

**class** **Test**

{

T1 a;

T2 b;

**public:**

Test(T1 x, T2 y)

{

a = x;

b = y;

}

**void** show()

{

cout << a << " and " << b << endl;

}

};

// Main Function

**int** **main**()

{

// instantiation with float and int type

Test <**float**, **int**> test1 (**1.23**, **123**);

// instantiation with float and char type

Test <**int**, **char**> test2 (**100**, 'W');

test1.show();

test2.show();

**return** **0**;

}

Rethrowing an expression from within an exception handler can be done by calling throw, by itself, with no exception. This causes current exception to be passed on to an outer try/catch sequence. An exception can only be rethrown from within a catch block. When an exception is rethrown, it is propagated outward to the next catch block.

Example:

#include <iostream>

**using** **namespace** std;

**void** **MyHandler**()

{

try

{

**throw** “hello”;

}

**catch** (**const** **char**\*)

{

cout <<”Caught exception inside MyHandler\n”;

**throw**; //rethrow char\* out of function

}

}

**int** **main**()

{

cout<< “Main start”;

try

{

MyHandler();

}

**catch**(**const** **char**\*)

{

cout <<”Caught exception inside Main\n”;

}

cout << “Main end”;

**return** **0**;

}

Output:

Main start  
Caught exception inside MyHandler  
Caught exception inside Main  
Main end