2074 Chaitra (Regular)

1. ***What are the advantages of object oriented programming over procedural programming language? Explain the features of object oriented programming. Write a simple program that illustrates the object oriented concept.***

Ans:

The advantages of object oriented programming over procedural programming language are:

* + Incase of OOP, focus is given to data rather than procedures.
  + Incase of OOP, programs are divided into classes that specify data and their operations making it easier to navigate through the programs.

The features of OOP are:

* Class:

Classes are templates or models that specify data and their operations. After defining a class, any number of objects of its type can be created.

Example:

class Student

{

char name;

int reg\_no;

int marks[10];

};

* Objects:

Objects are defined as the physical instance of class. An object can have properties and behaviors. When the object is created, space for the object is allocated in primary memory. Objects are variables of type class.

* Encapsulation:

The wrapping up of data and functions in a single unit is called encapsulation. Data is not accessible to the outside world and only the functions which are wrapped in the class can access it.

A simple program illustrating the object oriented concept is as follows:

#include <iostream>

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

**class** **Demo**

{

**private:**

**int** n;

**public:**

**void** **showdata**(**int** num)

{

n= num;

cout<<"The entered integer is:"<<n;

}

};

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

**int** num;

Demo d;

cout<<"Enter an integer:"<<endl;

cin>>num;

d.showdata(num);

**return** **1**;

}

***2. Why do we need friend function? Explain how any member function of a class can be friend of other class with a suitable example.***

Ans:

Private members of a class cannot be accessed from outside the class i.e. a non-member function cannot have access to the private data of a class. Thus, a friend function acts as a bridge by allowing non-member function to access the private data of the class and operate on it.

Any member function of a class can be a friend of another class. This is called bridging classes with friend function. A program to illustrate about bridging classes with friend function is given below :-

#include<iostream>

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

**class** **second**;

**class** **first**{

**private:**

**int** data1;

**public:**

**void** **getdata1**(**int** x){

data1=x;

}

**friend** **int** **sum**(first,second);

};

**class** **second**{

**private:**

**int** data2;

**public:**

**void** **getdata2**(**int** y){

data2=y;

}

**friend** **int** **sum**(first,second);

};

**int** **sum**(first a, second b){

**return**(a.data1+b.data2);

}

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

first a;

a.getdata1(**100**);

second b;

b.getdata2(**200**);

cout<<"sum of first and second is:"<<sum(a,b);

**return** **0**;

}

***3. Explain the features of C++. What is namespace? Explain how memory is allocated and deleted dynamically for normal variable and for array in C++ with example program.***

Ans:

The features of C++ are (any two) :

* Function overloading:  
   Some functions conceptually perform the same task on objects of different types and numbers. In such cases, it is convenient to give them the same name. When the same name is used for different operations, it is called function overloading. When an overloaded function is called, the function with matching arguments is invoked.
* Friend function:  
   It is a function which allows non member functions to access the private data of a class. Thus, it acts as a bridge between two classes by operating on their private datas.

-Namespace:  
 The namespace mechanism is used for the logical grouping of variables, classes and functions in C++.   
Syntax :   
 namespace namespace\_name

{  
 //declaration of variables, classes, functions, etc.

}

-Dynamic Memory Allocation for normal variable:

#include <iostream>

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

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

// declare an int pointer

**int**\* pointInt;

// declare a float pointer

**float**\* pointFloat;

// dynamically allocate memory

pointInt = **new** **int**;

pointFloat = **new** **float**;

// assigning value to the memory

\*pointInt = **45**;

\*pointFloat = **45.45f**;

cout << \*pointInt << endl;

cout << \*pointFloat << endl;

// deallocate the memory

**delete** pointInt;

**delete** pointFloat;

**return** **0**;

}

Output:

45

45.45

In this program, we dynamically allocated memory to two variables of int and float types. After assigning values to them and printing them, we finally deallocate the memories using the code.

delete pointInt;

delete pointFloat;

-Dynamic memory allocation and deletion for arrays:

/\*C++ Program to store GPA of n number of students and display it

where n is the number of students entered by the user\*/

#include <iostream>

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

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

**int** num;

cout << "Enter total number of students: ";

cin >> num;

**float**\* ptr;

// memory allocation of num number of floats

ptr = **new** **float**[num];

cout << "Enter GPA of students:” << endl;

**for** (**int** i = **0**; i < num; ++i) {

cout << "Student" << i + **1** << ": ";

cin >> \*(ptr + i);

}

cout << "**\n**Displaying GPA of students:” << endl;

**for** (**int** i = **0**; i < num; ++i) {

cout << "Student" << i + **1** << ": " << \*(ptr + i) << endl;

}

// ptr memory is released

**delete**[ ] ptr;

**return** **0**;

}

Output:

Enter total number of students: 4

Enter GPA of students:

Student1: 3.6

Student2: 3.1

Student3: 3.9

Student4: 2.9

Displaying GPA of students:

Student1: 3.6

Student2: 3.1

Student3: 3.9

Student4: 2.9

In this program, we have asked the user to enter the number of students and store it in the num variable. Then, we have allocated the memory dynamically for the float array using new. We enter data into the array (and later print them) using pointer notation. After we no longer need the array, we deallocate the array memory using the code delete[ ] ptr;

***4. Explain why default arguments are used with functions. How can* *a* *function with default argument be implemented with function overloading?Explain with example.***

Ans:   
 A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn’t provide a value for the argument. In case any value is passed, the default value is overridden. Default arguments are used in functions in those programs where we do not want to change the values of the function but the function needs to be called multiple times in our programs. Using default arguments in such cases helps us save time and also reduces the size of our code.

Program that demonstrates the use of default argument during function overloading is given below:

#include<iostream>

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

**class** **number**{

**public:**

**void** displaynum(**int** n)

{

cout<<"Number:"<<n<<endl;

}

**void** displaynum(**float** n=**0.0**) //setting default argument

{   
 cout<<"Number:"<<n<<endl;

}

};

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

number n;

n.displaynum(**2**);

n.displaynum(**2.2f**);

n.displaynum();

**return** **0**;

}

Output:

Number: 2

Number: 2.2

Number: 0

***5. Define operator overloading. Write operator functions as member function of a class to overload arithmetic operator +, logical operator ‘<=‘ and stream operator ‘<<‘ to operator on the objects of user defined type time (hr, min, sec).***

Ans:

Operator overloading is a feature of C++ in which we can extend the functionalities of the existing operator for user defined data types.  
Syntax:  
 return\_type operator operator\_symbol(arg\_list)  
 {  
 //function body  
 }

Program:

#include<iostream>

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

**class** **time**{

**private:**

**int** hr,min,sec;

**public:**

time(){

hr=**0**;

min=**0**;

sec=**0**;

}

time(**int** a, **int** b, **int** c){

hr=a;

min=b;

sec=c;

}

time **operator**+(time t1){ //overloading + operator

time temp;

temp.hr=hr+t1.hr;

temp.min=min+t1.min;

temp.sec=sec+t1.sec;

**if** (temp.sec>**60**){

temp.sec=temp.sec-**60**;

temp.min++;

}

**if** (temp.min>**60**){

temp.min=temp.min-**60**;

temp.hr++;

}

**return** temp;

}

**void** display(){

cout<<""<<hr<<":"<<min<<":"<<sec<<"**\n**";

}

**friend** **int** **operator**<=(time t1,time t2);

**friend** ostream &**operator**<<(ostream &os, time&tm);

};

**int** **operator**<=(time t1, time t2){ //overloading<= operator

**if**(t1.hr<t2.hr||t1.hr==t2.hr){

**return** **1**;

}

**else** {

**return** **0**;

}

}

ostream &**operator**<<(ostream &os, time&tm){ //overloading<< operator

os<<tm.hr<<":"<<tm.min<<":"<<tm.sec<<flush;

**return** os;

}

**int** main(){

time t1(**3**,**45**,**52**),t2(**5**,**10**,**43**),t3;

t3=t1+t2; //overloading + operator

t3.display();

**if**(t1<=t2){ // overloading <= operator

cout<<"time 1 is smaller"<<"**\n**";

}

**else** {

cout<<"time 2 is smaller"<<"**\n**";

}

cout<<"the time is:::"<<t3; //for overloading<< operator

**return** **0**;

}

***6. What is ambiguity and function overriding? How can they be resolved? Explain each with a suitable example.***Ans**:** Ambiguity:  
 In multiple inheritance, when one class is derived from two or more base classes then there may be a possibility that the base classes have functions with the same name. This condition is known as ambiguity.   
- It can be resolved by:

* Applying scope resolution operator.
* Using virtual base class.  
  Example:

#include <iostream>

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

**class** **Base1**

{

**public:**

**void** display()

{

cout<<"Base one"<<endl;

}

};

**class** **Base2**

{

**public:**

**void** display()

{

cout<<"Base two"<<endl;

}

};

**class** **derived**: **public** Base1, **public** Base2

{

**public:**

**void** disp()

{

Base1::display();

Base2::display();

}

};

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

{

derived d;

d.Base1::display();

d.Base2::display();

return 0;

}

Function overriding:   
 The process of creating members in derived class with the same name as that of visible members of base class is called function overriding.  
- It can be resolved by:

* Applying scope resolution operator.
* Using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.  
  Example:

#include <iostream>

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

**class** **base**

{

**protected:**

**int** num;

**public:**

**void** **readdata**()

{

cout<<"Enter number in base class:";

cin>>num;

}

**void** **showdata**()

{

cout<<"Number in base class="<<num<<endl;

}

};

**class** **derived**: **public** base

{

**protected:**

**int** num;

**public:**

**void** **readdata**()

{

base::readdata();

cout<<"Enter number in derived class:";

cin>>num;

}

**void** **showdata**()

{

base::showdata();

cout<<"Number in derived class="<<num<<endl;

}

};

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

derived d;

d.readdata();

d.showdata();

**return** **0**;

}

***7. What is pure virtual function and abstract class? With suitable example, explain run time polymorphism.***

Ans:  
 -Pure virtual function:  
 A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class.

Syntax:  
 class test {  
 virtual return\_type function\_name( )=0;  
 };

-Abstract class:  
 Class that contains at least one pure virtual function is known as abstract class. However, pointer to abstract class can be created.

In the above syntax, class ‘test’ is an abstract class.

-Run-time polymorphism:  
When the information needed to call a function is known during program execution, it is known as run time polymorphism. It allows the postponement of the decision of selecting a particular function until run-time which is achieved by virtual function.

An example of run time polymorphism is as follows:

#include<iostream>

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

**class** **Animal**

{

**public:**

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

{

cout<<"Base class Animal"<<endl;

}

};

**class** **Cat**: **public** Animal

{

**public:**

**void** display()

{

cout<<"Derived class Cat"<<endl;

}

};

**class** **Dog**: **public** Animal

{

**public:**

**void** display()

{

cout<<"Derived class Dog"<<endl;

}};

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

{

Animal \*pa;

Animal a;

Cat c;

Dog d;

pa= &a;

pa ->display();

pa= &c;

pa ->display();

pa= &d;

pa ->display();

**return** **1**;

}

In this program, the virtual void display( ) indicates that the member function display( ) is virtual and binding of a call to this function must be postponed until runtime.  
  
***8. Discuss about stream class hierarchy. How a file can be open in C++? Explain with suitable example and syntax. Write a program to write the information of 10 employee in a file. And also display their details in console.***

Ans:

The stream class defines objects which accepts a sequence of characters.   
The stream class heirarchy includes:

* ofstream: It is used to display output to file (to write data to file)
* ifstream: It is used while scanning data from file.
* fstream: It helps to scan data from file and also help to write data to file.
* filebuf: It is used to set the file buffer to read and write.

A file can be opened in C++ in two ways:

* By using constructor :  
  Example:  
  ofstream outf(“student.txt”);   
  Here, “out” object is initialized with file named “student.txt” for writing purpose only.  
  Syntax:  
  file\_stream\_class file\_stream\_object(“file name”);
* By using open( ) function:  
  Example:  
  ofstream outf; //outf is a object of “ ofstream class”.  
  outf.open(“student.txt”); //opening a file using open function.

Syntax:  
 file\_stream\_class file\_stream\_object;  
 file\_stream\_object.open(“file name”);

Program to write the information of 10 employees in a file and displaying their details in console:

#include<iostream>

#include<fstream>

#include<stdlib.h>

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

**class** **employee**

{

**private:**

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

**int** age;

**float** salary;

**public:**

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

{

cout<<"Enter name: "<<endl;

cin>>name;

cout<<"Enter age: "<<endl;

cin>>age;

cout<<"Enter salary: "<<endl;

cin>>salary;

}

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

cout<<"Employee name:"<<name<<endl;

cout<<"Employee age:"<<age<<endl;

cout<<"Employee salary:"<<salary<<endl;

}

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

fstream fout;

employee e;

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

cout<<"The employee record:"<<endl;

e.input();

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

fout.close();

}

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

{

fstream fin;

employee e;

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

fin.seekg(**0**);

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

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

{

e.display();

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

}

fin.close();

}

};

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

{

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

employee emp[**10**];

**for**(**int** i=**0**;i<**10**;i++)

{

emp[i].add();

}

cout<<"The details of the employees:"<<endl;

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

**return** **0**;

}

***9. Explain why do we need template. Explain the function template overloading with suitable example.***

Ans*:*

Templates in c++ can be defined as a blueprint or formula for creating a generic class or a function formula. Templates are necessary in oop as the concept of reusability is implemented with the help of templates. Using templates, we can create a single function or single class to work with different data types using templates. It is used for generic programming. generic programming is an approach of programming where generic types are used as parameters in algorithms to work for a variety of data types.

-Function Template Overloading:

If the function templates definition is not suitable for some specific data types, then it is necessary to override the function template by defining normal function for specific types.

If a normal function is an exact match that the function template, then the normal function is selected when the function is called.

Program to illustrate overloading function template with functions:

#include<iostream>

#include<cstring>

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

**template** <**class** **T**>

T findmax(T &a, T &b){

T result;

result=(a>b)?a:b;

**return** result;

}

**char** \*findmax(**char** \*a, **char** \*b){

**char** \*result;

**if**(strcmp(a,b)>**0**){

result=a;

}**else**{

result=b;

}

**return** result;

}

**int** main(){

**int** i1=**5**, i2=**10**;

cout<<"Greater is: "<<findmax(i1,i2)<<endl;

**float** c=**6.3**,d=**4.3**;

cout<<"Greater is: "<<findmax(c,d)<<endl;

**double** d1=**55.5**,d2=**67.777**;

cout<<"Greater is: "<<findmax(d1,d2)<<endl;

**char** c1='a', c2='g';

cout<<"Greater is: "<<findmax(c1,c2)<<endl;

**char** str1[**20**]="kathmandu", str2[**20**]="pokhara";

cout<<"Greater is: "<<findmax(str1,str2)<<endl;

**return** **0**;

}

***10. Explain about all Exception Handling constructs. With suitable example, explain multiple exceptions handling in C++.***

Ans:

Exception Handling Constructs are of three types i.e:

1.try:

The part of the code that can generate exception or call function that generates exception should be places within 'try' block.

2.catch:

The part of the code to handle appropriate exceptions should be placed within the 'catch' block.

3.throw:

When a program experiences an issue, it throws an exception. The throw keyword assists the program by performing throw.

Multiple Exception Handling in c++:

#include <iostream>

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

**void** **test**(**int** x){

try {

**if**(x==**1**){

**throw** x; //int

}

**else** **if**(x==**0**){

**throw** 'x'; //char

}

**else** **if**(x==-**1**){

**throw** **1.0**; //double

}

cout<<"end of try block";

}

**catch**(**char** c){

cout<<"caught an character"<<"**\n**";

}

**catch**(**int** m){

cout<<"caught an integer"<<"**\n**";

}

**catch**(**double** d){

cout<<"caught a double"<<"**\n**";

}

}

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

cout<<"testing multiple catches"<<"**\n**";

cout<<"x==1"<<"**\n**";

test(**1**);

cout<<"x==0"<<"**\n**";

test(**0**);

cout<<"x==-1"<<"**\n**";

test(-**1**);

cout<<"x==2"<<"**\n**";

test (**2**);

**return** **0**;

}

-THE END-