**2072 CHAITRA**

1. **Explain main characteristics of Object-Oriented Programming. Write a program to find the transpose of given Matrix using the concept of Object-Oriented Programming.**

→ The main characteristics of Object-Oriented Programming is as follows: -

1. It emphasizes on data rather than procedures.
2. In Object-Oriented Programming data can be hidden to prevent from accident modification.
3. It follows bottom-up approach in program design
4. It supports reusability of code as it supports inheritance, polymorphism, data abstraction and encapsulation.
5. Programs are divided into objects.
6. Objects may communicate with each other through function.

#include<iostream>

#include<conio.h>

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

**class** **matrix**

{

**int** mat[**20**][**20**],mat2[**20**][**20**],x,y,i,j;

**public:**

**void** **getdata**()

{

cout<<"Enter the number of rows ";

cin>>x;

cout<<"Enter the number of columns ";

cin>>y;

}

**void** **getdata\_2**()

{

cout<<"Enter the elements ";

**for** (i=**0**;i<x;i++)

{

**for** (j=**0**;j<y;j++)

{

cin>>mat[i][j];

}

}

}

**void** **transpose**()

{

cout<<"The transpose of the given matrix is "<<endl;

**for** (i=**0**;i<x;i++)

{

**for** (j=**0**;j<y;j++)

{

mat2[i][j]=mat[j][i];

cout<<mat2[i][j]<<"**\t**";

}

cout<<endl;

}

}

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

{

cout<<"The given matrix is "<<endl;

**for** (i=**0**;i<x;i++)

{

**for** (j=**0**;j<y;j++)

{

cout<<mat[i][j]<<"**\t**";

}

cout<<endl;

}

}

};

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

{

matrix m;

m.getdata();

m.getdata\_2();

m.display();

m.transpose();

}

1. **Define constructor. Why constructor is needed for a class? Explain about different types of constructor with a suitable program.**

**→** A constructor is a special member function which is called automatically during the creation of an object. The constructor is identified as a function whose name is the same as its class.

Constructor is needed for a class because with the help of a constructor we can execute a certain set of instructions that are present in the constructor and don't need to be called time and again, hence saving time and making the program short and easy to understand.

Types of constructor: -

1. Default constructor: -

The constructor which has no argument is known as default constructor.

A default constructor is automatically called when no arguments are supplied while creating an object.

Syntax: -

Class\_name object\_name;

1. Parameterized constructor: -

The constructor which has some argument is known as

Parameterized constructor. In parameterized constructor, we must pass some arguments while defining the object of the class.

Syntax: -

Class\_name object\_name(argument list);

1. Copy constructor: -

The constructor that initializes data member of the object by copying

the value of another object initialized by either default or parameterized constructor is

called copy constructor.

Syntax: -

Class\_name (class\_name &object)

{

//body}

#include<iostream>

#include<conio.h>

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

**class** **add**

{

**int** x,y,z, sum;

**public:**

add() //Default constructor

{

x=**0**;

y=**0**;

z=**0**;

}

add (**int** a,**int** b,**int** c) //Parameterized constructor

{

x=a;

y=b;

z=c;

}

add(add &p) //copy constructor

{

x=p.x;

y=p.y;

z=p.z;

}

**void** calculate()

{

sum=x+y+z;

}

**void** display()

{

cout<<"The sum of three digits are "<<sum;

}

};

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

{

add a(**1**,**2**,**3**);

add b(a);

b.calculate();

b.display();

}

1. **Write down the significance of the reference variable with suitable example. Define default argument. Write a program to show the relation between default argument and function overloading.**

**→** When a variable is declared as a reference, it becomes an alternating name for the existing variable. It is declared using & operator.

Syntax: -

Data\_type &reference\_variable = Original\_variable;

#include<iostream>

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

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

{

**int** x=**100**;

**int** &ref=x;

cout<<"x="<<x<<"**\t** ref = "<<ref;

}

In above program ‘ref ‘ is used as a reference variable of x. To declare ref as a reference variable, &

Is used.

Default Argument: -

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. With the help of default argument, we can call a function without specifying all its arguments.

#include<iostream>

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

**void** **mark**(**int** m1=**10**, **int** m2=**20**,**int** m3=**30**);//default argument

**void** **sub**(**int**,**int**);//function overloading

**void** **sub**(**int**);//function overloading

**void** **sub**(**int**,**int**,**int**);//function overloading

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

{

mark();

mark(**55**);

mark(**22**,**44**);

mark(**75**,**80**,**29**);

sub(**20**,**3**);

sub(**75**,**80**,**29**);

sub(**2**);

**return** **0**;

}

**void** **mark**(**int** m1,**int** m2,**int** m3)

{

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

}

**void** **sub**(**int** m1,**int** m2)

{

cout<<"Difference is "<<(m1-m2)<<endl;

}

**void** **sub**(**int** m1)

{

cout<<"Just one number cause just one argument hence given value is "<<(m1)<<endl;

}

**void** **sub**(**int** m1,**int** m2,**int** m3)

{

cout<<"Difference is "<<(m1-m2-m3)<<endl;

}

Default argument and function overloading look similar when written in code but are a bit different from each other. In the default argument, if we pass less arguments then still the program will be executed as it will take the default argument provided by the user. But in function overloading if less argument is provided by the user, then a different argument is required to make as it doesn’t take any previously provided data by the user. However with the help of default argument we can achieve function overloading.

1. **Why do you need operator overloading? What are the non-overloadable operators in C++? Write a program that will convert a object from a class Rectangle to object of a class Polar using Casting Operator.**

**→** Operator overloading is needed because it provides a flexible option for the extension of the meaning of the operator when applied to user defined data type.

The non-overloadable operators are: -

1. Scope resolution operator (::)
2. Member access operator (.)
3. Pointer to member access operator (.\*)
4. Conditional operator (?:)

#include<iostream>

#include<conio.h>

#include<math.h>

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

**class** **polar**

{

**float** rad,ang;

**public:**

polar()

{

rad=**0**;

ang=**0**;

}

polar(**double** r, **double** a)

{

rad=r;

ang=a;

}

**void** display()

{

cout<<"("<<rad<<","<<ang<<")";

}

};

**class** **rectangular**

{

**float** x,y;

**public:**

rectangular ()

{

x=**0**;

y=**0**;

}

rectangular (**float** a, **float** b)

{

x=a;

y=b;

}

**void** display()

{

cout<<"("<<x<<","<<y<<")";

}

**operator** polar ()

{

**float** r,a;

r=sqrt((pow(x,**2**))+(pow(y,**2**)));

a=atan(y/x);

**return** **polar** (r,a);

}

};

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

{

polar p;

rectangular r(**2**,**4**);

p=r;

cout<<"Rectangular coordinate is "<<endl;

r.display();

cout<<"Polar coordinate is "<<endl;

p.display();

}

1. **Explain the need of virtual base class with suitable example. Create a derived class manager from two base classes person and employee. Assume suitable data members in each class and display the information.**

**→** Virtual base class is used to remove the ambiguity in multipath inheritance. When two or more objects are derived from a common base class, we can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as “virtual” when it is inherited.

#include<iostream>

#include<conio.h>

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

**class** **person**

{

**protected:**

**char** a[**20**],b[**30**];

**int** c;

**public:**

**void** **getdata**()

{

cout<<"Enter the employee name ";

cin>>a;

cout <<"Enter the age of the employee ";

cin>>c;

cout<<"Enter the address ";

cin>>b;

}

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

{

cout<<"The given data is "<<endl;

cout<<"Name: "<<a<<endl;

cout<<"Age: "<<c<<endl;

cout<<"Address: "<<b<<endl;

}

};

**class** **employee**

{

**protected:**

**int** sal, exper;

**char** post[**20**];

**public:**

**void** **getdata**()

{

cout<<"Enter the Employee's salary ";

cin>>sal;

cout<<"Enter the person's post in the company ";

cin>>post;

cout<<"Enter the number of years the employee has experience ";

cin>>exper;

}

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

{

cout<<"Salary: "<<sal<<endl;

cout<<"Post: "<<post<<endl;

cout<<"Experience: "<<exper<<endl;

}

};

**class** **manager**:**public** person,**public** employee

{

**public:**

**void** getdata\_2()

{

person::getdata();

employee::getdata();

}

**void** display\_2()

{

person::display();

employee::display();

}

};

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

{

manager m;

m.getdata\_2();

m.display\_2();

}

1. **Explain about the stream class hierarchy by highlighting the different ios flags and their usage. Write a program to make billing system of a department store. Your program should store and retrieve data to/from files. Use manipulators to display the record in proper formats.**

**→**

**ios\_base**



Pointer

**ios**





**Stream\_buf**

**Istream**

**ostream**

**iostream**

**Fig: -Stream class hierarchy for console input/output**

Above figure is the block diagram of stream class hierarchy for the console input/output. All the stream classes are derived from the base class ios class. The ios class has an associated stream\_buf object that acts as the buffer for the stream. The classes istream and ostream are derived from ios stream to perform input and output tasks respectively. Iostream is derived from istream and ostream.

The different types of flags and their usage are as follows: -

1. width() : To specify required field size for displaying an output value
2. precision() : To specify the number of digit to be displayed after the decimal
3. fill() : to specify the character that will be filled in place of unused portion of field
4. setf() : to specify format flag that can control the form of output display

#include<iostream>

#include<conio.h>

#include<fstream>

#include<iomanip>

**const** **char**\* record="billing.dat";

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

**class** **bill**

{

**public** :

**char** item[**20**],ch;

**int** pri,qua,j=**0**,gt;

**public:**

**void** **read\_data**();

**void** **show\_data**();

**void** **write\_file** ();

**void** **read\_file**();

**void** **tot**();

**void** **remove**();

**void** **show\_all**();

**void** **calc**();

};

**void** bill::read\_data()

{

cout<<"**\n** Enter item name ";

cin>>item;

cout<<"Enter price ";

cin>>pri;

cout<<"Enter quantity ";

cin>>qua;

cout<<endl;

j=j+(pri\*qua);

}

**void** bill::show\_data()

{

cout.setf(ios::internal,ios::adjustfield);

cout<<item;

cout.setf(ios::internal,ios::adjustfield);

cout.width(**25**);

cout<<qua;

cout.width(**15**);

cout<<pri;

cout.width(**15**);

cout<<(qua\*pri)<<endl;

}

**void** bill::tot()

{

cout<<"The total amount is ";

cout.setf(ios::right,ios::adjustfield);

cout.width(**40**);

cout<<j<<endl;

cout<<"VAT";

cout.width(**45**);

cout<<"13%"<<endl;

cout<<"GRAND TOTAL";

cout.width(**40**);

cout<<(j+(**.13**\*j));

}

**void** bill::write\_file()

{

ofstream outfile(record,ios::binary|ios::app);

read\_data();

outfile.write(**reinterpret\_cast**<**char**\*>(**this**),**sizeof**(\***this**));

}

**void** bill::read\_file()

{

ifstream infile(record,ios::binary);

**if** (!infile)

{

cout<<"File not found ";

**return**;

}

**while**(!infile.eof())

{

**if** (infile.read(**reinterpret\_cast**<**char**\*>(**this**),**sizeof**(\***this**))>**0**)

show\_data();

}

}

**void** bill::remove()

{

ofstream outfile(record);

}

**void** bill::show\_all()

{

ifstream infile(record,ios::binary);

**if**(!infile)

{

cout<<"File not found";

**return**;

}

**while**(!infile.eof())

{

**if** (infile.read(**reinterpret\_cast**<**char**\*>(**this**),**sizeof**(\***this**))>**0**)

show\_data();

}

}

**int** main()

{

bill b;

**int** c;

cout<<"Enter Data"<<endl;

b.write\_file();

**while**(true)

{

cout<<"Do You want to enter more data? "<<endl;

cout<<"Press '1' for YES "<<endl;

cout<<"Press '2' for NO "<<endl;

cin>>c;

**switch**(c)

{

**case** **1**:

b.write\_file();

**break**;

**case** **2**:

cout.setf(ios::internal,ios::adjustfield);

cout.width(**35**);

cout.fill('\*');

cout<<endl<<endl<<"BILLING SYSTEM";

cout.width(**20**);

cout.fill('\*');

cout<<"\*"<<endl<<endl;

cout<<"ITEM ";

cout.width(**30**);

cout.fill(' ');

cout<<"QUANTITY";

cout.width(**15**);

cout<<"PRICE";

cout.width(**15**);

cout<<"TOTAL"<<endl<<endl;

b.show\_all();

b.remove();

b.tot();

exit(**0**);

**break**;

}

}}

1. **Why do you need virtual destructor? Explain with example. Write a program having polygon as an abstract class with length and height as its data member. Create derived class rectangle and triangle. Make area() as pure virtual function and redefined it in derived class to calculate respective area.**

**→** A virtual destructor is needed to free up the memory space allocated by the derived class object or while deleting the instances of the derived class using a base class pointer object.

#include<iostream>

#include<conio.h>

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

**class** **vehicle**

{

**public:**

vehicle ()

{

cout <<"constructor of vehicle invoked"<<endl;

}

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

{

cout<<"vehicle"<<endl;

}

**virtual** ~vehicle()

{

cout<<"Destructor of the vehicle"<<endl;

}

};

**class** **car**:**public** vehicle

{

**public:**

car()

{

cout<<"Constructor of car"<<endl;

}

**void** showdata()

{

cout<<"Car"<<endl;

}

~car()

{

cout<<"destructor of car ";

}

};

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

{

vehicle \*vp;

vp=**new** vehicle;

vp->showdata();

**delete** vp;

vp=**new** car;

vp->showdata();

**delete** vp;

**return** **0**;

}

#include<iostream>

#include<conio.h>

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

**class** **polygon**

{

**protected:**

**int** length, height;

**public:**

**virtual** **void** area()=**0**;

};

**class** **rectangle**:**public** polygon

{

**protected:**

**int** a;

**public:**

**void** **area**()

{

cout<<"Enter length";

cin>>length;

cout<<"Enter height";

cin>>height;

a=length\*height;

cout<<"The area of rectangle is "<<a<<endl;

}

};

**class** **triangle**:**public** polygon

{

**protected:**

**float** a;

**public:**

**void** **area**()

{

cout<<"Enter length ";

cin>>length;

cout<<"Enter height ";

cin>>height;

a=(length\*height)/**2**;

cout<<"The area of triangle is "<<a;

}

};

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

{

polygon \*ptr;

rectangle rectangle;

triangle triangle;

ptr=&rectangle;

ptr->area();

ptr=&triangle;

ptr->area();

}

1. **Define function template and class template with respective syntax. Write a program to find the square root of given number. Check the validity of input number and raise the exception as per requirement.**

**→**  Function template is a single function template that works with multiple data types. A function template can be used to operate on different types of data by taking data type as a parameter.

Syntax: -

Template<class type>

Return\_type dunction\_name(parameter\_list)

{

//body of a function

}

Class template is similar to function template. We can declare class template that operate on any type of data. A class that operates on any type of data is called class template.

Syntax: -

Template<class template\_type,…>

class class\_name

{

private:

//data member of template type or non template type

//……..

public:

//function member with template type argument

// and return type

};

#include<iostream>

#include<exception>

#include<math.h>

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

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

{

**float** a;

cout<<"Enter a number ";

cin>>a;

try

{

**if**(a>**0**)

{

cout<<"The square root of the given number is "<<sqrt(a);

}

**else**

{

**throw**(a);

}

}

**catch** (**float** i)

{

cout<<"Exception is caught as it is less than or equal to zero ";

}

**return** **0**;

}