**2073 Chaitra**

1. **What are the advantages and disadvantages of object oriented programming over procedural programming? Briefly describe the features of C++.**

**Answer:**  The advantages and disadvantages of oop over pop are:

Advantages:

* Redundant code is eliminated by various techniques like inheritance and templates.
* Through data hiding programmer can built secure programs.
* Existing classes can server as library class for further enhancements.
* Division of programs into objects makes software development easy.
* Code reusability is much easier.
* Message passing techniques makes communication easier among various objects.
* Upgrading and maintenance of software is easily manageable.
* Models real world system perfectly.
* Software complexity can be managed easily.
* Data can be hidden from outside world using encapsulation and data hiding.
* User can create/define new data type or user-defined type by making class.

Disadvantages:

* Compiler and runtime overhead is high because object oriented program requires more time during compilation.
* For dynamic and runtime support it requires more resource and processing time.
* The message passing between mainly objects of complex application may be difficult to trace and debug.
* Benefits only in long run while managing large software projects.
* Requires the mastery in software engineering and programming methodology.

Features of C++ are:

* Namespace
* Classes
* Derived classes
* Access controllers
* Constructors and destructors
* Friend function and classes
* Reference variables and classes
* Default arguments
* Function overloading
* Operator overloading
* Inline function
* Run time polymorphism
* Generic programming
* Exception Handling
* Template
* Runtime type information(RTTI)

1. **Define dynamic memory allocation. How do you use it in C++? Explain reference variable with suitable example. Write a program to swap to numbers using pass by reference concept.**

**Answer:** The process of allocating and de-allocating memory at run-time id known as Dynamic memory allocation.

C++ provides ‘new’ operator for dynamic memory allocation and ‘delete’ operator for dynamic memory de-allocation.

**Syntax**(for ‘new’ operator)**:**

data\_type \*data\_type\_ptr;

data\_type\_ptr = **new** data\_type; // allocates single variable memory

data\_type\_ptr = **new** data\_type[size]; //allocates an array

**Syntax**(for ‘delete’ operator)**:**

**delete** data\_type\_ptr; //release single dynamic variable

**delete** [] data\_type\_ptr; //release dynamically created array

A reference variable is another name (i.e. alias) for a variable. It is declared using ‘&’ operator.

**Syntax:**

datatype & reference\_variable=orignal\_value;

**Example:**

#include<iostream>

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

{

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

**int** &ref=x;

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

}

Swapping two numbers using pass by reference:

#include<iostream>

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

**void** **swap**(**int**&,**int**&);

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

{

**int** a=**5**,b=**6**;

cout<<"Before swapping: a="<<a<<"and b="<<b<<endl;

swap(a,b);

cout<<"After swapping: a="<<a<<"and b="<<b;

**return** **0**;

}

**void** **swap**(**int** &x, **int** &y)

{

**int** temp;

temp=x;

x=y;

y=temp;

}

1. **Define ‘this’ pointer with its applications. Explain the order in which constructor and destructors are invoked with suitable example.**

**Answer:** ‘this’ pointer is a pointer which represents an object that invokes a member function. ‘this’ is a pointer that points to the object for which this function was called.

**Applications:**

* It can be used to pass the current object as a parameter to another method.
* It can be used to refer current class instance variable.
* It can be used to declare indexers.
* It acts as an implicit argument to all the member function.
* It is automatically passed to a member function when it is called.

C++ constructor call order is from top to bottom that is from base class to derived class and destructor call is in reverse order that is from derived class to base class.

Example:

#include<iostream>

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

**int** count=**0**;

**class** **alpha**

{

**public:**

alpha()

{

count++;

cout<<"**\n** no of object created "<<count;

}

~alpha()

{

cout<<"**\n** no of object destroyed "<<count;

count--;

}

};

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

{

cout<<"**\n\n** enter main **\n**";

alpha a1,a2,a3,a4;

{

cout<<"**\n\n** enter block 1**\n**";

alpha a5;

}

{

cout<<"**\n\n** enter block 2**\n**";

alpha a6;

}

cout<<"**\n\n** re-center main**\n**";

**return** **0**;

}

/\* OUTPUT:

enter main

no of object created 1

no of object created 2

no of object created 3

no of object created 4

enter block 1

no of object created 5

no of object destroyed 5

enter block 2

no of object created 5

no of object destroyed 5

re-center main

no of object destroyed 4

no of object destroyed 3

no of object destroyed 2

no of object destroyed 1 \*/

1. **Define operator overloading. What are the rules of operator overloading? How do you overload unary operator? Explain in detail with example.**

**Answer:** Operator overloading is a feature of C++ in which we can extend the functionalities of the existing operator for the user defined data types.

Rules for operator overloading:

* Only existing operator can be overloaded. New operator can’t be created.
* The overloaded operator must have at least one user-defined type operand.
* Overloaded operator follow the syntax rules of the original operators. They can’t be overridden.
* The ‘precedence’ and ‘associativity’ of operator can’t be changed. It will be according to their use with built-in types.
* If an operator can be used as both unary and binary operator (like + and -), we have to overload each separately.
* The operator function for overloaded operator can’t have default arguments.
* We can’t change the number of operands that an operator takes.
* Basic meaning of operator can’t be changed. That is we cannot redefine the plus(+) operator to divide or subtract one value from the other.
* Friend function can’t be used to overload operators:

1. =
2. ( )
3. [ ]
4. ->

The operator which operates in single operand(data) are called unary operator. Example:

* Unary minus (-)
* Unary plus (+)
* Increment (++)
* Decrement (--)
* Logical not (!)
* Pointer indirection or deference (\*)
* Address of (&)

The unary operator either prefix or postfix can be overloaded.

**Syntax:**

**class** **class\_name**

{

**public:**

return\_type **operator** operator\_symbol(); //prefix

return\_type **operator** **operator\_symbol**(**int**); //postfix

};

// for prefix

return\_type class\_name :: **operator** operator\_symbol()

{

//body

}

// for postfix

return\_type class\_name :: **operator** operator\_symbol(**int**);

{

//body

}

Program to overload a unary minus operator :

#include<iostream>

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

**class** **sample**

{

**int** x,y,z;

**public:**

**void** **getdata**(**int** a, **int** b, **int** c);

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

**void** **operator** -(); //overload unary operator

};

**void** sample::getdata(**int** a, **int** b, **int** c)

{

x=a;

y=b;

z=c;

}

**void** sample::display(**void**)

{

cout<<x<<endl;

cout<<y<<endl;

cout<<z<<endl;

}

**void** sample::**operator** -()

{

x=-x;

y=-y;

z=-z;

}

**int** main()

{

sample s;

s.getdata(**10**,-**20**,**30**);

cout<<"Before activating operator function:"<<endl;

s.display(); //output= 10 -20 30

-s;

cout<<"After activating operator function:"<<endl;

s.display(); //output= -10 20 -30

**return** **0**;

}

1. **What is function over-riding? How scope resolution is used with over ridden function? Explain the need of virtual base class with example.**

**Answer:** Function Overriding in C++ is a feature that allows us to use a function in child class that is already present in its parent class. The child class inherits all the data members and the member functions present in the parent class. *The process of creating member functions in derived class with the same name as that of visible member functions of base class is called Function overriding.*

Derived class’s function can call the base class functions by using scope resolution operator (::) with base class name in the function body.

**Example:**

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

{

**protected:**

**int** num;

**public:**

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

{

base:: readdata();

cout << "enter number in derived.";

cin>> num;

}

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

{

base :: showdata();

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

}

};

If only one copy of base is actually required then we use virtual base class. When two or more objects are derived from a common base class, we can prevent multiple copies of base class from being present in an object derived from those objects by declaring the base class as ‘virtual’, when it is inherited.

Example of virtual base class:

#include<iostream>

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

**class** **student**

{

**protected:**

**int** roll;

**public:**

**void** **get\_roll**(**int** r)

{

roll = r;

}

**void** **put\_roll**()

{

cout<< roll<<endl;

}

};

**class** **test1** : **virtual** **public** student

{

**protected:**

**float** mark1;

**public:**

**void** **get\_mark1** (**float** x)

{

mark1=x;

}

**void** **put\_mark1**()

{

cout<< mark1<<endl;

}

};

**class** **test2** : **virtual** **public** student

{

**protected:**

**float** mark2;

**public:**

**void** **get\_mark2** (**float** x)

{

mark2=x;

}

**void** **put\_mark2**()

{

cout<< mark2<<endl;

}

};

**class** **result** : **public** test1, **public** test2

{

**float** total;

**public:**

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

{

total = mark1 + mark2;

put\_roll();

put\_mark1();

put\_mark2();

cout<< total<<endl;

}

};

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

{

result s;

s.get\_roll(**10**);

s.get\_mark1(**45**);

s.get\_mark2(**55**);

s.display();

**return** **0**;

}

1. **Write short notes on the access pointer and their manipulators. Write a program to make simple library management system of a college. Your program should store and retrieve the information (Book Name, Book ID, Number of books and purchase date).**

**Answer:** A pointer is used to handle and keep track of the files being accessed. The get and put are known as file position pointers and these pointers can be manipulated or repositioned to allow random access of the file.

The manipulators of these pointers are as follows:

**seekg():**

It is used to move the get pointer to the desired location concerning a reference point.

**Syntax:** file\_pointer.seekg (number\_of\_bytes ,Reference\_point);

**Example:** fin.seekg(**10**,ios::beg);

**tellg():**

tellg() is used to realize which they get pointer is in a file.

**Syntax:** file\_pointer.tellg();

**Example:** **int** posn = fin.tellg();

**seekp():**

seekp() is used to move the put pointer to the desired location concerning a reference point.

**Syntax:** file\_pointer.seekp(number\_of\_bytes ,Reference\_point);

**Example:** fout.seekp(**10**,ios::beg);

**tellp():**

tellp() is used to realize which they get pointer is in a file.

**Syntax:** file\_pointer.tellp();

**Example:** **int** posn=fout.tellp();

**Library management system**

#include<iostream>

#include<fstream>

#include<iomanip>

#include<string.h>

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

**class** **library**

{

**private:**

**char** BookName[**20**];

**int** PurchaseDate;

**int** BookId;

**int** NumberOfBooks;

**public:**

//Acessor functions

**char**\* **getBookName**()

{

**return** BookName;

}

**int** **getPurchaseDate**()

{

**return** PurchaseDate;

}

**int** **getBookId**()

{

**return** BookId;

}

**int** **getNumberOfBooks**()

{

**return** NumberOfBooks;

}

library() //default consrructor

{

PurchaseDate=**0**;

strcpy(BookName,"");

BookId=**0**;

NumberOfBooks=**0**;

}

**void** AddBook();

**void** **ShowBooks**();

**void** **header**();

**void** **listview**();

}l;

**void** **Menu**();

**void** **Heading**();

**void** **SearchByBookName**();

**void** **AddToFile**();

**void** **DisplayFromFile**();

**void** library::AddBook() //adding books

{

cout<<"**\t** Enter Book ID: ";

cin>>BookId;

cout<<"**\t** Enter Book Name: ";

cin.get();

cin.getline(BookName,**20**);

cout<<"**\t** Enter Number Of Books: ";

cin>>NumberOfBooks;

cout<<"**\t** Enter Purchase Date: ";

cin>>PurchaseDate;

}

**void** library::ShowBooks() //displaying books

{

cout<<endl;

cout<<"Book ID :"<<BookId<<endl;

cout<<"Book Name :"<<BookName<<endl;

cout<<"Number Of Books :"<<NumberOfBooks<<endl;

cout<<"Purchase Date :"<<PurchaseDate<<endl;

}

**void** library::header()

{

cout.setf(ios::left);

cout<<setw(**5**)<<"I.D."<<setw(**20**)<<"Book Name"<<setw(**20**)<<"Number of Books"<<setw(**20**)<<"Purchase Date"<<endl;

**for**(**int** i=**1**;i<=**72**;i++)

cout<<"=";

cout<<endl;

}

**void** library::listview()

{

cout.setf(ios::left);

cout<<setw(**5**)<<BookId<<setw(**20**)<<BookName<<setw(**20**)<<NumberOfBooks<<setw(**20**)<<PurchaseDate<<endl;

}

//function declaration

**void** heading()

{

cout<<"\*\*\*\*\*\*\*\*\*\*LIBRARY MANAGEMENT SYSTEM\*\*\*\*\*\*\*\*\*\*"<<endl;

}

**void** AddToFile() //writing data to file

{

ofstream fout;

fout.open("books.txt",ios::app);

l.AddBook();

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

cout<<"book data saved in file......**\n**";

fout.close();

}

**void** DisplayFromFile() //reading data from file

{

ifstream fin("books.txt");

**int** rec=**0**;

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

{

**if**(rec<**1**)

l.header();

l.listview();

rec++;

}

fin.close();

cout<<"**\n**Total"<<rec<<"Records in file.....**\n**";

}

**void** SearchByBookName()

{

**int** flag=**0**;

**char** bookname[**20**];

ifstream **fin**("books.txt");

cout<<"Enter Book Name:";

cin.ignore();

cin.getline(bookname,**20**);

**while**(fin.read((**char**\*)&l,**sizeof**(l)))

{

**if**(strcmpi(bookname,l.getBookName())==**0**)

{

l.ShowBooks();

flag++;

}

}

fin.close();

**if**(flag==**0**)

cout<<"Book with Name::"<<bookname<<"not available...**\n**";

}

**void** Menu()

{

**int** ch;

**do**

{

system("cls");

heading();

cout<<"0. EXIT **\n**";

cout<<"1. Add New Books **\n**";

cout<<"2. Show All Books **\n**";

cout<<"3. Search Books **\n**";

cout<<"Enter Your Choice **\n**";

cin>>ch;

system("cls");

heading();

**switch**(ch)

{

**case** **1**: l.AddBook(); **break**;

**case** **2**: l.ShowBooks(); **break**;

**case** **3**: SearchByBookName(); **break**;

}

system("pause");

}**while** (ch!=**0**);

}

**int** main()

{

Menu();

**return** **0**;

}

1. **Explain the need of virtual function with suitable example. Define runtime type information (RTTI). How dynamic cast and typeid operators are used to achieve RTTI?**

**Answer:** A virtual function is a member function in base class that we expect to redefine in derived classes. A virtual function is used in the base class in order to ensure that the correct function id overridden.

**Example:**

#include<iostream>

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

**class** **animal**

{

**public:**

**virtual** **void** display() //virtual function

{

cout<<"from base class animal"<<endl;

}

};

**class** **cow** : **public** animal

{

**public:**

**void** display ()

{

cout<<"from derived class cow"<<endl;

}

};

**class** **dog** : **public** animal

{

**public:**

**void** display()

{

cout<<"from derived class dog"<<endl;

}

};

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

{

animal \*panim;

animal anim;

cow cw;

dog dg;

panim = &anim;

panim-> display();

panim = &cw;

panim-> display();

panim = &dg;

panim-> display();

**return** **0**;

}

//OUTPUT:

/\* from base class animal

from derived class cow

from derived class dog \*/

During compilation, pointer always selects the base class function as it is a pointer to the base class object. That is why, the virtual function is created in base class to prevent the pointer to select base class function.

Output without virtual function:

/\* from base class animal

from base class animal

from base class animal \*/

Run-Time Type Information (RTTI) is the feature of oop that helps us to change the object of one type to the object of another type during run time and it also helps to provide the information of object during run time.

RTTI is achieved by using two operators. They are:

* dynamic\_cast Operator
* typeid Operator

**Dynamic\_cast Operator:**

The dynamic cast operator will help to change object of one type to the object of another type during run time and also checks the validity of cast.

Syntax:

**dynamic\_cast** <target\_type> expression;

**Typeid Operator:**

The typeid () operator will help to give information of object during run-time.

Syntax:

**typeid** (expression);

1. **Explain how default arguments are used with class template with example. How do you throw only specified expectation from a function? Exemplify.**

**Answer:** Like the default arguments with normal function the class template can have a default argument associated with template parameter.

Syntax for Declaration:

**template** <**class** **template\_name** = default\_data\_type>

**class** **class\_name**

{

//data and function declaration

};

Example:

**template** <**class** **T**=**int**>

**class** **test**

{

//data member and member function declaration

};

test<**float**> ft; // template argument is float

test<> **int**; // template argument int when not supplied

Example Program using default argument with class template:

#include <iostream>

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

**template** <**class** **T**, **class** **U** = **char**>

**class** **A** {

**public:**

T x;

U y;

A() //default constructor

{

cout << "Constructor Called" << endl;

}

};

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

{

A<**char**> a; // This will call A<char, char>

**return** **0**;

}

/\* OUTPUT: Constructor Called \*/

It is possible to restrict a function to throw only certain specific exceptions. This is achieved by adding throw list.

Syntax:

return\_type **function\_name** (argument\_list) **throw** (type1, type2,....)

{

//body of function

}

Following are different ways of specifying exception list with functions:

**void** **func1**(); //can throw any exceptions

**void** **func2**() **throw**(); //cannot throw any exception

**void** **func3**() **throw**(x); /\*throws only x exceptions and exceptions derived from x \*/

**void** **func3**() **throw**(x,y); /\*throws only x and y exception and exception

Example:

#include <iostream>

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

// Here we specify the exceptions that this function

// throws.

**void** **fun**(**int** \*ptr, **int** x) **throw** (**int** \*, **int**) /\* Dynamic Exception specification\*/

{

**if** (ptr == NULL)

**throw** ptr;

**if** (x == **0**)

**throw** x;

/\* Some functionality \*/

}

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

{

try {

fun(NULL, **0**);

}

**catch**(...) {

cout << "Caught exception from fun()";

}

**return** **0**;

}