2078 Bhadra Solutions

1. Is Object Oriented Programming is better than Procedure Oriented Programming? If yes support with appropriate statements. Explain the features of object oriented programming in brief.

Answer: Yes, object oriented programming is better than procedure oriented programming because of the following reasons:

* Data can be hidden in OOP which can prevent from accidental modification whereas data can’t be hidden in POP.
* Code maintenance and enhancement is easy in OOP whereas it is difficult in POP.
* Reusability of code is difficult in POP whereas it is much easy in OOP compared to POP.

The features of object oriented programming is as follows:

* Objects:

Objects are defined as the physical instance of a class. An Object can have attributes (properties) and behaviors when the object is created, space for that object is allocated.

For example a T.V. is an object having size, weight, color, etc. as attributes and turning on/off T.V., increasing/decreasing volume, etc. as operation or function.

* Classes:

Classes are templates/model/blueprints that specifies data and their operations. Basically, class is the design off which objects are made. Once a class is defined, we can create any number of objects of its type.

For example “Fruit” is a class of objects like ‘Orange’, ‘Apple’, etc.

Classes are user defined data types and behave like the built in types of programming language.

* Abstraction:

Abstraction feature of OOP hides the internal details of how any objects works. It only provides the interface to use the service that the object provides. This feature helps to manage complexity.

For example a class ‘car’ is made of various components like engine, gearbox, brakes, etc. The abstraction allows the user to drive the car without knowing the complexity of parts.

* Encapsulation:

The wrapping up of data and function into a single unit is called encapsulation. Data is not accessible in OOP to the outside world and is only accessible to the member function because of encapsulation.

* Polymorphism:

The word “polymorphism” is made up of two words ‘poly’ meaning “many” and “morphe” meaning “forms”. Thus, the word polymorphism means the ability to take many forms. In OOP, an operation may exhibit different behaviors in different instances. For example operator overloading, function overloading, dynamic binding, etc.

1. Define constructor and destructor. Explain different types of constructor with suitable example.

Answer: A constructor is a special member function of a class having the same name of class which initializes member of its object.

A destructor is special member function of a class having the same name as a constructor which deallocates the memory, initialized by the constructors.

The different types of constructors are as follows:

* Default constructors:

The constructor which has no arguments is known as default constructor. It is automatically called when no argument is supplied while creating the object. Example:

#include<iostream>

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

**class** **Time**

{

**int** hr,min,sec;

**public:**

Time()

{

hr=**10**;

min=**10**;

sec=**10**;

}

**void** display()

{

cout<<hr<<":"<<min<<":"<<sec<<endl;

}

};

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

{

Time t1;

t1.display();

**return** **0**;

}

* Parameterized constructor:

The constructor which takes some argument is known as parameterized constructor. We should supply some arguments while defining the object of the class. Example:

#include<iostream>

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

**class** **Time**

{

**int** hr,min,sec;

**public:**

Time(**int** a,**int** b,**int** c)

{

hr=a;

min=b;

sec=c;

}

**void** display()

{

cout<<hr<<":"<<min<<":"<<sec<<endl;

}

};

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

{

Time t1(**10**,**10**,**10**);

t1.display();

**return** **0**;

}

* Copy constructor:

The constructor that initializes data members of the object by copying the value of another object initialized by either default or parameterized constructor is called copy constructor. Example:

#include<iostream>

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

**class** **Time**

{

**int** hr,min,sec;

**public:**

Time(**int** a,**int** b,**int** c)

{

hr=a;

min=b;

sec=c;

}

Time(Time &a)

{

hr=a.hr;

min=a.min;

sec=a.sec;

}

**void** display()

{

cout<<hr<<":"<<min<<":"<<sec<<endl;

}

};

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

{

Time t1(**10**,**10**,**10**);

Time t2(t1);

t2.display();

**return** **0**;

}

1. How do you compare C and C++? Explain different components (Lexical elements) of C++.

Answer: The comparison between C and C++ is given below:

|  |  |
| --- | --- |
| C | C++ |
| a)This language was introduced in 1972 by Dennis Ritchie at Bell Lab.  b)It follows procedural approach of program development.  c)C applications are faster to compile than C++ applications.  d)C has fewer libraries and keywords than C++.  e)C has comparatively lesser implementation than C++.  f)C does not support extension in programming.  g)This language is influenced by B, BCPL, ALGOL 68, etc.  h)Languages such as C++, perl, php, javascript are influenced by C. | a)This language was introduced in 1985 by Bjarne Stroustup at Bell Lab.  b)It follows object oriented approach of program development.  c)C++ applications are slower to compile than C applications.  d)C++ has more libraries and keywords than C.  e)C++ has comparatively higher implementation than C.  f)C++ supports extension in programming.  g)This language is influenced by C, Simula 67, Ada 83, etc.  h) Languages such as Java, C# are influenced by C++. |

C++ programs statements are constructed by many different small components such as commands, variables, constants, and many more symbols. These individual elements are collectively called Lexical elements.

The lexical elements of C++ are explained below:

* Comments:

Comments are the description of the source which written in the program. Comments are the non functional lines of a program which are skipped by the copiler. // (double slash) is used for comments.

* Tokens:

Tokens are valid sets (collection) of different characters, symbols, operators or punctuators. Tokens are the smallest individual units in a program. Different types of tokens that are used are:

a)Keywords: Keywords are predefined or reserved words of the C++ library. All keywords have fixed meanings and these meanings cannot be changed.

b)Identifiers: The name given to variables, functions, arrays, classes etc. by the programmers is called identifiers. These are tokens which are sequence of letters, digits, and underscore (\_). Identifiers are used to gibe unique names to the element in the program.

c)Constants: Constants also called ‘literals’ are the fixe values that cannot be changed during execution of the programs. The various constants of C++ are Numeric constants, Character constants and String constant.

d)Operators: Operators are special symbols which are used for logical operations. Some of the operators used in C++ are:

i) Arithmetic operators (+,-,\*, /, %)

ii) Assignment (=)

iii) Self assignment (++, --)

iv) Relational operator (<, >, <=, >=)

v) Other operators (?:, (), [], ->,~, sizeof())

1. How does inline function differ from a pre-processor macro? What is the main difference of passing argument by reference? Illustrate with a suitable example.

Answer: Inline function is the feature of C++ in which the function call is replaced by the function codes whereas pre-processor macro also replaces the ‘name’ called with the lines of code defined to it. However a pre-processor macro is defined using “#define” at the global section of the program whereas inline function can be defined anywhere by using the keyword ‘inline’.

The main difference of passing argument by reference is that the memory address of the argument of passed instead of the argument itself.Example:

#include<iostream>

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

**float** &max(**float** &a,**float** &b)

{

**if**(a>b)

{

**return** (a);

}

**else**

{

**return** (b);

}

}

**int** main()

{

**float** t1,t2;

cout<<"Enter two temperatures.";

cin>>t1>>t2;

max(t1,t2)=**100**;

cout<<**100**<<endl;

cout<<t1<<endl;

cout<<t2;

**return** **0**;

}

1. Explain the rules of operator overloading in C++. Write a program to concatenate two user given strings using the concept of operator overloading.

Answer: The rules of operator overloading in C++ are as follows:

* Only existing operators can be overloaded. New operator cannot be created.
* The overloaded operator follow the syntax rule of the original operator. They cannot be overridden.
* Basic meaning of the operator cannot be changed. That is we cannot redefine the plus operator (+) to divide or subtract.
* If an operator can be both used as unary and binary operator then we have to overload them separately.
* The ‘precedence’ and ‘associativity’ of operator cannot be changed. It will be according to their use with built-in types.
* The operator function for operator overloading cannot have default argument.

String concatenation using operator overloading:

#include<iostream>

#include<string.h>

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

**class** **mstring**

{

**private:**

**char** s[**20**];

**public:**

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

{

cout<<"Enter the string.";

cin>>s;

}

**friend** mstring **operator** +(mstring a,mstring b)

{

mstring c;

strcpy(c.s,a.s);

strcat(c.s," ");

strcat(c.s,b.s);

**return** c;

}

**void** display()

{

cout<<"The string is "<<endl;

cout<<s;

}

};

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

{

mstring s1,s2,s3;

s1.input();

s2.input();

s3=s1+s2;

s3.display();

**return** **0**;

}

1. Explain the need of virtual base class with example. Write a program to show the order of constructor invocation in multiple inheritance.

Answer: Virtual base classes are needed in C++ to prevent multiple instances of a given class from appearing in an inheritance hierarchy in case of multiple inheritance that this the diamond problem of inheritance.

The program below illustrates the above mentioned case:

#include<iostream>

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

**class** **raja**

{

**protected:**

**int** a;

**public:**

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

{

cout<<"Enter the value of a.";

cin>>a;

}

};

**class** **kaji\_1**:**virtual** **public** raja

{

**protected:**

**int** b;

**public:**

**void** **get**()

{

cout<<"Enter the value of b.";

cin>>b;

}

};

**class** **kaji\_2**:**virtual** **public** raja

{

**protected:**

**int** c;

**public:**

**void** **take**()

{

cout<<"Enter the value of c.";

cin>>c;

}

};

**class** **paji**:**public** kaji\_1, **public** kaji\_2

{

**protected:**

**int** d;

**public:**

**void** **set**()

{

cout<<"Enter the value of d.";

cin>>d;

}

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

{

cout<<"The values are:"<<endl;

cout<<"a="<<a<<endl;

cout<<"b="<<b<<endl;

cout<<"c="<<c<<endl;

cout<<"d="<<d<<endl;

}

};

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

{

paji p;

p.input();

p.get();

p.take();

p.set();

p.display();

**return** **0**;

}

The program to show the constructor invocation in multiple inheritance is below:

#include<iostream>

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

**class** **c1**

{

**protected:**

**int** a;

**public:**

c1()

{

cout<<"Constructor of base class called."<<endl;

}

};

**class** **c2**:**public** c1

{

**protected:**

**int** b;

**public:**

c2()

{

cout<<"Constructor of intermediate base class called."<<endl;

}

};

**class** **c3**:**public** c2

{

**protected:**

**int** c;

**public:**

c3()

{

cout<<"Constructor of derived class called.";

}

};

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

{

c3 s;

**return** **0**;

}

1. Explain the need of virtual function with suitable examples. How dynamic cast and typeid operators are used to achieve RTTI?

Answer: A virtual function is a member function that we expect to redefine in derived class. A virtual function is needed so that it can be overridden. This especially applies to a case where a pointer of base class points to an object of derived class. Example:

#include<iostream>

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

**class** **Animal**

{

**public**:

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

{

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.";

}

};

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

{

Animal \*panm;

Animal anm;

Cow c;

Dog d;

panm=&anm;

panm->display();

panm=&c;

panm->display();

panm=&d;

panm->display();

**return** **0**;

}

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);

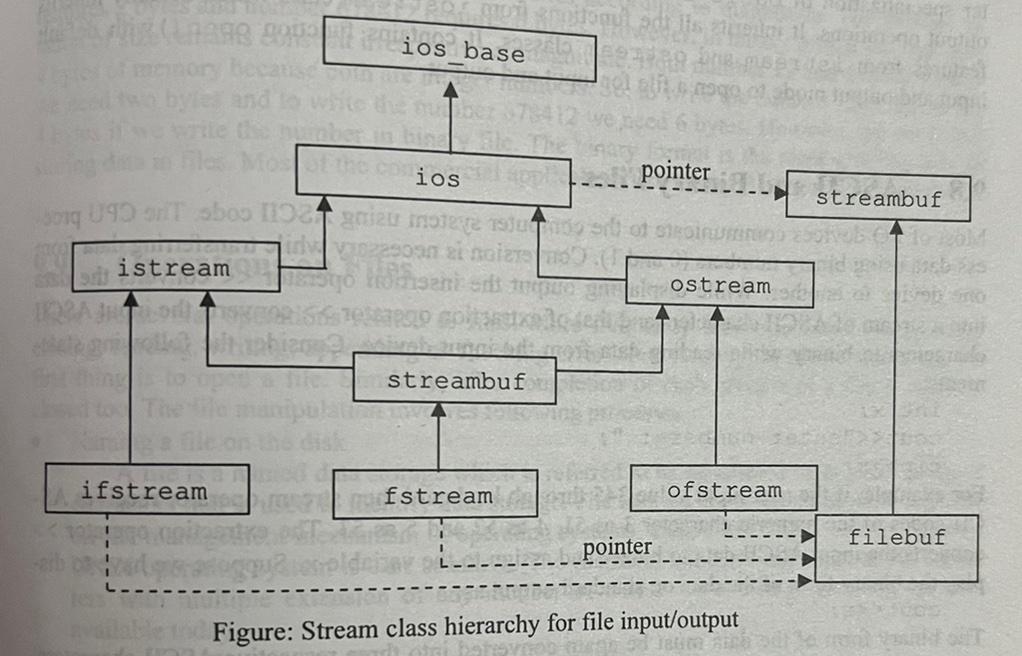
The typeid operator is like a function as that of sizeof operator. When an expression is passed as its operand, typeid returns a reference to a type\_info class type that represents the type of the object denoted by the expression. When a type name is passed as its operand, typeid returns a reference 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.

1. Discuss about classes for file stream operator with a suitable block diagram. Write a program to write the information of students in a file. And also display their details in console.

Answer: The various classes for stream operator are as follows:

* Ostringstream: The ostringstream is a derived class of class ostream and is used for formatted output.
* Istringstream: The istringstream is a derived class of class istream and is used for input.
* Stringstream: The stringstream is derived from iostream and is used for both input and output.

To use all these classes we need the header file <sstream>.



#include<iostream>

#include<fstream>

#include<stdlib.h>

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

**class** **student**

{

**private:**

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

**int** roll;

**int** marks;

**public:**

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

{

cout<<"Enter the student roll no.:"<<endl;

cin>>roll;

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

cin>>name;

cout<<"Enter the student marks."<<endl;

cin>>marks;

}

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

{

cout<<"Student roll:"<<roll<<endl;

cout<<"Student name:"<<name<<endl;

cout<<"Student marks:"<<marks<<endl;

}

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

{

fstream fout;

student s;

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

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

s.input();

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

fout.close();

}

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

{

fstream fin;

student s;

fin.open("student.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 student:"<<endl;

student s1[**5**];

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

{

s1[i].add();

}

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

**return** **0**;

}

1. Explain how default arguments are used in template. Define class template and all of its function members with suitable examples.

Answer: Like the default argument with normal functions the class template can have a default argument associated with template parameter. The default template class argument is specified in the template declaration in the following form:

template<class\_template\_type=default\_data\_type>

class class\_name

{

//data and function declarations;

}

Where template\_type is the template parameter for generic data type and default\_data\_type is the type name to be used when template argument is not specified. The actual class template with default template argument is as:

template<class T = int>

class test

{

//……

}

Class template is defined as the blueprint or formula of creating a generic class. In other words, it is the basic design for creating a class of any data type.

Template class is declared and defined in the syntax shown below:

template <class template\_type>

class class\_name

{

private:

//data member of template or non template type

public:

//function members with template type argument and return type

};

When the function of a template class is defined outside the class then the member function must be explicitly declared as a template.

The member function of template class can be accessed in the syntax shown below:

template <class template\_type>

class class\_name

{

private:

//template\_type variable\_name

public:

//return\_type function\_name(template\_type argument)

};

template <class template\_type>

return\_type class\_name<template\_type>::

function\_name(template\_type argument)

{

body of function…

}

Example:

#include<iostream>

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

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

**class** **number**

{

**private:**

T n,m;

**public:**

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

T **sum**();

};

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

**void** number<T>::input()

{

cout<<"Enter two numbers.";

cin>>n>>m;

}

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

T number<T>::sum()

{

T d;

d=n+m;

**return** d;

}

**int** main()

{

number<**int**> Y;

number<**float**> Z;

Y.input();

cout<<"The sum is:"<<Y.sum()<<endl;

Z.input();

cout<<"The sum is:"<<Z.sum()<<endl;

**return** **0**;

}

1. How is exception handling better than conventional handling? Explain how multiple exceptions are handled with a suitable example.

Answer: Conventional error handling is inconvenient because every call must be surrounded with if else …… statements to handle error or call the error routine. This increases the size of program and makes the product bulky. Whereas a single exception handling can handle all the errors helping in reducing the size of program. Plus exception handling separates the code from the program making it more readable.

In programming there will always be the chance of encountering multiple errors at once during runtime. In such case, C++ permits to design to throw as many exceptions as needed. The following program can illustrate multiple exception handling:

#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."<<endl;

}

**catch** (**int** i)

{

cout<<"Caught an integer."<<endl;

}

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

{

cout<<"Caught a character."<<endl;

}

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

{

cout<<"Caught a double."<<endl;

}

}

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

{

cout<<"Testing multiple catches."<<endl;

cout<<"x==1"<<endl;

test(**1**);

cout<<"x==0"<<endl;

test(**0**);

cout<<"x==-1"<<endl;

test(-**1**);

cout<<"x==2"<<endl;

test(**2**);

**return** **0**;

}