# 2079 Baishak

1. *What are the drawbacks of procedural programming and the advantages of object-oriented programming? With a program code differentiate between procedure-oriented programming and object-oriented programming?*

Ans: The drawbacks of procedural programming are as follows:

* Focus on functions rather than data.
* The use of global data is error prone and it could be an obstacle in code maintenance and enhancement.
* It is difficult to hide information to unauthorized user.

The advantages of object-oriented programming are as follows:

* Data can be hidden to prevent from accidental modification.
* It supports reusability of code, inheritance, polymorphism, data abstraction and encapsulation.
* Emphasis is on data rather than procedures.

Code to differentiate POP and OOP:

Program to add two numbers:

**In POP:**

#include<iostream>  
using **namespace** std;  
  
int **main**()  
{  
 **int** num1, num2, add;  
 cout<<"Enter Two Numbers: ";  
 cin>>num1>>num2;  
 add = num1+num2;  
 cout<<"**\n**Result = "<<add;  
 cout<<endl;  
 **return** **0**;  
}

**In OOP:**

#include <iostream>  
using **namespace** std;  
class **A** {  
 **private:**  
 **int** num1,num2,num3;  
 **public:**  
 **void** **getdata**(){  
 cout<<"Enter two numbers:"<<endl;  
 cin>>num1>>num2;  
 }  
 **void** **add**() {  
 num3 = num1 + num2;  
 cout<<"Result="<<num3<<endl;  
 }  
};  
  
int **main**() {  
 A a;  
 a.getdata();  
 a.add();  
 **return** **0**;  
}

1. *Explain the order of constructor and destructor invocation with example. When do we use static data member and static member function in a class? Explain with example.*

Ans: C++ constructor invocation order will be from top to down that is from base class to derived class and c++ destructor call order will be in reverse order, which can be clearly understood by following example.

Constructor and Destructor invocation in Multiple Inheritance:

#include <iostream>  
using **namespace** std;  
class **Base1**{  
 **public:**  
 Base1(){  
 cout<<"Base1 class constructor"<<endl;  
 }  
 ~Base1(){  
 cout<<"Base1 class destructor"<<endl;  
 }  
};  
  
class **Base2**{  
 **public:**  
 Base2(){  
 cout<<"Base2 class constructor"<<endl;  
 }  
 ~Base2(){  
 cout<<"Base2 class destructor"<<endl;  
 }  
};  
  
class **Derived**: **public** Base2, **public** Base1 {  
 **public:**  
 Derived(){  
 cout<<"Derived class constructor"<<endl;  
 }  
 ~Derived(){  
 cout<<"Derived class destructor"<<endl;  
 }  
};  
  
  
int **main**() {  
 Derived d;  
  
 **return** **0**;  
}

Output:

Base2 class constructor

Base1 class constructor

Derived class constructor

Derived class destructor

Base1 class destructor

Base2 class destructor  
  
  
2nd Part:

A static data member is used when all objects of the same class must share common information. Unlike regular data members, individual copies of static member variables are not made for each object of a class, only one copy of a static data member exist.

A class may also have a static function as its member, it is defined by using the keyword ‘static’ before the member function. Static functions can access only static data. They are useful in accessing private static data.  
  
The following example shows use of static data member and static function:

#include<iostream>  
using **namespace** std;  
class **element** {  
private:  
 **static** **int** count; //static data member  
 **int** data;  
public:  
 element ()  
 {  
 count++;  
 data=count;  
 }   
~element ()  
{  
count--;  
cout<<"Destroying element with value "<<data<<endl;  
}  
static **void** showcount () // static function  
{  
cout<<"Number of elements are: "<<count<<endl;  
}  
void showdata () {  
cout<<"The data is:"<<data<<endl;  
}  
};  
int element::count=**0**;  
int **main**() {  
element s1;  
element:: showcount ();  
element s2,s3;  
element::showcount ();  
s1.showdata();  
s2.showdata();  
s3.showdata ();  
return **0**;  
}

1. Write down the brief history of C++. Compare C with C++ with example.

Ans: C++ was developed by Bjarne Stroustrup at Bell Laboratories over a period starting in 1979. Since C++ is an attempt to add object-oriented features (plus other improvements) to C, earlier it was called as “C with Objects”. As the language developed, Stroustrup named it as C++ in 1983. The name C++ suggests “C incremented” (recall the ++ is an increment operator of C).

The comparison between C and C++ are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| SN | C | SN | C++ |
| 1. | This language was introduced in 1972 by Dennis Ritchie at Bell Lab. | 1. | This language was introduced in 1985 by Bjarne Stroustrup at Bell Lab. |
| 2. | It follows procedural approach of program development. | 2. | It follows object-oriented program development. |
| 3. | C applications are faster to compile than C++ applications. | 3.. | C++ applications are slower to compile than C applications. |
| 4. | C has fewer libraries, keywords than C++. | 4. | C++ has more libraries, keywords than C. |

The example is shown by comparing C and C++ program:

A program to calculate Area of Cirlce:

|  |  |
| --- | --- |
| C | C++ |
| #include < stdio.h >  #include < conio.h >  #define PI 3.141  int **main**()  {   **float** radius, area;   printf("Enter radius of circle**\n**");   scanf("%f", & radius);   area = PI \* radius \* radius;   printf("Area of circle: %0.4f**\n**", area);   getch();   **return** **0**;  } | #include <iostream> using **namespace** std;   **int** **main**() {  **float** radius, area\_circle;   cout <<"Enter the radius of circle:";  cin >>radius;   area\_circle = **3.14** \* radius \* radius;  cout <<"Area of circle:"<<area\_circle << endl;   **return** **0**; } |

1. Explain how function selection is done in function overloading? Where can enumerate data types be used in C++ programming?

Ans: When the same name of function is used for different operation then it is called function overloading. It is done either by passing different number of arguments or different types of arguments.

When an overloaded function is called the function with matching arguments is invoked. When a function is called, the compiler figures out which of the function is to be selected. This is done by the compiler by comparing the types and numbers of the actual arguments to type and numbers of formal arguments. So, the function is selected according to the best match on the arguments otherwise shows error if there is no match. It can be illustrated by following example:

**Program:**

#include<iostream>  
using **namespace** std;  
int **volume**(**int**);  
double **volume**(**double**, **int**);  
long **volume**(**long**, **int**, **int**);  
 **int** **main**(){  
cout<<volume(**5**)<<endl;  
cout<<volume(**1.5**,**5**)<<endl;  
cout<<volume(**100l**,**75**,**10**)<<endl;  
  
return **0**;  
  
 }  
  
 **int** **volume** (**int** s){  
 **return**(s\*s\*s);  
 }  
  
 **double** **volume**(**double** r, **int** h) {  
 **return**(**3.145**\*r\*r\*h);  
 }  
  
 **long** **volume** (**long** l, **int** b, **int** h){  
 **return**(l\*b\*h);   
 }

The overloaded function volume() is selected by comparing the types and number of arguments of actual arguments to formal parameter.  
  
  
The enumeration data type is a construct taken from C. The enum keyword is used to declare the enumeration types. It is mainly used to assign names to integral constants, that makes a program easy to read and maintain.

For example if enumeration is defined as:  
enum fruit {apple, banana, grape, mango}; then   
apple gets value 0, banana 1, grape 2, mango 3.

1. How do you convert one class type to another class type? Write a program to overload the relational operators (> and ==) to compare two distance objects using non-member function.

Ans: The conversion from object of one class to object of another class can be performed by two methods:

1. Conversion in Destination Object  
   If the basic type is replaced with user defined in basic to user-defined conversion, then the same logic can be used to specify the conversion routine from any class to destination class.  
   Eg:  
   ClassA objA;  
   ClassB objB;  
   objA=objB;
2. Conversion in Source Class  
   Like the conversion from user defined to basic data types the conversion can be done by defining the cast operator of the destination class type.

**Program:**

#include <iostream>  
using **namespace** std;  
class **Distance** {  
 **private:**  
 **int** len;  
 **public:**  
 **void** **input**(){  
 cout<<"Enter the value of distance"<<endl;  
 cin>>len;  
   
 }  
 **friend** **int** **operator** >(Distance len1, Distance len2);  
 **friend** **int** **operator** ==(Distance len1, Distance len2);  
   
};  
  
 **friend** **int** **operator** >(Distance len1, Distance len2) {  
 **if**(len1.len>len2.len)  
   
 { **return** **1**;}  
 **else**   
 { **return** **0**;}  
 }  
   
 **friend** **int** **operator** ==(Distance len1, Distance len2) {  
 **if**(len1.len == len2.len)  
 { **return** **1**;}  
 **else**   
 { **return** **0**;}  
   
 }  
 **int** main() {  
 Distance len1, len2;  
 len1.input();  
 len2.input();  
 **if** (len1 > len2)   
 { cout<<"Length 1 is greater"<<endl;}  
 **else** **if**(len1 == len2)   
 {cout<<"Both length are equal"<<endl;}  
 **else**  
 {cout<<"Length 2 is greater"<<endl;}  
   
return **0**;  
   
 }

1. Explain multipath and multiple inheritances. Write a program to demonstrate example of Hierarchical inheritance.

Ans: Derivation of a class from other derived classes which are derived from same base class is called multipath inheritance. It is a hybrid inheritance which is combination of multiple, multilevel and hierarchical inheritance.



Fig: Multipath Inheritance

Multiple inheritance is a type of inheritance in which new class is created from more than one base class i.e., a single derived class uses behavior of more than two base classes.



Fig: Multiple Inheritance

Synatx:

class A {  
…....

};

class B {

…...

};

class C : visibiltiy\_mode A, visibility\_mode B {

…....

};

**Program of Hierarchical Inheritance:**

#include<iostream>  
using **namespace** std;  
class **employee**  
{  
protected:  
 **float** salary;  
public:  
 **void** **get\_sal**(**float** s){  
 salary = s;  
 }  
};  
class **HOD**: **public** employee  
{  
private:  
 **float** bonus;  
public:  
 **void** **get\_bonus**(**float** b) {  
 bonus = b;  
 }  
 **void** **put\_net**() {  
 cout<<"Net salary ="<<salary + bonus;  
 }  
};  
class **DHOD**: **public** employee {  
 **float** bonus;  
 **public:**  
 **void** **get\_bonus**(**float** b){  
 bonus = b;  
 }  
 **void** **put\_net**(){  
 cout<<"Net salary="<<salary+bonus;  
 }  
   
};  
int **main**(){  
 HOD h;  
 cout<<"Salary of HOD"<<endl;  
 h.get\_sal(**15000**);  
 h.get\_bonus(**5000**);  
 h.put\_net();  
  
 DHOD d;  
 cout<<"Salary of DHOD"<<endl;  
 d.get\_sal(**10000**);  
 d.get\_bonus(**4000**);  
 d.put\_net();  
 **return** **0**;  
   
}

1. How can you eliminate member function overriding in virtual function? Consider a book shop which sells both books and video-tapes. Create a class known as media that stores the title and price of a publication. Then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of tape.

Ans: When pointer to base class is created, pointer always selects the base class function as it is the pointer to the base class object. In C++ we have provision for selecting the particular function as according to the address of the object the pointer holds. Thus, to eliminate the member function overriding in virtual function the pointer should point to the object of derived class.  
  
  
**Program:**  
  
#include<iostream>  
using **namespace** std;  
  
class **media** {  
 **protected:**  
 **char** title[**20**];  
 **int** price;  
  
 **public:**  
 **void** **getdata**() {  
 cout<<"Enter title of publication"<<endl;  
 cin>>title;  
 cout<<"Enter price of publication"<<endl;  
 cin>>price;  
 }  
 **void** **showdata**() {  
 cout<<"**\n** Title:"<<title;  
 cout<<"**\n** Price:"<<price;  
 }  
  
};  
class **Pages** : **public** media{  
 **protected:**  
 **int** nopage;  
 **public:**  
 **void** **getdata**() {  
 media::getdata();  
 cout<<"Enter number of pages"<<endl;  
 cin>>nopage;  
 }  
 **void** **showdata**(){  
 media::showdata();  
 cout<<"**\n**Number of Pages"<<nopage;  
 }  
   
};  
class **ptape** : **public** media {  
 **protected:**  
 **int** playtime;  
 **public:**  
 **void** **getdata**(){  
 media::getdata();  
 cout<<"Enter playing time of tape"<<endl;  
 cin>>playtime;  
 }  
  
 **void** **showdata**() {  
 media::showdata();  
 cout<<"**\n**Playing time"<<playtime;  
 }  
};  
  
int **main**() {  
 media m;  
 Pages pg;  
 ptape pt;  
 m.getdata();  
 pg.getdata();  
 pt.getdata();  
 m.showdata();  
 pg.showdata();  
 pt.showdata();  
return **0**;  
}

1. Explain how do you achieve random access to file? Write a program to store and retrieve ‘n’ records of items (item\_ID, name, price, mfd\_date, company) in inventory system.   
   Ans:  
   Random access is necessary for writing and reading contents of a file at a specific location, modifying particular record or deleting some desired records. For random access to file, we use file access pointer function *seekg()* for input and *seekp()* for output.   
     
     
   **Program:**  
     
   #include<iostream>  
   #include<fstream>  
   #include<stdlib.h>  
   using **namespace** std;  
   class **Inventory** {  
    **private:**  
    **int** id, price, mfddate;  
    **char** name[**20**], company[**20**];  
    **public:**  
    **void** **input**(){  
    cout<<"**\n**Enter the id of product:";  
    cin>>id;  
    cout<<"**\n** name:";  
    cin>>name;  
    cout<<"**\n**Price:";  
    cin>>price;  
    cout<<"**\n**Mfd-Date:";  
    cin>>mfddate;  
    cout<<"**\n**Enter name of Company:";  
    cin>>company;  
    }  
     
    **void** **display**() {  
    cout<<"**\n**Product ID:"<<id;  
    cout<<"**\n**Product Name:"<<name;  
    cout<<"**\n**Price:"<<price;  
    cout<<"**\n**Mfd-DAte:"<<mfddate;  
    cout<<"**\n**Name of Company:"<<company;  
    }  
     
    **void** **add**() {  
    fstream fout;  
    Inventory Is;  
    fout.open("inventory.txt",ios::app | ios::out | ios::binary);  
    cout<<"**\n**The Inventory System record:";  
    Is.input();  
    fout.write((**char** \*)&Is,**sizeof**(Is));  
    fout.close();  
    }  
     
    **void** **displayall**()  
    {  
    fstream fin;  
    Inventory Is;  
    fin.open("inventory.txt", ios::in | ios::binary);  
    fin.seekg(**0**);  
    fin.read((**char** \*)&Is,**sizeof**(Is));  
    while(!fin.eof())  
    {  
    Is.display();  
    fin.read((**char** \*)&Is,**sizeof**(Is));  
    }  
    fin.close();  
    }  
     
     
   };  
   int **main**()  
   {  
    **int** n;  
    cout<<"Enter the detail of the product:"<<endl;  
    cout<<"Enter the number of products:"<<endl;  
    cin>>n;  
    Inventory s1[n];  
    for(**int** i=**0**;i<n;i++)  
    {  
    s1[i].add();  
    }   
    s1[**0**].displayall();  
    return **0**;  
   }
2. What are the uses of Function Template? Explain the case when all the template parameters are not used in function arguments. Write a program that illustrates the overloading of two function template.  
     
   Ans:  
   Function template allows us to write generic programming.  
   We can create a single function to work with different data types by using a template. It promotes code reusability and demotes code redundancy.  
     
   When all the template parameters are not used in function argument, as shown in example.  
   Template<class T1, class T2>  
   void test\_func(T2 a)  
   {  
   T1 b;  
   //........  
     
   }  
     
   The compiler fails to compile because the compiler cannot deduce the template parameter T1, as T2 is only used in the function parameter list. To solve this problem function instantiation is done explicitly by specifying the template parameter as   
   test\_func <float> (inum)  
     
     
   **Program:**  
     
   #include<iostream>  
   using **namespace** std;  
     
   template <**class** **T**>  
   void display(T data) {  
    cout<<data<<endl;  
   }  
     
   template <**class** **T**>  
   void display(T data, **int** n) {  
    **for** (**int** i = **0**; i < n; i++)  
    {  
    cout<<data<<endl;  
    }  
      
   }  
   int main() {  
    display(**1**);  
    display(**2.5**);  
    display(**420**,**2**);  
    display("Hello World",**3**);  
    **return** **0**;  
   }
3. Explain exception along with exception handling mechanism. Write a program to demonstrate example of rethrowing exception.  
   Ans:  
   Exceptions are errors that occur at run time. They are also called anomalies. They are caused by a wide variety of exceptional circumstances, such as running out of memory, not being able to open a file, etc.   
   Detecting abnormal behaviour of the software at the runtime and taking preventive measure is called exception handling. In C++, the exception handling mechanism uses three keywords try, throw and catch. The part of the code that can generate the exception or call the function that generates exception should be placed within the try block and the part of the code to handle appropriate exception should be placed within the catch block. When the condition for exception arises, the condition is indicated by throwing the exception. The condition for exception is detected by placing the detecting code within the try block. The condition for exception is indicated by throwing the exception. The thrown exception is caught and handled by the catch part of the exception handling construct.  
     
   **Program:**  
     
   #include<iostream>  
   using **namespace** std;  
   void **divide**(**double** x, **double** y) {  
    cout<<"Inside function**\n**";  
     
     
   try {  
    **if** (y==**0.0**)  
    **throw** y;  
    **else**   
    cout<<"**\n**Division ="<<x/y;  
   }  
   catch(**double**) {  
    cout<<"Caught double inside function**\n**";  
    **throw**;  
   }  
   cout<<"End of Function**\n**";  
   }  
     
   int **main**() {  
    cout<<"Inside main**\n**";  
    try {  
    divide(**10.5**,**2.0**);  
    divide(**20.0**,**0.0**);  
    }  
    **catch**(**double**) {  
    cout<<"Caught double inside main**\n**";  
    }  
    cout<<"End of main**\n**";  
   return **0**;  
   }