Object Oriented Programming Using C++

Year – Nov. 2019 **BCA–301**

|Time : Three Hours| **|Maximum Marks : 75|**

Note : Attempt all questions as per instructions.

This solution is provided by Mrs. Tushina Bedwal

Section – A

(Very Short Answer Questions)

Note : Attempt all Five questions. Each question carries 3 marks. Very short answer in required not exceeding 75 words. **[3 × 5 = 15]**

Q1. What is Destructors ? Give example.

Ans. Destructors in C++ :

A 'destructor', as the name implies, is used to destroy the objects that have been created by a constructor. Like a 'constructor', a 'destructor' is a member function whose name is the same as the class name but is preceded by a 'tilde (~)'.

For example: The 'destructor' for the class integer can be :

```
~integer( ){ }
```

A 'destructor' never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.

Implementation of Destructors :

```
#include<iostream.h>
int count = 0;
class alpha
{
  public :
  alpha( )
  { count++;
    cout<< "\nNo. of object created"<<count;
  }
  ~alpha( )
  {
    cout<<"\nNo. of object destroyed"<<count;
    count--;
  }
}
main( )
{ cout<<"\n\nEnter Main\n";
  alpha A1, A2, A3, A4;
  { cout<< "\n\nEnter Block1\n";
    alpha A5;
  }
  cout<<"\n\nRe-Enter Main\n";
}
```

Output :

```
Enter Main
No. of object created1
No. of object created2
No. of object created3
No. of object created4
Enter Block1
No. of object created5
No. of object destroyed5
Enter Block2
No. of object created5
No. of object destroyed5
Re-Enter Main
No. of object destroyed4
No. of object destroyed3
No. of object destroyed2
No. of object destroyed1
```

Q2. Give the significance of Protected access specifiers.

Ans. Significance of Protected Access Specifiers :

'**Protected**' access modifier is similar to that of '**private**' access modifiers, the difference is that the class member declared as '**Protected**' are inaccessible outside the class but they can be accessed by any subclass (derived class) of that class.

If 'protected' access specifiers is used while deriving class then the public and protected data members of the base class becomes the '**protected member**' of the derived class and private member

of the base class are inaccessible.

In this case, the members of the base class can be used only within the derived class as protected members except for the private members.

Following example explain how data members of base class are inherited when derived class access mode is protected :

Example :

```
#include <iostream>
using namespace std;
class base
{
   private :
      int x;
   protected:
      int y;
   public :
     int z;
   base() //constructor to initialize data members
   {       x = 1;
           y = 2;
           z = 3;
   }
};
class derive: protected base
{
   //y and z becomes protected members of class derive
   public :
     void showdata()
     { cout << "x is not accessible" << endl;
       cout << "value of y is " << y << endl;
       cout << "value of z is " << z << endl;
     }
};
int main()
{ derive a; //object of derived class
   a.showdata();
   //a.x = 1;  not valid : private member can't be accessed outside of class
   //a.y = 2;  not valid : y is now private member of derived class
   //a.z = 3;  not valid : z is also now a private member of derived class
     return 0;
}        //end of program
```

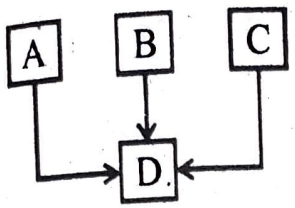
Output :

```
x is not accessible
value of y is 2
value of z is 3
```

Q3. How the ambiguity in multiple inheritance can be resolved ?

Ans. Multiple Inheritance :

A single derived class is derived from more than one base class.



Syntax :

```
class D : visibility A, visibility B ,visibility C
{
.....
..... //body of class D
.....
};
```

Ambiguity : Ambiguity arises when same function name in base classes exist :

Example :

```
class A
{
protected :
      int a;
public :
      void get_a(int ) { a=x; }
      void display() { cout<<a<<endl; }
};
class B
{
protected :
      int b;
public :
      void get_b(int ) { b=y; }
      void display()
      { cout<<b<<endl; }
};
class C : public A, public B
{
public :
      void show( );
};
void C :: show( )
{ display();
      // Ambiguity : which display() function
      should be called?
}
int main( )
{
      C c;
      c.get_a(10);
      c.get_b(20);
      c.show();
      return 0;
}
```

Resolution :

Ambiguity can be resolved by defining a named instance within the derived class using the class resolution operator with the function.

For example :

```
class C : public A, public B
{
public :
      void show( );
};
void C :: show( )
{
      A :: display();
}
```

Q4. What are 'Default arguments'.

Ans.

[Please Refer Q14 Unit-IV Page-89]

Q5. Explain the term 'Data hiding'.

Ans. Data Hiding :

Data hiding is an aspect of object-oriented programming (OOP) that allows developers to protect private data and hiding the implementation details.

Data hiding is the process of hiding the details of an object or function. It is also a potent technique in programming that results in data security and less data complexity.

Data hiding is used as a method to hide information inside a computer code after the code is broken down and hidden from the object. All objects in the state of data hiding are in isolated units which is the main concept of main object oriented programming.

Example :

```
#include<iostream.h>
```

```
using namespace std;
class abc
{
    int a
    public :
    void read( );
    void print( );
}
void abc : : read( )
{
    cout << "Enter any value";
    cin >> a ;
}
void abc : : print( )
{
    cout << "The value is" <<a<<endl ;
}
int main( )
{
    abc k;
    k.read( );
    k.print( );
    return 0;
}
```

Section – B

(Short Answer Questions)

Note : Attempt any Two questions out of the following Three questions. Each question carries 7½ marks. Short Answer is required not exceeding 200 words.

[7½ × 2 = 15]

Q6. What are 'Inline functions' ? How are they useful ?

Ans. Inline Functions :

Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

Need for 'Inline Functions' :

When the program executes the function call instruction, it involves following steps :

- Jumping to a function
- Saving the registers, the CPU stores the memory address of the instruction following the function call
- Pushing arguments in to the stack
- Returning to the calling function

For functions that are large, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code.

To reduce the function call overhead, C++ provides inline functions.

Points to remember :

- If a function is inline then, compiler puts its code at the place where it is called at compile time.
- To define a function as inline function, use the keyword "inline" just before the return type.
- All inline functions must be defined before they are called.
- The compiler ignore the inline qualifier in case defined function is too long.

Syntax :

```
inline function-header
{
    Function body
}
```

Example :

```
#include <iostream.h>
#include<conio.h>
int multiply(int);
int main( )
{
    int x;
    cout<< "\n Enter the Input Value:";
    cin>>x;
    cout<<"\n The Output is:" << multiply(x);
    getch();
}
inline int multiply(int x1)
{
    return 5*x1;
}
```

Q7. Explain : Overloading Vs. Overriding.

Ans.

Operator Overloading	Operator Overriding
Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.	Operator overriding is the process of redefining an operator (method) in a derived class with the same signature as a member in the base class but performing different task.
Operator overloading is used to overload or refines most of the operators available in C++.	Overriding occurs when you override the base implementation.
It is used to perform the operation on the user-defined data type.	It is used to achieve runtime polymorphism.

Syntax of Operator Overloading :
return_type class_name::operator op(argument_list)
{
// body of the function
}
where,
return_type → type of the value returned by the function.
class_name → name of the class
operator op → operator function where op is the operator being overloaded and the operator is the keyword.
	Example :
	class A {
	public :
	void display( ) {
	// overriding
	}
	};
	class B ; public A {
	public :
	void display( )
	{
	// overridden method in class A
	}
	};
	#display( ) in class A is overridden by class B

Q8. Explain the concept of 'Abstract classes' and 'Virtual base classes' with a suitable example.

Ans. Abstract Classes :

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class. It is a design concept in program development and provides a base upon which other class may be built.

Abstract class is a class which contains atleast one pure virtual function in it.

Abstract classes are used to provide interface for its sub classes. Classes inheriting an Abstract class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristic of Abstract Class :

1. It cannot be instantiated.
2. It can have normal functions & variable along with pure virtual functions.

Pure Virtual Functions : These are virtual functions with no definition. They start with virtual keywords & ends with 0.

e.g., virtual void f( ) = 0;

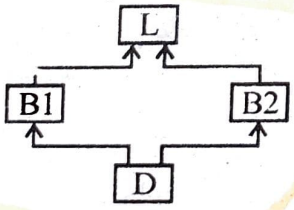
Example :

```
class Base
{
  public :
   virtual void show( ) = 0;
};
class Derived : public Base
{ public :
   void show( )
   {
       cout<<"abc";
   }
};
int main( )
{
  derived d;
  d.show( );
  return 0;
}
```

Virtual Base Class :

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as **Virtual** when it is being inherited. Such a base class is known as **Virtual Base Class.** This can be achieved by preceding the class name with "**Virtual**".

For Example :



Class L {/*.....*/}; //indirect base class

Class B1 : virtual public L{/*.....*/};

Class B2 : virtual public L{/*.....*/};

Class D : public B1, public B2 {/*.....*/}; //valid

Using the keyword **virtual** in this example ensures that an object of class D inherits only one sub object of class L.

Section – C

(Detailed Answer Questions)

Note : Attempt any Three questions out of the following Five questions. Each question carries 15 marks. Answer is required in detail. [15 × 3 = 45]

Q9. What do you mean by 'Exception handling'? How exceptions are handling is done in C++. Illustrate with example.

Ans. Exceptions :

These are run time anomalies or unusual conditions that a program may encounter during execution. These are problems in program other than logic errors and syntax errors. Following conditions are considered as exceptions :

- Division by zero
- Access to an array outside of its bounds
- Running out of memory

Exception Handling Mechanism :

It is a separate error handling code performing the following:

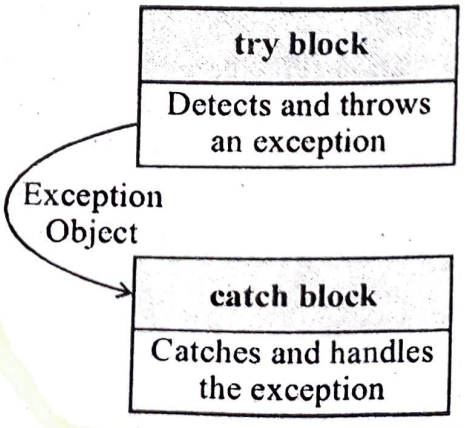
- Find the problem (Hit the exception)
- Inform that an error has occurred (Throw the exception)
- Receive the error information (Catch the exception)
- Take corrective action (handle the exception)

It is basically build upon three keywords :

- Try
- Throw
- Catch

The error handling code has two segments,

- One to detect error and throw exceptions and
- Other to catch the exceptions and to take appropriate actions.



Example of Exception Handling in C++ :

```
int main()
{
int a,b;
cout<<"enter the values of a and b";
cin>>a;
cin>>b;
int x = a - b;
try
{
    if(x!=0)
    {
      cout<<"result(a/x) ="<<a/x<<"\n";
    }

else
    {
      throw(x);
    }
}
    catch(int i)
    {
      cout<<"exception caught : x =" <<x<<"\n";
    }
    cout<<"end";
    return 0;
}
```

Output :

```
enter value of a and b
20 15
result(a/x)=4
end
Second run
Enter value of a and b
10 10
exception caught : x=0
end
```

Q10. In what ways object oriented paradigm is better than structured programming paradigm ? Explain the features of OOPs.

Ans. Object Oriented Programming :

Object Oriented Programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

Following Advantages of Object oriented paradigm explain the supporting of OOPs over Structured programming paradigm are :

- User can create new data type or users define data type by making class.
- Data can be hidden from outside world by using encapsulation.
- Code can be reuse by using inheritance
- Operators or functions can be overloaded by using polymorphism, so same functions or operators can be used for multitasking.
- Easily upgraded from small system to large system.
- Easy to partition the work in a project.
- New data and functions can be easily added.
- Message passing feature makes the communication easier.

Features of OOPs :

(a) Object : This is the basic unit of object oriented programming. ***That is both data and function that operate on data are bundled as a unit called as 'object'.***

(b) Class : When we define a class, we define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

(c) Abstraction : ***Data abstraction refers to, providing only essential information to the outside world and hiding their background details,*** i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

(d) Encapsulation : ***Encapsulation is placing the data and the functions that work on that data in the same place.*** While working with

procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

(e) **Inheritance :** One of the most useful aspects of object-oriented programming is '**code reusability**'. As the name suggests ***Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.*** This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

(f) **Polymorphism :** The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called '**polymorphism**'. Poly refers to many. That is ***a single function or an operator functioning in many ways different upon the usage is called polymorphism.***

(g) **Overloading :** The concept of '**overloading**' is also a branch of polymorphism. ***When the exiting operator or function is made to operate on new data type, it is said to be overloaded.***

Q11. What do you mean by Polymorphism ? Explain with the help of example how polymorphism is achieved at (i) compile time (ii) run time.

Ans. **Polymorphism :**

The term '**polymorphism**' is made up of two words – **poly** and **morphism**, were 'poly' means 'many' and 'morphism' means 'forms'. So collectively the term means many forms or the ability to take many forms. ***When operators and methods are used in different ways, depending on what they are operating on, it is called Polymorphism (one thing with several distinct forms).***

[For More Information Please Refer Q9 Unit-III Page-79]

Q12. Explain :
(i) Constructors
(ii) Inheritance
(iii) Aggregation

Ans. (i) Constructors :

[Please Refer Q28 Unit-II Page-58]

(ii) Inheritance :

[Please Refer Q1 Unit-III Page-61]

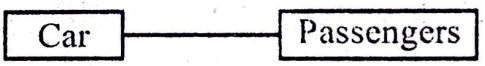
(iii) Aggregation :

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class.

In aggregation, objects have their own life cycle but there is ownership and child object cannot belongs to another parent object. But this is only an ownership not the life-cycle control of child control through parent object.

Example 1 : Departments has professors, if department is closed, professors still live.

Example 2 :



Cars may have passengers, they come and go.

Q13. What is Pointer Variable ? What are the applications of Pointer variable ? What are its advantages and disadvantages? What operations can be performed on the pointer variables ? What are basic data and derived data types which can be expressed in pointer variables ?

Ans. Pointer Variable Pointer in C++ :

A 'pointer' is a variable that stores a memory address. Pointers are used to store the addresses of other variable or memory items. Pointers are very useful for another type of parameter passing, usually referred to as '**Pass By Address**'. Pointers are essential for dynamic memory allocation.

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any constant or variable, we must declare a pointer before using it

to store any variable address. ***The general form of a pointer variable declaration is :***

type*var_name;

Here, '**type**' is the pointer's base type; it must be a valid C data type and var_name is the name of the pointer variable.

The '**asterisk ***' used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is bring used to designate a variable as a pointer. Some valid pointer declarations:

```
int *ip;        /* pointer to an integer */
double *dp;     /* pointer to a double */
float *fp;      /* pointer to a float */
char *ch;       /* pointer to a character */
```

We can create a special variable that stores the address rather than the value. This variable is called 'Pointer Variable' or simply a 'Pointer'.

Pointer variable is normal variable is used to store value. A pointer variable is used to store address / reference of another variable.

Syntax for declaring Pointer variable:

Data-type *pointer-name;

Examples for declaring Pointer variable:

```
int A=10;       //  statement 1
int *ptr;       //  statement 2
ptr = &A;       //  statement 3
```

Applications of Pointer Variable :

There are many usage of pointers in C++ language.

1. Dynamic memory allocation : In C language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2. Arrays, Functions and Structures : Pointers in C language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Like other variables, pointer variables can be used in arithmetic expressions. We can use the pointer variable to add, subtract, multiply and divide operations. For example, if p1 and p2 are properly declared and initialized pointers, then they can be used as -

```
y=*p1 + *p2; y= * p1 + 5;
y=*p1-*p2; y=*p2 - 5;
y=*p1 * *p2; y=*p2 *3;
y=*p1/*p2; y=*p1 / 2;
```

In division operation, there should be a blank space between/and*, otherwise it will be considered as comment and will not be executed.

In above statements the values at address p1 & p2 are being operated.

We can not perform addition, subtraction, multiplication and division. For example.

```
y = p1 + p2 ;
y = p1 * p2 ;
y = p1 - p2 ;
y = p1 / p2 ;
```

are not allowed. Only p1-p2 is allowed only when both are pointers to the elements of an array. It will give the number of elements between p1 and p2.

Pointers can also be compared using the relational operators. For example, the expression such as,

p1 > p2, p1 = =p2 , p1 ! p2

are allowed.

Pointers can also be used with addition or subtraction with integers. For example

```
p1 = p2+2;  p1 = p2-2;
p1=p1+1; p1=p1-1;
```

The statement

p1 ++ ; or P1 = p1 +1 ;

will cause the pointer p1 to point to the next value of its type. For example, if p1 is an integer pointer with an initial value, say 2000, then after the operation p1 = p1+1, the value of p1 will be 2002, and not 2001. That is, when we increment a pointer, its value is increased by 'length' of the data type that it points to.

Advantage of Pointer :

1. Pointers provide direct access to memory.
2. Pointers are more efficient in handling arrays and data tables.
3. Pointers can be used to return multiple values from a function.
4. Pointers increase the execution speed and thus reduce the program execution time.
5. Pointers permit references to functions and thus allow passing functions as arguments to other functions.

6. Pointers allow C to support dynamic memory management. Pointers provide a way to perform dynamic memory allocation and deallocation.
7. Pointers provide an efficient tool for manipulating dynamic data structures such as structure, union, linked list etc.
8. Using pointer arrays to store character strings, saves data storage space in memory.
9. Pointers reduce length and complexity of programs.

Disadvantages of Pointer :

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Arithmetic operations can be carried out on pointers :

- Increment operator (++)
- Decrement operator (--)
- Addition (+)
- Subtraction (-)

The usage of these operations in an Example program :

```
#include <iostream>
#include <string>
using namespace std;
int main()
 {
  int myarray[5] = {2, 4,6, 8,10};
  int* myptr;
  myptr = myarray;
  cout<<"First element in the array :"<<*myptr<<endl;
  myptr ++;
  cout<<"next element in the array :"<<*myptr<<endl;
  myptr +=1;
  cout<<"next element in the array :"<<*myptr<<endl;
  myptr--;
  cout<<"next element in the array :"<<*myptr<<endl;
  myptr -=1;
  cout<<"next element in the array :"<<*myptr<<endl;
  return 0;
 }
```

Output :

```
First element in the array : 2
next element in the array : 4
next element in the array : 6
next element in the array : 4
next element in the array : 2
```

Pointer variables can be created for all basic data types (int, float, char, string) and derived data types (functions, arrays, pointers).