University of Sargodha

BS 2nd Term Exam 2015

Subject: Computer Science **Paper: Object Oriented Programming (CMP-2124)**

Time Allowed: 2:30 Hours **Maximum Marks: 80**

Objective Part Compulsory

Q.No.1: Attempt all parts and each require answer 2 – 3 lines (16*2=32)

1. Stands for "**Object-Oriented Programming**" OOP (not Oops!) refers to a programming methodology based on objects, instead of just functions and procedures. These objects are organized into classes, which allow individual objects to be group together.

2. Encapsulation: Encapsulation sometimes referred to as data hiding, a process of language mechanism for restricting direct access to some of object's component. The idea is "don't tell me how you do it, just do it!"

3. Abstract Class: A class which cannot be initiated but can be used as base class in inheritance. No object of an abstract class can be created.

4. Syntax of Class declaration: A class definition starts with the keyword **class** followed by the class name. The class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. Syntax:

```
class [class name]
{
access specifier:
        statement(s);
};
```

Example: A class name "box".

```
class Box{
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box };
```

6. Destructor functions are the inverse of constructor functions. They are called when objects are destroyed. Same name as class name with symbol (~) called tilde. It has no return type and there is only one destructor in one class.

7. Function Overriding: Function Overriding also allows to have two or more function with the same name but in this case, it is important to have Inheritance. When a function is already defined in the base class and we redefine it in derived class, that function is said to be overridden. In other words, if an inherited class contains a function with same name as that of its parent, then this concept is known as Function Overriding.

Function Overloading: Function overloading allows us to define two or more function having the same name but with different function body. It is useful in real world applications because practically, we encounter such situations very commonly.

8. The process of creating new class from existed class is called **inheritance**. New classes retain characteristics of the existed class. A class from which new class is inherited is called Base Class. Derived class is called Child class.

Parent Class=Base Class=Super Class

Child Class=Derived Class

9. A **friend function** of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition,

friends are not member functions. A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

10. In OOPs **Dynamic Binding** refers to linking a procedure call to the code that will be executed only at run time. The code associated with the procedure in not known until the program is executed, which is also known as late binding. C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time; this is called dynamic binding or late binding or run-time binding. Dynamic binding is achieved using virtual functions. Base class pointer points to derived class object. And a function is declared virtual in base class, then the matching function is identified at run-time using virtual table entry.

11. Purpose of try and catch block: Try block tries to solve the error but if error remains then throw block take that error and give it to the catch block. Catch block shows or display that error to the user.

12. Object: Object is an instance (variable) of class. "Object" refers to a particular instance of a class where the object can be a combination of variables, functions, and data structures.

Long Questions

Q2. Write a class car that contains attributes of car's name (type string), top speed (type float) and model name (type string). A constructor with no arguments initializes the data members and two-member functions to input and display these attributes.

```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class car{
   private:
         string name, model;
         float top_speed;
   public:
         car(){
               name=model="NULL";
               top_speed=0;
         }
         void input(){
               cout<<"Enter Car Name:";
               cin>>name;
               cout<<"Enter Car Model:";
               cin>>model;
               cout<<"Enter Top Speed of Car:";
               cin>>top_speed;
         }
         void display(){
               cout<<"Name of car:"<<name<<endl<<"Model of Car:"<<model<<endl;
               cout<<"Top Speed:"<<top_speed;
         }
};
int main(){
   car c1;
   c1.input();
   c1.display();
   getch();
   return 0;
}
```

Q6. Describe the function template and class template. Discuss with the help of an example.

C++ has a mechanism called templates to reduce code duplication when supporting numerous data types. Templates allow programmer to create a common class or function that can be used for a variety of data types. The main advantage of using a template is the reuse of same algorithm for various data types, hence saving time from writing similar codes. Templates in C++ can be divided into major two types, they are:

- **Function Template**
- **Class Template**

Function Template

A generic function that represents several functions performing same task but on different data types is called function template. **For example**, a function to add two integer and float numbers requires two functions. One function accept integer types and the other accept float types as parameters even though the functionality is the same. Using a function template, a single function can be used to perform both additions. It avoids unnecessary repetition of code for doing same task on various data types.

Syntax of Function Template:

```
template < class T1, class T2, ... >
returntype function_name (arguments of type T1, T2, ...)
{
  statement(s);
  ... ... ...
}
```

Example of Function Template

C++ program to add two numbers using function template.

```
#include <iostream>
#include <conio.h>
using namespace std;

template<class t1,class t2>
void sum(t1 a,t2 b) // defining template function
{
    cout<<"Sum="<<a+b<<endl;
}

int main()
{
    int a,b;
    float x,y;
    cout<<"Enter two integer data: ";
    cin>>a>>b;
    cout<<"Enter two float data: ";
    cin>>x>>y;
    sum(a,b); // adding two integer type data
    sum(x,y); // adding two float type data
    sum(a,x); // adding a float and integer type data
    getch();
    return 0;
}
```

This program illustrates the use of template function in C++. A template function sum() is created which accepts two arguments and add them. The type of argument is not defined until the function is called. This single function is used to add two data of integer type, float type and, integer and float type. We don't need to write separate functions for different data types. In this way, a single function can be used to process data of various type using function template.

Output

```
Enter two integer data: 6 10
Enter two float data: 5.8 3.3
Sum=16
Sum=9.1
Sum=11.8
```

Class Template

Like function template, a class template is a common class that can represent various similar classes operating on data of different types. Once a class template is defined, we can create an object of that class using a specific basic or user-defined data types to replace the generic data types used during class definition.

Syntax of Class Template

```
template <class T1, class T2, ...>
class classname
{
    attributes;
    methods;
};
```

Example of Class Template

C++ program to use class template

```
#include <iostream>
#include <conio.h>
using namespace std;

template<class t1,class t2>
class sample {
    t1 a;
    t2 b;
    public:
        void getdata()  {
            cout<<"Enter a and b: ";
            cin>>a>>b;
        }
        void display(){
            cout<<"Displaying values"<<endl;
            cout<<"a="<<a<<endl;
            cout<<"b="<<b<<endl;
        }
};
int main(){
    sample<int,int> s1;
    sample<int,char> s2;
    sample<int,float> s3;
    cout<<"Two Integer data"<<endl;
    s1.getdata();
    s1.display();
    cout<<"Integer and Character data"<<endl;
    s2.getdata();
    s2.display();
    cout<<"Integer and Float data"<<endl;
    s3.getdata();
    s3.display();
    getch();
```

```
    return 0;
}
```

In this program, a template class sample is created. It has two data a and b of generic types and two methods: getdata() to give input and display() to display data. Three object s1, s2 and s3 of this class is created. s1 operates on both integer data, s2 operates on one integer and another character data and s3 operates on one integer and another float data. Since, sample is a template class, it supports various data types.

Output

```
Two Integer data
Enter a and b: 7 11
Displaying values
a=7
b=11
Integer and Character data
Enter a and b: 4 v
Displaying values
a=4
b=v
Integer and Float data
Enter a and b: 14 19.67
Displaying values
a=14
b=19.67
```

Q7. Explain the following with example.

I- Polymorphism

The term Polymorphism get derived from the Greek word where *poly* + *morphos* where poly means many and morphos means forms. The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, an employee. So, a same person possesses have different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object-Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

❖ Compile time polymorphism:

This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

```
// C++ program for function overloading
#include <iostream>
using namespace std;
class Geeks {
    public:
    void func(int x) {              // function with 1 int parameter
        cout << "value of x is " << x << endl;
    }
    void func(double x){            // function with same name but 1 double parameter
        cout << "value of x is " << x << endl;
    }
    void func(int x, int y) {        // function with same name and 2 int parameters
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};
```

```
int main() {
    Geeks obj1;       // Which function is called will depend on the parameters passed
    obj1.func(7);         // The first 'func' is called
    obj1.func(9.132);     // The second 'func' is called
    obj1.func(85,64);     // The third 'func' is called
    return 0;
}
```

Operator Overloading: C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add to operands. So, a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.

```
#include<iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0) {
       real = r;   imag = i;
       }
    // This is automatically called when '+' is used with between two Complex objects
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
    }
    void print() {
       cout << real << " + i" << imag << endl;
       }
};
int main(){
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

❖ Runtime Polymorphism:

This type of polymorphism is achieved by **Function Overriding**.

Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

```
// C++ program for function overriding
#include <iostream>
using namespace std;
class Parent {       // Base class
    public:
    void print()
    {
        cout << "The Parent print function was called" << endl;
    }
};
class Child : public Parent  {      // Derived class
    public:
    void print()                    // definition of a member function already present in Parent
    {
        cout << "The child print function was called" << endl;
    }
};

int main()
{
    Parent obj1;             //object of parent class
    Child obj2 = Child();      //object of child class
    obj1.print();            // obj1 will call the print function in Parent
    // obj2 will override the print function in Parent and call the print function in Child
    obj2.print();
```

```
    return 0;
}
```

II- Dynamic Memory Management

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack.

- **The stack** – All variables declared inside the function will take up memory from the stack.
- **The heap** – This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator. If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by **new** operator. Following example is to show how 'new' and 'delete' work –

```
#include <iostream>
#include <conio.h>
using namespace std;

int main () {
   double* pvalue  = NULL;             // Pointer initialized with null
   pvalue  = new double;               // Request memory for the variable
   *pvalue = 29494.99;                 // Store value at allocated address
   cout<<"Value of pvalue : "<< *pvalue << endl;
   delete pvalue;                      // free up the memory.
   Getch();
   return 0;
}
```