PAGE NO:
DATE: / /

College Roll. No: 191725
Level: Bachelors
Programme: Software.
Semester: 2nd
Subject: OOP in C++.
Signature of the Examinee/Student: Date: 16/03/2078

Saroj Dahal - 191725

PAGE NO.:
DATE: / /

Q.N.1.

A. Ans.

- An object oriented programming language is one of the computer languages which make use of object as a real-world instance to simulate the system.

- In object oriented programming, the clarity in coding can be made as same as of real-world by making use of different OOP concepts such as Polymorphism, Inheritance, Classes and many more.

- Despite many features of OOP, the common feature that makes real-world modeling easy is because of agents, methods, behaviors, and their responsibilites. Mostly, all these stuffs are defined inside the class; and the class's instance i.e. Object plays the role.

- For example:

  Let us take a student that studies in a certain University. S/he might have different responsibites, behavior, or may be permission associated with themself and with the university.

  So, In OOP we model student as a class and we define their data (state) as data members, their action (responsibility) as methods. And, to play around with Student class, we make an instance of that class, which just clones the class funcionalities but has its own separate memory space. We call that instance as Object.

As, In same regard, in order to model another student, Now we can simply create another instance of the class. In our example, student might have:

setStudentInfo()
payMonthlyFee() methods.

and, may have data such as:

name,
rollNumber
registrationNumber.
emailAddress and soon.

For the agents, there might be,

Teacher class.
Department (class).
Block (classes).

In this way, Real-world phenomenon can be easily modeled by using object oriented concepts.

B. Ans.

Static data members:

→ Static data members are special type of class members, whose value is reused and shared across the object without creating extra memory location for it.

→ They are defined by adding static keyword infront of normal variable declaration.

PAGE NO:
DATE: / /

Ex: static int a = 10;

→ They do have the lifetime for the entire program but they are scoped (limited) within the class only. and, they comes handy and into use, when we want to make update to the same variable, the number of times.

→ Syntax to define static data member is:

static datatype ~~static~~_variable_name;

Ex: static float marks;

Static functions:

→ Static functions are special type of function which holds same behavior as static member but ~~acts~~ as a function for making use of static members and other calls without depending on particular object.

→. Yes, its true, static function doesn't require an object and dot (.) operator to call it. It can be simply called by using Classname followed by scope resolution operator and static function name.

Ex: student :: dosomething ();

Here, student is the class name.

dosomething() is static function.

PAGE NO.:
DATE: / /

Syntax to declare static function:

It is defined as same as normal function but we just have to add static keyword before function return type to declare it.

```
static return_type functionName.{

    .....

    ....

}
```

Ex:

```
static void counter() {
    count ++;
}
```

Example of using static data member:

```
#include <iostream.h>

class A {
    private:
        static int count;

    public:
        void displayCount()
        {
            cout << " count :" << count;
        }
```

```
    void increaseCount() {
        count++;
    }
};

int A::count = 1;

void main() {
    A obj1, obj2;

    obj1.displayCount();      // prints 1
    obj1.increaseCount();     // count increase to 2.
    obj2.displayCount();      // prints 2
    obj2.increaseCount();     // count increases to 3
    obj1.displayCount();      // prints 3.

}
```

From the above example what we can see is, although we have use obj1 to increase count value, the another obj2 gets the increament value. Here the count variable value is shared among these two objects and changes made in one can be seen from another object.

Q.N.2.

A. Ans.

No, it is not directly accessible outside the class. But, we can make use of certain technique to access it. In simplest way, we can make getters in order to access it.

for Example: (In normal case)

```
#include <iostream.h>

class Student {
    private:
        int roll;

    public:
        ....
}
```

Here, Student obj;
obj.roll; // this doesn't work, as roll is under private access specifier.

Another Example: (To access private member).

```
#include <iostream.h>

class Student {
    private:
        int roll;

    public:
```

```
int getRoll()      // created public method to ~~access~~
{                     return private data.
    return roll;
}
};

void main()
{
    Student obj;
    int r = obj.getRoll(); //sets 'r' variable with roll.
}
```

In this way, we can access private data member of class.

B. Ans.

Constructor is a special member function of the class which is responsible for initializing values to members of class, when the object associated with the class is created.

Yes, a class can have multiple constructor. This is made possible by varying number and type of parameter signatures in constructor.

Program:

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

```
class Adder {
    public:
        char[20] fname, mname, lname;
        char[100] fullName;

        Adder()  // Default constructor (constructor 1)
        {

        }

        Adder (char[20] fname, char[20] mname, char[20] lname)
        {            // Constructor second.
        strcpy(this.fname, fname);
        strcpy(this.mname, mname);
        strcpy(this.lname, lname);
        }

        void  concatName(){
            strcat (fullName, fname);
            strcat (fullName, mname);
            strcat (fullName, lname);
            cout << "Full name : " << fullName;
        }

};

void main()
{
    Adder obj ("Saroj", "Ram", "Shrestha");
    obj.concatName();
}   getch();
```

Saroj Dahal - 191725

Q.N.3.

A. Ans.

Yes, composition provide re-usability as like inheritance. With inheritance, we can create classes that derive their attributes from existing classes so while using inheritance to create a class, we can expand on an existing class.

On contrary, using an object within another object is known as composition. It is an alternative to class inheritance.

Example program:

```
#include <iostream.h>
#include <conio.h>
class B;
class A {
  public:
      void functionA() {
          cout << "A function";
      }
};

class B {
  public: class A; // composition.
      void functionB() {
          cout << " B function";
      }
      void callFunction() {
          A.functionA();
          functionB(); }
```

```
};
void main()
{
    B objB;
    objB.allFunction();  // Output : A function
                                     B function.
    getch();
}
```

second part:

IS-A relationship.	Has-A relationship.
• "is-a" relationship is a relation between two classes where one class is a specialized form of second class.	• If one concept is a component of another concept then there exists "has-a" relationship. It is similar to a class having certain members.
• This relationship ~~can~~ is represented by inheritance.	• It cannot be represented by inheritance.
• Example satisfying "is-a" relationship: Ram is a person. car is a vechile.	• Example satisfying "has-a" relationship. student has a Books. College has a teacher

B. Ans.

Dynamic Memory Allocation (DMA):

It is the process of allocating memory space dynamically as per the requirement of the program or by the user.

→ The memory which are allocated dynamically are stored in heap.

PAGE NO:
DATE: / /

→ For dynamically allocated memory, we must be careful in its use and make sure it is destroyed after its used else it will drastically decrease app performance.

→ Memory allocated dynamically are not deleted even if exceeds the program scope, so the programmer have to make sure and keep an eye for memory optimization when allocating memory dynamically.

→ We can allocate and de-allocate memory dynamically using new and delete operator respectively.

Second part:

```
//Dynamic memory allocation for object.

#include<iostream.h>
#include<conio.h>

class Student{
        char name[50];
    public:
        void setName(){
            cout<<" Enter student Name:";
            cin>>name;
        }

        void displayName(){
            cout<<" Student Name:"<<name;
        }
};
```

PAGE NO.:
DATE: / /

```
void main(){
    student *ptr = new student; // declaring object
                                   dynamically.
    ptr -> setName();
    ptr -> displayName();

    delete ptr;
    getch();
}
```

Q.N.4.

A. Ans.

The different forms of inheritance are described below:

a. Inheritance of specialization:

- Child class is specialized form of parent class.
- Child class satisfies the specification of parent in all relavant aspects.

b. Inheritance for specification:

- The parent class is just used to provide behavior or which function should be available but doesn't provide its implementation.

c. Inheritance for construction:

- If the parent class is used as a source for the behavior but the child has no is-a relationship to parent, the the child is being using inheritance for construction.

PAGE NO.:
DATE: / /

d. Inheritance for generalization:
- Subclass extends the behavior of superclass to create more general kind of object but doesn't override any method with completly new features.

e. Inheritance for existence:
- Extension simply adds new methods in the child to those of the parent.

f. Inheritance for limitation:
- It is used when the behavior of subclass is smaller or more restrictive than the behavior of parent class.

g. Inheritance for variance:
- It is used when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between concepts represented by the classes.

h. Inheritance for combination:
- The subclass represents a combination of features from two or more parent class.

PAGE NO.:
DATE: / /

B. Ans

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class student {
    public:
        char[50] name, roll, registrationNumber;

        Student() {}.
        Student (char[50] name, char[50] roll, char[50] r){
            strcpy (this.name, name);
            strcpy (this.roll, roll);
            strcpy (this.registrationNumber, r);
        }

                    student
        void displayInfo (){
            cout<<"Name :"<<name<<endl;
            cout <<"Roll :"<<roll<<endl;
            cout << "Registration Number :"<<registrationNumber;
        }
};

Class TheoryMarks : public student {
    public:
        int obtainedMarks;
        TheoryMarks () {}
        TheoryMarks (int marks){
            obtainedMarks = marks;
        }
```

PAGE NO.:
DATE: / /

```
    void displayObtainedThMarks(){
        cout << "Theory Marks :" << obtainedMarks;
    }
};

class PraticalMarks : public Student {
    public:
        int obtainedMarks;
        PraticalMarks(){}.
        PracticalMarks (int marks){
            obtainedMarks = marks;
        }

        void displayObtainedPrMarks(){
            cout << " Practical Marks :" << obtainedMarks;
        }
};

class PerformanceMarks : public Student {
    public:
        int obtainedMarks;
        PerformanceMarks(){}
        PerformanceMarks (int marks){
            obtainedMarks = marks;
        }

        void displayObtainedPfMarks(){
            cout << " Performance Marks :" << obtained
                                              Marks;
        }
};
```

PAGE NO.:
DATE: / /

```
class Result : public TheoryMarks, public PraticalMarks, public
                PerfomanceMarks.
{
    public:
        ~~void setStudentInfo (Student s)~~
        Result () {
            cout << "Result object created";
        }
};

void main () {
    Result result;
    Student student ("saroj", "1", "191725");
    TheoryMarks thObject (95);
    PraticalMarks prObject (100);
    PerfomanceMarks pfObject (100);

    result.displayStudentInfo();
    result.displayObtainedThMarks();
    result.displayObtainedPrMarks();
    result.displayObtainedPfMarks();
    getch();
}
```

PAGE NO.:
DATE: / /

Q.N.5.

A. Ans.

this pointer:

→ this pointer holds the address of current calling object when calling a member function.

objectA . function();

Here this pointer holds the address of object A.

→ For non static member function, this pointer is passed as hidden argument implicitly by the compiler.

→ In simple words, this pointer is use to refer the current using state; in order to access that current type's members and methods.

Example program:

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

class Complex {
    int real, img;
    public:
        void getAddress()
        {
            cout << "Address :" << this;
        }
};
```

```
complex (int real, int img){
    this->real = real;
    this->img = img;
}
```

```
void main()
{
    Complex c1,c2;
    c1.getAddress(); // 0xFFCC123
    c2.getAddress(); // 0xFFCC124.

    Complex c3(1,2);
}
```

Second part:

```
// overloading + and - operator.

#include <iostream.h>
#include <conio.h>

class Distance {
    private:
        int km, meter;
    public:
        Distance(){}
        Distance (int km, int meter) {
            this->km = km;
            this->meter = meter;
        }

        void operator + (int value){
            this->km+ =value;
            this-> meter+ =value;
        }
```

```
void operator - (int value) {
        this -> km - = value;
        this -> meter - = value;
    }

    void display () {
        cout << "Kilometer : " << km << endl;
        cout << "Meter : " << meter << endl;
    }
};

void main()
{
    Distance d1(1,2), d2(2,3);
    d1.operator+(5);
    d1.display();

    d2.operator-(1);
    d2.display();
    getch();
}
```

Output:

```
Kilometer : 6
Meter : 7
Kilometer : 1
Meter : 2.
```

PAGE NO.
DATE / /

5. B. Ans.

In this case, ambiguity occurs. i.e. the compiler gets confused while calling the function. Although the same function gets overridden but calling will be confusion. In our case, if it is pointing to child class object it will call child class function. But if we want to call function of parent class then we should define virtual function in base class, from which we can specifically call the function of parent class and child class without requiring type of reference.

Second part:

Virtual functions are used when we need to ensure that the current function is called for an object, regardless of the type of reference (or pointer) used for function call.

Q.N. 6.

A. Ans. Exception Handling:

→ Exception Handling is a mechanism which is used for detecting, reporting and handling the unexpected exceptions (errors) of our programs.

→ From exception handling. we can easily catch out any possible errors that can occur and redirect our program to run smoothly inspite of errors.

PAGE NO.:
DATE: / /

→ Exception handling is very useful technique to write the program and to handle it.

→ Generally, Exception handling code consits of two segments:
1. One segment to detect errors and throw exceptions.
2. The other segment to catch the exception and take appropriate actions.

Implementation Technique.

```
....
try
  {
    ......
    throw exception;
    ....
  }
  catch (type argument)
  {
      // block statement that handle exceptions..
  }
```

Example program:
// Program to handle division by zero.

```
#include <iostream>
void main(){
    char error[] = " Error occured. Division by zero";
    float num1, num2, result;
```

```
cout << "Enter two numbers:";
cin>> num1>>num2;

try {
    if (num2 != 0)
    {
        result = num1/num2;
        cout << "The division result is" << result << endl;
    }
    else
        throw error;
}

catch ( char e[] )
{
    cout << "Divisor should not be zero." << endl << e;
}

} //end of program.
```

6. B. Ans.

CRC card stands for Class Responsibility Collaborator card which is one of the system design technique in which a class has to be identified, list of responsibility has to be identified and collaboration has to be listed.

It consists of three sections:
Class, Responsibility and Collaborator.

PAGE NO:
DATE: / /

- Class : a class represents a collection of similar objects.
- Responsibility : It is something that a class knows or does.
- Collaborator : It is simply another class that interacts with that class to fulfill its responsibilites.

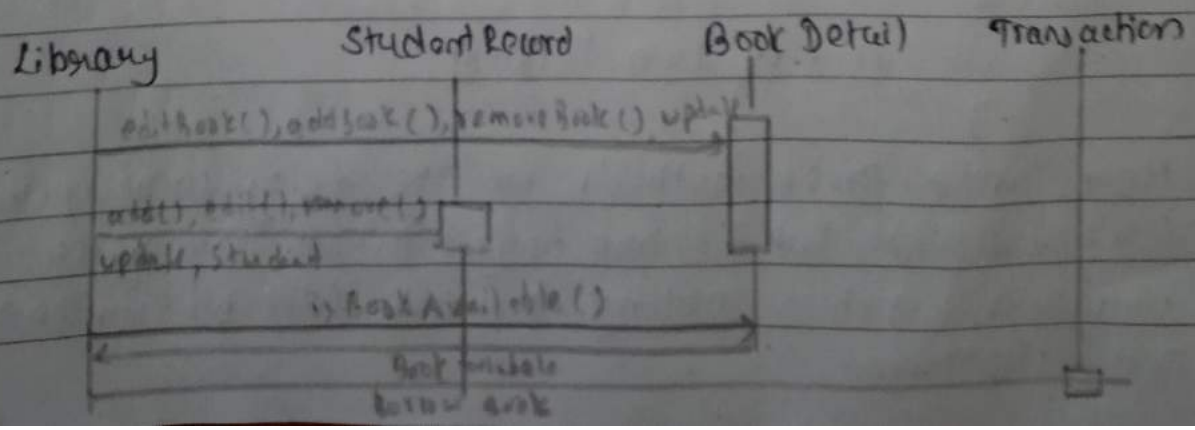
CRC card :

Component.	
Responsibilites	Collaborators

Example :
// CRC card for student.

Student.	
Responsibilities	Collaborators.
name	Exam Department.
roll	Library.
takeExam.	Account Department
request for transript	

Sequence diagram for library Management system.



DATE: / /

Q.N. 7

A. Template and generic programming:

In programming relam templates and generic programming is very crucial part. It not only simplifies the programming technique but also provides a better way of code writing.

Lets take a simple case. Lets say, I want to add two numbers. For this, I can create function in this way:

```
void add ( int a, int b)
{
    cout << "sum :" << (a+b);
}
```

But this function supports only for integer value but for floating point or decimal value support we have to overload the function as shown below:

```
void add (float a, float b)
{
    cout << "sum :" << (a+b);
}
```

OR,

```
void add (double a, double b)
{
    cout << "sum :" << (a+b);
}
```

Here, inside three methods we are doing same operation (i.e. addition) but we are using different parameter signature, which is making our code little bit unclean and unoptimized.

So, In this case, template turns out to be handy for us. In template we can generalize datatype to our method and do the operation as per our required datatype.

```
template <class t1, class t2>
void add(t1 a, t2 b) // defining function template
{
    cout << "sum : " << a+b;
}

int main()
{
    int a, b;
    float x, y; cout << "Enter two integer data:"; cin>>a>>b;
    add(a,b);
    ~~add f~~ cout << " Enter two floating data:";
        cin>>x>>y;
    add(x,y);
    return 0;
}.
```

Now, we can pass any data-type to our function, which makes it generic function, which is accomplished by using templates.

c. Function overloading:

Function overloading is the process of defining same function with same name but with different parameter signatures.

→ Function overloading falls under the concept of polymorphism. (which means, one name - many forms).

Example program to find maximum of 3 numbers using the concept of function overloading.

```
#include <iostream.h>

int max (int a, int b)
{
    return (a>b) ? a : b;
}

int max (int a, int b, int c)
{
    if (a>b && a>c) return a;
    else if (b>a && b>c) return b;
    return c;
}

int main()
{
    cout << "Max is :" << max(2,3);
    cout << "max is :" << max (2,3,6);
    return 1;
}
```

Output : Max is : 3
Max is : 6