Final Term Report

Introduction To Programming Language

Name: Nafincur Leo

Id : 20-42195-1

*Object Oriented Programming:

OOP is a paradigm that provides many concepts such as inheritance, data binding, polymorphism etc. The programming paradigm where everything is represented as an object is known as truly object-oriented programming language.

Example: Let's say, we have a class Car which

has data members such as speed, weight, price and functions such as gearChange(), slowDown(), brake() etc.

* Class And Object:

A class in C++ is the building block, that leads to OOP. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instant of that class. An object is an instance of a class. A C++ class is like a blueprint for an object.

Example:

* Class:

```
private:
    int x;
public:
    int getX(){
    return x;}
    void setX(int value){
        x= value;}
```

* Object:

```
private:
    int x;
public:
    int getX(){
    return x;}
    void setX(int value){
    x= value;}
```

* Properties, Methods:

Methods are functions attached to specific classes in OOP. Properties are an object-oriented idiom. The term describes a one or two functions - a 'getter' that retrieves a value

and a 'setter' that sets a value.

Example:

Properties:

```
private:
  int x;

public:
  int getX() {
  return x;}

void setX(int value){
  x= value;}
```

Class method:

```
class MyClass{ // The class

public: // Access specifier

void myMethod() { // Method/Function defined the class

cout << "Hello World!";}
};

int main(){

MyClass myObj; // Create an object of MyClass
myObj.myMethod(); // Call the method

return 0;}
```

* Access Specifiers:

The access restriction to the class members is specified by the labeled public, private and protected sections within the class body. The keywords public, private and protected are called access specifiers. A class can have multiple public, protected or private labeled sections.

Example:

Access Specifiers:

```
class MyClass{
  public: // Public access specifier
    int x; // Public attribute
```

```
private: // Private access specifier
int y; // Private attribute };
int main(){
  MyClass myObj;
  myObj.x=25; // Allowed (public)
  myObj.y=50; // Not allowed (private)
  return 0;}
```

* Constructor:

A constructor is a member function of a class which initializes objects of a class. In C++, constructor is automatically called when object create. It is special member function of the class. The purpose of constructor is to initialize

the object of a class while the purpose of method is to perform a task by executing java code.

Example:

Constructor:

```
class MyClass { // The class
public:     // Access specifier
MyClass () { // Constructor
cout << "Hello World!"; }
};
int main() {
MyClass myObj; // Create an object of myClass
                (this will call with constructor)
return 0; }
```

* Destructor:

A destructor is a member function that is invoked automatically when the object goes out of scope or is explicitly destroyed by a call to delete. A destructor has the same name as the class.

Example:

Destructor:

```
#include <iostream>
using namespace std;
class HelloWorld {
public:
// Constructor
HelloWorld() {
```

```
cout<<" Constructor is called"<<endl;}

// Destructor
~Hellow world (){
 cout<<" Destructor is called "<<endl;}

// Member function
 void display (){
   cout <<"Hello world!" <<endl;}
 };

 int main (){
  // Object created
     HelloWorld obj;

 // Member function called
  obj. display ();
  return 0;}
```

* Static Class Member:

We declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present

Example:

Static Class Member:

Live Demo

```
#include <iostream>
using namespace std;
```

```
class Box{
public:
  static int objectCount;
// Constructor definition
Box(double l=2.0, double b=2.0, double h=2.0){
cout << "Constructor called." <<endl;
length = l;
breadth = b;
height = h;

// Increase every time object is created
objectCount++;}
double Volume(){
return length * breadth * height;}
```

```
private:
  double length;  // Length of a box
  double breadth;  // Breadth of a box
  double height;  // Height of a box
};

// Initialize static member of class Box
int Box::objectCount=0;
int main(void){
  Box Box1(3.3, 1.2, 1.5);  // Declare box1
  Box Box2(8.5, 6.0, 2.0);  // Declare box2
  // Print total number of objects.
  cout<< "Total objects:" << Box::objectCount<<endl;
  return 0;}
```

* Operator Overloading:

Operator overloading means C++ has the ability to provide the operators with a special meaning for a data type. This ability is knows as operator overloading.

Example;

Operator Overloading:

```
#include <iostream>
using namespace std;

class Box{
public:
  double getVolume(void){
return length*breadth*height;}
```

```
void setLength(double len){
  length = len;}
void setBreadth(double bre){
  breadth = bre;}
void setHeight(double hei){
  height = hei;}

// Overload '+' operator to add two Box objects.

Box operator+(const Box& b){
   Box box;
  box.length = this->length+b.length;
  box.breadth = this->breadth+b.breadth;
  box.height = this->height+b.height;
  return box;}
```

```
private:
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
};

// Main function for the program
int main(){
    Box Box1;    // Declare Box1 of type Box
    Box Box2;    // Declare Box2 of type Box
    Box Box3;    // Declare Box3 of type Box
    double volume=0.0; // store the volume of a
                       box here
```

```
// box 1 specification
   Box1.setLength(6.0);
   Box2.setBreadth(7.0);
   Box1.setHeight(5.0);

// box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

// Volume of box 1
   volume=Box1.getVolume();
   cout << "Volume of Box1:" << volume << endl;

// Volume of box2
   volume = Box2.getVolume();
   cout<< "Volume of Box2:" << volume<<endl
```

```
// Add two object as follows:
   Box3 = Box1 + Box2;

// volume of box3
   volume = Box3.getVolume();
   cout << "Volume of Box3: " << volume << endl;
   return 0;
}
```

* Inheritance:

Inheritance is a process in which one object acquires all the properties and behaviours of its parent object automatically. In such way, you can reuse, extend or modify the attributes

and behaviours which are defined in other class. The derived class is the specialized class for the base class.

* Single Inheritance:

Single inheritance in c++ programming. Inheritance is the process of inheriting properties of objects of one class by object of another class. When a single class is derived from a single parent class, it is called single inheritance. It is the simplest of all inheritance.

* Multiple Inheritance:

Multiple inheritance is a feature of c++ a class

can inherit from more than one classes. The constructors of inherited classes are called in the same order, they are inherited.

Example:

Inheritance:

```
#include <iostream>
using namespace std;

class Person {
    public:
     string profession;
     int age;
  Person (): profession ("unemployed"), age (16){}
  void display ()
```

```
{
cout<<" My profession is: "<< profession<<endl;
cout<<"My age is: "<<age<<endl;

walk();
talk();}
void walk(){cout<<" I can walk." <<endl;}
void talk(){cout<<"I can talk." <<endl;}
};

//MathsTeacher class is derived from base
class Person.
class MathsTeacher: Public Person{
Public:
  void teachMaths(){cout<<"I can teach maths."
  <<endl;}
};
```

```
// Footballer class is derived from base class Person.

class Footballer : public Person{
  public:
    void playFootball(){cout<<"I can play football."
    <<endl;}
  };

int main(){
  MathsTeacher teacher;
  teacher.profession = "Teacher";
  teacher.age= 23;
  teacher.display();
  teacher.teachMaths();
```

```
Footballer footballer;
footballer.profession = "Footballer";
footballer.age = 19;
footballer.display();
footballer.playFootball();
return 0; }
```

* Polymorphism:

Polymorphism means having many forms. Polymorphism occurs when there is a hierarchy of classes and they are related by inheritance 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 the same time can have different characteristic.

Example;

Polymorphism;

```
#include <iostream>
using namespace std;
class Add {
  public:
    int sum (int num1, int num2){
        return num1+num2;}
    int sum (int num1, int num2, int num3){
        return num1+num2+num3;}
    };
```

```
int main(){
    Add obj;
    // This will call the first function
    cout<<" output :"<<obj.sum(10,20)<<endl;
    // This will call the second function
    cout<<" output :"<<obj.sum(11,22,33);
    return 0;}
```

* Overloading:

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters. Function overloading can be considered as

an example of polymorphism feature in c++.

Example:

Overloading:

```
class Dog{
    public void bark(){
        System.out.println("woof");}

// overloading method
public void bark(int num){
    for(int i=0; i<num; i++)
    System.out.println("woof");}
}
```

Overriding:

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables us to provide specific implementation of the function which is already provided by its base class.

Example:

Overriding:

```
class Dog{
  public void bark(){
  System.out.println("woof");}
}
```

```
class Hound extends Dog{
    public void sniff(){
    System.out.println("sniff");}
    public void bark(){
    System.out.println("bowl");}
}
public class OverridingTest{
    public static void main(String[] args){
    Dog dog = new Hound();
    dog.bark();}
}
```

* Encapsulation:

Encapsulation is an oop concept that binds together the data and functions that manipulate the data and that keeps both safe from outside inheritance and misuse. Data encapsulation led to the important oop concept of data binding.

Example:

Encapsulation:

```
#include <iostream>
using namespace std;
class ExampleEncap{
private:
```

/* Since we have marked these data members private.
* any entity outside this class can't access these
* data members directly, they have to use getter and
* setter functions.
*/

int num;
char ch;

public:
/* Getter functions to get the value of data members.
* since these functions are public, they can be accessed
outside the class. thus provide the access -

```
of data members
* through them
*/
int getNum() const{
    return num;}
char getCh() const{
    return ch;}
/* Setter functions, they are called for assigning
  the values.
* to the private data members.
*/
void setNum (int num){
    this->num=num;}
void setCh (char ch){
    this->ch=ch;}
};
```

```
int main(){
ExampleEncap obj;
 obj.setNum(100);
obj.setCh('A');
cout << obj.getNum() << endl;
cout << obj.getCh() << endl;
return 0;}
```

* Abstraction:

Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or

implementation.

Example:

Abstraction:

```
#include <iostream>
using namespace std;
class Adder{
  public;
  //Constructor
  Adder(int i=0){
    total=i;}

// interface to outside world
void addNum(int number){
    total+=number;}
```

```
// interface to outside world
int getTotal(){
    return total;}
private:
// hidden data from outside world
  int total;};

int main(){
  Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

cout<<"Total"<<a.getTotal()<<endl;
return 0;}
```

* Exception:

Exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another.

* Defined Exceptions:

Defined exception is a response to an exceptional circumstance that arises while a

program is running, such as an attempt to divide by zero. Defined exception handling is built upon three keywords: try, catch and throw.

Example:

Exception:

```
#include <iostream>
using namespace std;
int main () {
  try {
  throw 20; }
  catch (int e) {
  cout << "An exception occurred. Exception Nr. "
<< e << '\n'; }
return 0; }
```

* File Processing:

Files are nothing but a sequence of bytes without any structure. As we open a file, an object is created. This object is associated with a stream. This stream provides the communicating channel between the program and the file.

Example:

A file processing either ends with a specific byte number maintained by the underlying platforms administrable data structure or

with a marker called EOF (end-of-file). As we open a file, an object is created.

* Input And Output streams:

Input and output is done with streams. Stream is nothing but the sequence of bytes of data. A sequence of bytes flowing into program is called input stream. A sequence of bytes flowing out of the program is called output stream.

Example:

Input And Output streams:

```
#include<iostream>
```

```
#include<stdlib.h>
using namespace std;
int main(){
  float x=5.999;
  float *y, *z;
    y = &x;
    z=y;
  cout<<x<<", "<<*(&x)<<", "<<*y <<",
    "<<*z <<"\n";.

  return 0;}
```

* Class Template:

A class template provides a specification for generating classes based on parameters Class

Templates are generally used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments.

Example:

Class Template:

```
#include <iostream>
using namespace std;

template < typename A, typename B, typename SUM,
typename MUL, typename SUB, typename DIV,
typename RETURN >
RETURN testFunction.(A a, B b) {
```

```
SUM   sum = a+b;
SUB   sub = a-b;
MUL   mul = a*b;
DIV   div = a/b;

cout << sum << endl;
cout << sub << endl;
cout << mul << endl;
cout << div << endl;

return (RETURN) true; }

int main (){
  Point< double, double, int, double, double>.
    P(5.5, 6.5);
  P-showPoint();
```

```
cout << p.sum(3.5, 4) << endl;
cout << testFunction<double, double, double,
    double, double, double, bool>(5, 5, 6);
return 0;}
```