Assignment 2

Page No.
Date:

1) what is Constructor? Explain Explicit and parameterized Constructor in detail

→ A Constructor is a member functions of a class whose name is same as the class name.

→ As the name of the Constructor is same as its class name, it is called special function, with an only limitation that it can not return any value.

The Syntax of Constructor

```
<class_name> (list_of_parameters) {

}
```

if the Constructor is declared outside of class the structor / Syntax of Constructor be like

```
<class_name> :: <class_name> (list_of_Parameter) {
   // definition
}
```

→ Suppose we take example of stack, if the value of stack pointer is not initialized with zero, Both push and Pop operation would not work as expected. This problem is avoided in the following Program. Each time an object of stack class is defined, the stack() function would automatically be called.

→ Example

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

class stack{
  private:
    int stackpointer;
    int StackArray[10];
  public:
    stack(){
      Stackpointer = 0;
    }
  void push(int value){
    if (stackpointer == 0){
      count << "StackOverflow";
    }
    else{
      StackArray[stackpointer] = value;
      stackpointer++;
    }
  }
  int pop(){
    if(Stackpointer == 0){
      count << "Stack underflow!";
    else{
      stackpointer--;
      return StackArray[Stackpointer];
    }
  }
};
```

```
Void main(){
    Stack S1;
    S1.push(1);
    S1.push(2);
    Count << S1.pop() << End1;
    Count << S1.pop() << End1;
}
```

→ Explicit Constructor -

In most cases automatic conversions add to readability, However, there are times when we can do without such implicit conversion The creation of the conversion operator can be avoided by using keyword Explicit before the classname while defining object. If we do not want a form of

brother forth = "Ricky"

to work like

brother forth = ("Ricky")

then we need to proceed the Constructor name by keyword explicit.

→ Example of Explicit Constructor

```
#include <iostream>
class Brother {
    String Name;
  public:
    explict Brother (String brotherName){
        name = brotherName;
    }
};

Void main() {
    Brother first = brother ("Steve");
    Brother Second ("Mark");
    Brother Third ("Gilchrist");
}
```

→ Parameterized Constructor:

When the Constructor Contains a single or multiple arguments, it is known as Parameteriz Constructor.

This is useful when Constructor are needed for Creating objects, which require data value initialization.

We need to pass values to parameterized Constructo Unlike default Constructor, This can be done in two way. ① Pass a set of arguments when the object is defined ② Explicitly Call Constructor fonction to reinitialize an object

Page No.
Date :

⇒ Example

```
#include <iostream>
class student {

public :
  int rno : string name
student () { }

Student (int trno, string tname) {
    rno = trno;
    name = tname;
}

Void display () {
  Count << "Rno : " << rno;
  count << "Name : " << name;
}
};

Void main () {
  Student cricket stu (1, "Brain");
  Student footballStu (2, "Devid");

  Student tenisStud;
  // Explicit Call to constructor function
  tenis stud = student (3, "steffy");

  cricketstu. display ();
  football stu. display ();
  tenis stud. display ();
}
```

Q2 Explain Copy Constructor with example.

→ Copy Constructor is one with a single argume as a reference to the very class it belongs

→ The process of initializing member of objects th a copy constructor is known as copy initializatio

→ Copy Constructor is also a kind of parameterize Constructor, because the new object is created and the object is equated to the newly crea object. That is existing object is passed a a reference parameter to copy Constructor.

→ Example :

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

class Code {
  int id;
  public:
    void Code(int x) {
        id = x;
    }
    void display() {
        cout << id << endl;
    }
};
```

```
int main() {
    Code obj1;
    obj1.init(5);
    obj2.display();
    Code obj2 = obj1;
    obj2.display();
    return 0;
}
```

Q7 Explain MIL with Example

→ Member wise initialization list or member initialization list (MIL) is the method for initialization of the member of class using the Constructor function.

→ It provides an alternative for providing initializations outside the Constructor body.

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

Class Time {
  public:
    int Hours;
    int Minutes;
    int Seconds;
    Void Showtimes() {
        Count<<"Time is" << Hours << "Hours:";
            << Minutes << "minus: and" <<
            Seconds << "seconds \n";
    }
```

```
Time(){}

/* Starting of MIL Constructor */

Time(int tHours, int tMinutes, int tseconds)
  : Hours(tHours);
    Minutes(tMinutes);
    Seconds(tSeconds)
    { } // Empty block (body)

};

Void main(){
  Time time1(12,15,15);

  Count << "Time number 1 \n";
  Time1.showTime();
}
```

→ The Normal parameterized Constructor goes for the above program is following:

```
Time(int tHours, int tMinutes, int tSeconds){

    Hours = tHours;
    Seconds = tseconds;
    Minutes = tMinutes;
}
```

→ The parameterized Constructor and MIL work the same way. An MIL appears in between

the header and body starting with colon.

→ The MIL is the only way to initialize constants, reference and object that are data members of a class for which the constructor is being written.

→ When the MIL is used, the order of initialization is not the same as when they are defined in the list It is the order of their declaration in the class

Q4 What is destructor? Why we need it?

→ Destructor are special functions that are used to execute automatically when the object of the class goes out of scope.

→ They have the same name as class prefixed by '~' for example ~Customer() is a destructor for class Customer.

→ The code written in body of function ~Customer() would be executed. When the object of type Customer goes out of scope. The destructor would also be called when delete is executed.

→ Characteristics of destructor:

A destructor is invoked automatically by the compiler

When its corresponding Constructor goes out of scope and release the memory that is no longer required by the program.

→ A destructor will not return any value and does not accept arguments and therefore it cannot be overloaded

→ A destructor cannot be declared as static, const or volatile.

→ A destructor should be declared in public section.

→ It is necessary that a destructor use a delete expression to deallocate the memory, if the Constructor in the program use the new expression for allocating the memory

→ A destructor is called in reverse order of its Constructor invocation.

→ Example :

```
#include <iostream>
using namespace std;
class point{
  private:
    float x,y;
  public:
    point(point & otherpoint){
```

```
    x = otherpoint.x;
    y = otherPoint.y;
}

Point (float tempX=0, float tempY=0){
    x = tempX;
    y = tempY;

}
~Point(){
    count << "Point x" << x << "Point y" << y <<
        "Destroy";
}

};

Void main(){
    point Point1(2,3);
    point Point2(4,5);
        // destructor called

}
```

Output: ~~x 2 y 3 destroy~~

Point x 2 y 3 destroy
Point x 4 y 5 destroy

Q5 Explain Unary and Binary Operator Overloading with example?

→ Operator overloading enables the use of our own objects using operators which were reversed for built in types in C.

→ Operator Overloading is performed by adding special member function to the class, these function are known as Operator function and can help convert one object into another.

→ Unary Operator

- Operator that have a single argument are known as Unary operator, when these operators are overloaded as member functions it is not necessary to pass any argument explicitly

- The this pointer pointing to the invoking objects is passed as an implicit argument

→ Example

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

```
class Matrix {
    int Element[3][3];
    public:
        Void read(){
            for(int i=0; i<3; i++){
                for(int j=0; j<3; j++){
                    cin >> Element[i][j];
                }}
        }
        Void operator-(){
            for(int i=0; i<3; i++){
                for(int j=0; j<3; j++){
                    Element[i][j] = -Element[i][j];
                }
            }
        }
        Void display(){
            for(int i=0; i<3; i++){
                for(int j=0; j<3; j++){
                    Count << Element[i][j] << " ";
                }
                Count << "\n";
            }
        }

};
Void main(){
    Matrix m1;
    Count << "Enter value: ";
    m1.Read();
    -m1;
```

```
    cout << "Result";
    m1.display();
}
```

→ Binary Operator overloading

- Binary Operator are operators which operate on two operands.
- The first arguments which is passed in binary operator overloading is reference of second object, first object is implicitly called.

- Example.

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

class Sample {
  private:
    int a,b;

  public:
    Sample(int x, int y){
      a=x; b=y;
    }
    Sample(){ }

    Sample operator+(Sample &s){
      Sample k;
      k.a = a+s.a;
      k.b = b+s.b;
      return k;
    }
```

```
Void display(){
    Cout << "a = " << a << End1;
    Cout << "b = " << b << End2;
}
};
Void main(){
    Sample S1(5,6);
    Sample S2(6,7);
    Sample S3;
    S3 = S1 + S2;
    S3.display();
}
```

Q6 Explain the need of friend as Operator function in overloading an Operator

→ We know that friend function can be used as operator function.

→ There are two cases where it is really important to use friend function.

→ The first case involves a non-class first argument while second case involves conventional Overloading of operators that require objects on the right hand side (RHS) and not on the LHS.

→ Example [Case 1:]

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

Class Matrix {
    int Element[3][3];
    public:
        Matrix() { }
        Void read () {
            for (int i=0; i<3; i++) {
                for (int j=0; j<3; j++) {
                    Cin >> Element [i][j];
                }
            }
        }
        Void display {
            for (int i=0; i<3; i++){
                for (int j=0; j<3; j++){
                    Count << Element [i][j] << " ";
                }
                Count << "\n";
            }
        }
        friend Matrix Operator *(Matrix, int);
};

Matrix Operator *(Matrix tmat, int mul){
    for (int i=0; i<3; i++){
        for (int j=0; j<3; j++){
            tmat.Element [i][j] = mul * tmat.
                                  Element[i][i];
        }
    }
    return matrix (tmat.Element);
}
```

Page No.
Date :

→ Example [Second case]

```
#include <iostream>

int plus (int tempX, int tempY){
    cout << "plus() is called";
    return tempX + tempY;
}

int Minus (int tempX, int tempY){
    cout << "Minus() is called";
    return tempX - tempY;
}
int FunctionPointer (int (*Funptr)(int, int), int arg1,
                     int arg2){

    return Ptr(Arg1, Arg2);
}
void main(){
    int arg1 = 20, arg2 = 5;
    cout << FunctionPointer(Plus, arg1, arg2);
    cout << FunctionPointer(Minus, arg1, arg2);
}
```

Output:

```
plus() is called
25
minus() is called
15
```

Q7 Why we need user defined conversion and explain four different cases where user defined conversion are needed.

→ Object assignment is simple when both object involve are of same type. We call it member-to-member copy. But when they are different type then we need to define user defined conversion.

→ As the objects are different and the members are not the same the compiler cannot go for member-by-member copy.

→ There are two type of conversion:

1. Implicit conversion
2. Explicit conversion

→ In Implicit conversion Compiler Convert object it self.

- Explicit conversion have ~~four obj~~ type:

① Build in dates types to Object:

One specific method to solve the problem of dissimilar objects assignments is to use Constructor. For example, Complex C1(2, 3) takes two arguments of built-in data type and converts to a complex objects. wherever

we use Constructor, we Convert the argument type to native object type of Constructor.

Example:

Suppose we define a class length and would like to have Constructor as follow:

```
Class Length {
  int L;
  public:
    Length (int templength = 0) {
        L = templength;
    }
};
```

if we define object length Door(8), basically we Construct the object from 8; an integer value In a way we are converting an integer value into a object.

(2) Object to Built-in-Data type

Suppose we have logged In user class with Contents such as Name, Tokens In the function we may need to print error message. In that case we write Count << [IU], name where [IU] is object In the statement there is a problem if Name is declared private the statement will not work We have to either make Name public or write a member function to access name.

- When we write a function such as printName we need to write LUI.PrintName() It is better to write just loggedInUser to get the userid instead of count << LIUI. This can be easily be alone using a conversion function.

- Example

```
#include <iostream>
#include <string>

Class LoggedInUsers {
    String Name;
    int Token NO;
    Static int TotalLogged In;
    public:
    Logged In User ( ) {
        Totallogged In ++;
        Count<< "In you are user NO" <<
            Total Logged In:
    }
    Void Insert Name ( ) {
        Cout <<"Insert name of new user";
        Cin >>name;
    }
    String ( ) {
        return Name;
    }
};
Int LoggedInUser :: TotalLogged In;
Void main ( ) {

    LoggedInUser *Arrayofuser [100];
```

```
int index = 0;
int choice;

While (true){
    Cout << "\n1. new User";
    Cout << "\n2. List User";
    Cout << "\n3. Exit;
    Cout << "\n Your choice:";
     Cin >> choice
     if (choice == 1){
         if (index == 100){
          Cout << "Too many User";
           exit(1);
         }
     }
     Array of users[index] = new LoggedIn
                             User;
     Array of users[index] -> InsertName();
      index ++;
    }
    else if(choice == 2){
      for (int i=0; i<index; i++){
         String Nameofuser = Array ofuser[i];
         Cout << Name of user;
      }
    }
    else
      exit(0);
  }
}
```

(3) Conversion of object type Using Constructor.

- It is possible to convert from one type of object into another using either Constructor or Conversion function. There are two different cases in such conversion.

- They are Conversion from foreign objects into a native one and visa versa.

Example:

```
#include <iostream>
#include <string>
#include <cmath>

Class Polar{
    double Radius;
    double Angle;
    Public:
      Polar (double tRadius=0, double tAngle=0){
        Radius = tRadius;
        Angle = tAngle;
      }
      double getRadius(){
        return Radius;
      }
      double getAngle(){
        return Angle;
      }
};
```

```
class Cartesian {
    double x;
    double y;
  Public :
    Cartesian (double tempX = 0, double tempY = 0) {
        X = tempX;
        y = tempY;
    }
    Cartesian (Polar PolarPoint) {
        double tRadius = PolarPoint.getRadius();
        double tAngle = PolarPoint.getAngle();
        X = tRadius * cos(tAngle);
        y = tRadius * Sin(tAngle);
    }

    Void Show() {
        Count << "(" << x << ", " << y << ")" "\n";
    }
};

Vad main() {
    Cartesian CPoint(10,10);
    polar PPoint (10, 45);
    Polar PPoint;
    Cartesian CPoint2;
    CPoint2 = PPoint;
    CPoint2. Show();
}
```

Page No.
Date :

(4) Conversion of object Type Using Conversion function;

→ When a native object needs to be converted into foreign object, Operator functions Conversion function are used.

→ Example

```
#include<iostream>
#include<string>
#include<math>

Class Cartesian;
Class Polar; {
   double Radius ; double Angle;
   Public:
      Polar(double tRadius=0, double tAngle=0){
         Radius = tRadius ; Angle = tAngle;
      }
      double getRadius{
            return Radius;
      }
      double get angle{
            return Angle;
      }
      void show(){
         Count << "(" << Radius << ", " << Angle
                  << ")\n";
      }
};
```

```
class Cartesian {
    double X, Y;
    Public:
        Cartesian (double tx=0, double ty=0) {
            X = tx;
            Y = ty;
        }

        Cartesian (Polar Polar Point) {

            double tRadius = Polar Point.getRadius();
            double tAngle = Polar Point.getAngle();
            X = tRadius * Cos (tAngle);
            Y = tRadius * Sin (tAngle);
        }

        Operator Polar() {
            double tangle = atan(X/Y);
            double tRadius = Sqrt (X*X + Y*Y);
            return Polar (tRadius, tAngle);
        }

        Void Show {
            Cout << "(" << X << "," << Y << ")\n";

        }
};
Void main() {
    Cartesian CPoint1(10,10);
    Polar PPoint2(10, double(0.5));
    Polar PPoint1; Cartesian CPoint2;
    CPoint2 = PPoint2;
```

```
PPoint1 = (Point1);
(Point1.show());
PPoint->show();
}
```

Q8 What is template function? Explain non generic parameters in Template function with example?

→ Function template are generic function that work for any data type that is passed to them. the data type is passed to them, specified while writing the function.

→ While using that function, the data type is passed and the required functionality is obtained

Syntax is:

```
template <class T>
return-type function-name(Parameters of T){
    // function body
}
```

→ Non-generic Parameter
A non-generic parameters means we have to pass the data type. It will not substitute for a type but it is instead replaced by value.

Example:

```
#include <iostream>
#include <string>
template <typename Type>
Void BubbleSort (Type GenericArray [], int size){
  for (int i=0; i<size-1; i++){
        for (int j=i+1; j<size; j++){
            if (GenericArray[i] < GenericArray[j]){
                Type Temp = GenericArray[i];
                GenericArray[i] = GenericArray[j];
                GenericArray[j] = Temp;
            }
        }
    }
}
Void main(){
    int Array2[] = {1, 5, 10, 12, 15, 18, 20};
    Char Array3[] = "Hello";
    BubbleSort(Array2, 7);
     for (int i=0; i<7; i++){
          Cout << "\n" << Array2[i] << ",";
     }
     Count << "\n";
     BubbleSort(Array3, 5);
     for (int i=0; i<5; i++){
         Count << " " << Array3 << ",";
     }
}
```

Q9 What is class Template Explain classes with multiple generic data types

→ like generic classes, which take data type as parameters are also possible in C++.

Example; let us consider we make use of three set of classes each of these classes does the same kind of operation but the only difference being that each operation operate on different data member. This increases the program size and hence maintaince of code become more tedious. Hence to avoid these kind of situation we make use of class Templates
→ Using class Templates, we can write a single generic class which does the job of all three.

* classes with multiple Generic data type.

→ classes, Similar to functions templates, can have more than one generic type.

Example.

```
#include <iostream>
#include <string>

Template <typename Type1, typename Type2>
class ClassWithType
{
    Type1 firstVal;
```

```
    Type2 SecondVal;
    public:
      classWithType (Type1 Tval1, Type2 Tval2){
        firstVal = Tval1;
        SecondVal = Tval2;
      }
      void display(){
        Count << FirstVal << " " << SecondVal;
      }
};
void main(){
    classWithType <int, char> objectIC(12, 'b');
    classWithType <char, String> objectIs('b', "str");
    ObjectIc.display(); Count << "\n";
    objectIs.display();
}
```

Q10 Explain static data member in class template and explain use of expert keyword.

→ The template class can also have static data member. The static variable will have one instance for one initialization, of the template class.

→ Thus there will be two different static member for Stack<int> and for Stack<char>.

→ For all objects of a single class there is only one instance of static member. The way a static member is defined is analogous to the way member functions are defined outside the template class.

Example

```
#include <iostream>
#include <string>
template <typename ElementTypes>
class Stack {
    int StackPointer;
    ElementType StackArray[10];
  Public:
    Static int TotalStacks;
    Stack(){
      StackPointer = 0;
      total Stack ++;
    }
    Void Push(ElementType);
      ElementType Pop();
};

template <typename ElementType>
int Stack<ElementType>::total Stacks;

template <typename ElementType>
Void Stack<ElementType>::push(ElementType Value){
    if (StackPointer > 9){
      Count << "StackOverFlow";
    }
  else{
    StackArray[StackPointer] = Value;
    StackPointer ++;
  }
}
```

```
template <typename Elementtypes>
Elementtype Stack <Elementtype> :: pop() {
    if (Stackpointer == 0) {
        Count << "Stack Underflow";
    }
    else
        Stackpointer--;
    return StackArray [Stackpointer];
    }
}
Void main() {
    Stack <int> mystack1;
    Count << "int stack element " << Stack <int> ::
                                    Total Stacks;

    Stack <char> yourstack;
    Stack <int> mystack2;
    Stack <char> yourstack2;
    getchar();
}
```

→ Export keyword

- The keyword export is useful when a template function is defined at a single place and the declaration are used at other place.

→ This is useful instances where the templates defines while developing one application are found to be useful for other applications.

→ Suppose we define Bubblesort app to contain definition of bubblesort. Then it is possible to modify the definition as follow:

export template <typename Type>

Void BubbleSort (Type TempIntArray [ ]) {
// Body of function
}

Q11 Explain the advantages of C++ I/O over C I/O

→ Though C I/O is robust and proven, there are a few distinct advantages of using C++ I/O which are as follows:

→ C++ I/O is object-oriented, objects represent Stream in C++. Cout is an object of the outputstream class, where as Cin is an object of the input Stream class.

<< and >> are overloaded operators in those Streams. Using objects from istream or Ostream class.

→ C++ I/O Stream Contains richer formatting operation than C. It is possible to have to programmers own format operators known as User-defined manipulators in C++.

→ Though apperent at first glace, C++ I/O is much easier to use. We can take input using Cin without the '&' Operator Unlike Scanf(), we also use overloaded << and >> operators for I/O

Q12 Explain ios members functions for formatting

→ The build in ios functions is also called ios member function.

→ The following are the list of ios member functions:-

width()
precision()
fill()
setf()
unsetf()

→ The prototype for all the following function is =

<old value of stream> fun-name(<specified new val>)

→ The function set the new value to the stream and return the old value.

→ width()
- It specifies the minimum field width for display. The width() reset itself after the first output after statement The output of the statements.

```
cout.width(10);
cout << "C++";
```

→ precision()
- it specifies precision, that is, the number of digit to be displayed after the decimal point.

→ default value is six

→ The precision function is important while displaying numbers in scientific notation pointing amount digits where possible is two, or aligning floating point numbers for vertical alignment.

→ fill()
- The function fill(char) fills the subsequent empty position of field by the fill character specified.

```
Count.fill('*');
Count.width(10);
Count<<"C lang";
```

Output will be : * * * * C lang

→ setf()

→ This specifies the format flags that control output display such as left or right justification scientific notations display and displaying base of the number.

→ Unsetf()
- This provides undo operations for the options with setf()

Q3 What is manipulator? Explain different formatting manipulator also give brief description about user defined manipulator

→ Manipulators are special function for formatting They can do all the formatting that is done by the ios member functions.

→ Manipulators are better in some circumstances and provide an alternative way to solve the same problems

① Set w()
It is used to set field width to 'width'.
Example
cout << setw(5) << 30 << endl;
cout << setw(-5) << 30 << endl;

② set precision()
Used to set the floating point precision to decimal
Example
cout << setprecision(3) << sqrt(3) << endl;
cout << set precision(4) << sqrt(3) << endl;

Output
1.732
1.7321

(3) setfill()
- Used to fill empty column obtain after using the manupilators 'setw()' by character

Example:
cout: setfill('$') << setw(6) << 10 << endl;

Output:
$$$$10

(4) setiosflags()
- set the format flags to flags.
- flags include 'ios::showpoints', ios::

(5) resetiosflag()
- clear the format flag specified by flags.

(6) endl:
- used to end line in program and flush stream.

* User defined Manupilators
- Manupilators can also be defined to suit particular requirements. The following program will demonstrate how we can write and use one's own manipulators.

Example

```
#include <iostream>
#include <iomanip>

//first manipulator
Ostream & printHeading(Ostream & Templat) {
  Tempout << setw(80) << setiosflags(ios::left);
  Tempout << "LL-GLS Higher Secondary" << endl <<
    setw(80) << "Standard kII" << endl;
  return Tempout;
}

//Second manipulators
Ostream & printMarksheet(Ostream & Tempout) {
  Tempout << setw(15) << setiosflags(ios::left)
  << setiosflags(ios::fixed) << setprecision(2) <<
  setiosflags(ios::showpoint);

  TempOut << "Roll number" << setw(15) << "Name"
  << setw(10) << setprecision(2) << "Marks" <<
  endl;
  return Tempout;
}
// Third manupilator
Ostream & printline(Ostream & Tempout) {

  Tempout << "-----";
  Tempout << endl;
  return Tempout;
}
```

```
int main() {
    cout << printline;
    cout << printheading << printline <<
    printmarksSheetHeading;
    cout << printline;
    cout << setw(15) << 1 << setw(15) <<
"Usra" << setw(10) << setprecision(2)
<< 355.50 << endl;

cout << setw(15) << 2 << setw(15) <<
  "steff" << setw(10) << setprecision(2)
<< 290.75 << endl;
  cout << printlines
}
```