CE0316-Object Oriented Programming with UML

ASSIGNMENT-1

INSTRUCTIONS:

1. Neat presentation.
2. Include well labelled diagrams.
3. Write down differences in tabular format.
4. Mention and Highlight important text.
5. Use only blue and black ink for presentation

QUESTION/ANSWERS:-

1. Explain features of object oriented programming language.

Ans

- Object - oriented programming - as the name suggests uses object in programming. It's aims to implement real-world entities like inheritance, hiding, polymorphism etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

- The basic concepts of Object-Oriented Programming language are :

i). Class,
ii). Objects,
iii). Encapsulation,
iv). Abstraction
v). Polymorphism
vi). Inheritance

i). Class :-

- The building block of C++ that leads to object-oriented programming is a class. 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 instance of that class. A class is like a blueprint for an object. Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behaviour of the objects in a class.

ii). Objects :-

- An object is an identifiable entity with some characteristics and behaviour. An object is an instance of a class. When a class is defined, no memory is allocated but when it is instantiated memory is allocated. Objects take up space in memory and have an associated address like a record in pascal or structure or union. When a program is executed the objects interact by sending messages to one another.

iii). Encapsulation :-

- Encapsulation is defined as wrapping up data and information under a single unit. It is defined as binding together the data and the functions that manipulate them. It also leads to data abstraction or data hiding. Using encapsulation also hides the data.

iv). Abstraction :-

- Data- Abstraction is one of the most essential and importance features of object-oriented programming. It refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Abstraction can be implemented in C++ using class. The class helps us to group data members and member functions using available access specifiers. A class can decide which data member will be visible to the outside world and which is not.

v). Polymorphism :-

- 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. An operation may exhibit different behaviours in different instance. The behaviour depends upon the

types of data used in the operation. C++ supports operator overloading and function overloading.

vii). Inheritance :-

- The capability of a class to derive properties and characteristics from another class is called inheritance. It is one of the most important features of Object-oriented programming. The class that inherits properties from another class is called Sub-class. The class whose properties are inherited by a Sub-class is called Super class.

2. Describe data types in C++ in detail.

Ans

- The data types that are supported in C++ are as follows :

i). Primary data Type.
ii). Derived data Type.
iii). User-defined data Type.

i). Primary data Types :-

- These data types are built-in or predefined data types and can be used directly by the user to declare variables. Primitive data types available in C++ are :

i). Integer,
ii). Character,
iii). Boolean,
iv). Floating point
v). Double floating point
vi). Valueless or void
vii). Wide character.

ii). Derived data Types :-

- These data types are derived from the primitive or built-in datatypes are referred to as derived data types. There are 4 types of derived data types :

i). Function,
ii). Array,
iii). Pointer,
iv). Reference

iii). User-Defined data Types :-

- User-defined data types are defined by the user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes :

i). Class,
ii). Structure,
iii). Union,
iv). Enumeration,
v). Typedef defined datatype.

3. Write different applications of OOP.

Ans

- Object-Oriented Programming (OOP) is widely used in various fields of software development due to its ability to model complex systems, promote code reuse, and simplify maintainance. The applications of OOP are :

i). Software development,
ii). Web development,
iii). Mobile-Application development,
iv). Real-Time Systems,
v). Database Management Systems,
vi). Simulation and modeling
vii). Graphical User Interface's (GUI's)
viii). Artificial Intelligence and Machine learning
ix). Enterprise applications
x). Distributed Systems
xi). Security Systems
xii). Automated Testing

i). Software Development :-

- Object-oriented Programming is used extensively in developing desktops applications like word processors, media players, and graphic editors.

ii). Web Development :-

- Many Server-side languages, such as Java, Python, PHP, and Ruby, use OOP Principals to create web applications. Modern frameworks like React, Angular and Vue.js encourage the use of OOP concepts, such as components, to create modular and maintainable user interfaces.

iii). Mobile-Application Development :-

- OOP is fundamental in mobile application development. For android, Java and Kotlin are used. For Ios, Swift and Objective-C are the primary languages. These languages allow developers to create class representing UI elements, business logic, and more, making the codebase modular and resuable. Frameworks like Flutter and Xamarin use OOP principles to allow developers to create and write code once and deploy it across multiple-platforms like Ios, Android, Windows, MacOS etc.

iv). Real-Time Systems :-

- OOP is used in embedded system programming, such as in automotive software, medical devices, and industrial automation. C++ is often used to model real-time components and integrations, making the code more scalable and easier to manage.

v). Database Management System :-

- Object-Relational Mapping (ORM) like Hibernate (Java), SQLAlchmey (Python), and Entity Framework (C#) use OOP to map database tables to classes. This abstraction allows developers to interact with database using objects rather than writing raw SQL Queries. Data modeling is used in designing complex data models for database management systems, where classes represent tables and their relationships.

vi). Simulation and Modeling :-

- OOP is used in scientific simulations to model complex systems such as weather patterns, biological processes, and physical phenomena. Languages like Python and C++ are commonly used in scientific computing.

viii). Graphical User Interface (GUI) :-

- OOP is commonly used in developing GUI's, where different UI components like buttons, text fields etc. are represented as objects. Framework like Qt (C++) and JavaFx (Java) use OOP to manage user interfaces effectively.

viii). Artificial Intelligence and Machine Learning :-

- OOP is used in developing machine learning libraries like TensorFlow (Python), Scikit-learn (Python) and Keras (Python). OOP is also used in AI systems to model intelligent agents, environments, and interactions.

ix). Security Systems :-

- OOP is used to design systems for managing user's authentication and authorization. Objects can represent user's, roles, permissions, and sessions.

x). Automated Testing :-

- OOP facilitates the creation of testable units in software. Testing frameworks like JUnit (Java) and NUnit (C#) use OOP to create test cases and suites. Each test case is an object that can be executed independently. It is used in creating automated testing tools and scripts that simulate user interactions, test API's, and verify system behaviour.

4. Give the difference between POP and OOP Language.

Ans

Feature	POP (Procedural-Oriented Programming)	OOP (Object-Oriented Programming)
Basic unit of code	Functions or Procedures	Classes and Objects
Focus	Process and Logic	Data and behaviour
Approach	Top-down	Bottom-Up
Data Handling	Data is passed through functions, often global	Data is encapsulated within objects
Security	Less secure. Data is exposed and accessiable globally	More secure. Data is hidden and accessed through methods
Modularity	Functions are the modular unit	Classes and objects are the modular units.
Code Reusability	Achieved through functions but limited	High reusability through inheritance, polymorphism and encapsulation.
Complexity Management	Difficult to manage as the program grows larger	Easier to manage complexity through objects and classes.

5. Describe various operators in C++.

Ans. There are seven type of operators in C++. They are :

i). Arithmetic Operator
ii). Relational Operator
iii). Logical Operator
iv). Bitwise Operator
v). Assignment Operator
vi). Unary Operator
vii). Conditional Operator

i). Arithmetic Operator :-

- These operators are used to perform arithmetic or mathematical operations on the operand.

Operator	Description
'+'	Addition : Add two values
'-'	Subtraction : Subtracts one value from another
'*'	Multiplication : Multiplies two values
'/'	Division : Divided one value from another
'%'	Modulus : Returns the remainder of division

ii). Relational Operator :-

- These operators are used for the comparsion of the values of two operands.

Operator	Description
'=='	Equals To : Checks if two values are equal
'!='	Not Equals To : Checks if two values are not equal
'>'	Greater than : Checks if one value is greater
'<'	Less than : Checks if one value is less.
'>='	Greater or equal to : Check if value is greater or equal

iii). Logical Operator :-

- These operators are used to combine two or more conditions or constraints or to complemen evaluation of the original condition is consideration.

Operator	Description
'&&'	Logical AND : Returns true if both operands are true
'||'	Logical OR : Returns true if either of the operands is true
'!'	Logical NOT : Returns true if the operand is false or zero

iv). Bitwise Operator :-

- These operators are used to perform bit-level operations on the operands.

Operator	Description
'&'	Bitwise AND : Performs bitwise AND between two values
'|'	Bitwise OR : Performs bitwise OR if it exists in any values.
'^'	Bitwise XOR : Performs bitwise XOR between two values.
'~'	Bitwise NOT : Inverts all bits of the operand
'<<'	Left Shift : Shifts bits of the first operand left
'>>'	Right Shift : Shifts bits of the first operand right

v). Assignment Operator :-

- These operators are used to assign a value to a variable.

Operator	Description
'='	Assignment : Assigns a value to a variable
'+='	Add Assignment : Adds and assigns the result
'-='	Subt Assignment : Subtracts and assigns the result
'*='	Multiply Assignment : Multiplies and assigns the result.
'/='	Division Assignment : Divides and assigns the result
'%='	Modulus Assignment : Apply modulus and assigns the result.

vi). Unary Operator :-

- Unary operators are the operators that perform operations on a single operand to produce a new value.

Operator	Description
'+'	Unary Plus : Indicates Positive value
'-'	Unary minus : Negates the value
'++'	Increment : Increases value by 1
'--'	Decrement : Decreases value by 1
'&'	Address-of : Returns the memory address of a variable.
'*'	Dereference : Access the value at a memory address
'sizeof'	Sizeof : Returns the size of a data type or variable

vii). Conditional Operator :-

- This operator returns the value based on the condition.

Operator	Description
'?:'	Teracry : Returns one of two values based on a condition

6. Explain Scope Resolution Operator with Example.

Ans

- The Scope Resolution Operator ('::') in C++ is used to define the context in which a particular identifier is defined. It helps in resolving the scope of the identifiers, especially when there are multiple scopes or when global and local variables have the same name.

- The uses of Scope resolution operator are :

i). Accessing Global variables
ii). Accessing Class members
iii). Namespace resolution
iv). Accessing Nested classes

- Example of Scope resolution operator :

```
#include <iostream>
int x = 10;
class MyClass {
    public :
        static int count;
        void display();
    };
 int MyClass :: count = 0;

void MyClass :: display() {
    std:: cout << "Display function of MyClass" << std:: endl;
}
namespace First {
    int value = 1;
 }
namespace Second {
    int value = 2;
 }
```

```
int main() {
    int x = 20;
    std:: cout << "Local x = " << x << std :: endl;
    std :: cout << "Global x = " << :: x << std :: end1;

    Myclass obj;
    obj. display();

    Myclass :: count = 10;
    std:: cout << " Myclass :: cout = " << Myclass :: cout << std :: end1;

    std :: cout << " First namespace value :" << First :: value << std :: endl;
    std :: cout << " Second namespace value :" << Second :: value << std :: endl;

    return 0;
}
```

Output :-

```
Local x = 20
Global x = 10
Display function of Myclass
Myclass :: count = 10
First namespace value : 1
Second namespace value : 2
```

7. Explain Inline function and function overloading with the help of an example. Write rules of function overloading and inline function.

Ans

i). Inline Function :-

- An inline function is a function that is expanded in line when it is called. The compiler replaces the function call with the function's code, thus eliminating the overhead associated with function calls. This can lead to faster execution for small, frequently used functions.

- The rules for inline function are :

i). The 'inline' keyword is a request to the compiler, not a command. The compiler may ignore it if the function is too complex.
ii). Inline functions should be small and simple, as large functions might not be inlined.
iii). Inline functions cannot contain loops, switch statement, or recursion.
iv). Inline functions should be defined in the header file if they are used in multiple files.

- Example of Inline function :

```
# include <iostream>

inline int square (int x) {
        return x * x;
}
int main() {
    std:: cout << "Square of 5 :" << square(5) << std:: endl;
    return 0;
}
```

Output :-

Square of 5 : 25

ii). Function overloading :-

- Function overloading allows you to define multiple functions with the same name but different parameters. The correct function to call is determined by the number and types of arguments passed.
- The rules for function overloading are :

i). Function overloading is based on the number and types of parameters. The return type alone cannot be used to differentiate overloaded functions.
ii). The parameters must differ in either number or type for overloading to work.
iii). Functions with the same name must be declared in the same scope or class.
iv). Defaults arguments can complicate overloading, so they should be used carefully.

- Example of function overloading :

```
include <iostream>

void print(int i) {
    std:: cout << "Printing int :" << i << std :: endl;
}
void print (double d) {
    std :: cout <<"Printing double :" << d << std :: endl;
}
Void print (Std:: string s) {
    std:: cout <<" Printing string :" << s << std :: endl;
}
```

```
int main() {
    print(10);
    print(5.5);
    print("Hello!");
    return 0;
}
```

Output :-

```
Printing int : 10
Printing double : 5.5
Printing String : Hello!
```

8. What is a friend function ? Explain with examples.

Ans

- A friend function is a function that is not a member of a class but has the privilege to access the private and protected members of the class. Normally, private and protected members of a class are accessible only by member functions or friends.

- Friend functions are useful when you need to allow a non-member function to access the internal members of a class, typically when a certain operation requires access to two different classes.

- Syntax of a friend function :

```
class ClassName {
    friend ReturnType FunctionName (Parameters);
};
```

- Example of a friend function :

```
#include <iostream>

class Box {
    private :
        double width;
    public :
        Box() : width(0) {}
        void setWidth (double w) {
            width = w;
        }
        friend void printWidth (Box box);
};
```

```
void printWidth (Box box) {
        std :: cout << "Width of box : " << box.width << std::endl;
}

int main() {
    Box myBox;
    myBox.setWidth(10.5);
    printWidth(myBox);
    return 0;
}
```

Output :-

```
Width of box : 10.5
```

9. Explain default arguments with example.

Ans

- Default arguments in C++ allow you to specify a value for a function parameter that will be used if no argument is provided for that parameter when the function is called. This feature makes functions more flexible and easier to use by reducing the need to overload functions with different number of parameters.

- Default arguments are specified in the function declaration. and can be provided for any or all of the parameters. However, once a parameter is given a default value, all subsequent parameters must also have default values.

- Syntax for default arguments :

```
ReturnType FunctionName (ParameterType1 param1 = defaultValue1,
                         ParameterType2 param2 = defaultValue2,
                         ..... );
```

- Example of default arguments :

```
# include <iostream>

void displayMessage (std:: string message = "Hello, World!", int
                                                    repeat = 1 ) {
    for ( int i = 0 ; i < repeat ; ++i ) {
        std:: cout << message << std:: endl ;
    }
}

int main() {
    displayMessage () ;
    displayMessage ("Hello, C++!") ;
    displayMessage ("This is fun!", 3) ;
    return 0 ;
}
```

Output :-

```
Hello, world !
Hello, C++ !
This is fun !
This is fun !
This is fun !
```

10. What is a constructor ? Write down different types of constructor. List some of the special properties of the constructor functions.

Ans

- A constructor is a special member function of a class that is automatically called when an object of the class is created. The primary purpose of a constructor is to initialize the object's data member to specific values. Constructors have the same name as the class and do not have a return type, not even 'void'.

- The types of constructor are :

i). Default constructor,
ii). Parameterized constructor,
iii). Copy constructor,
iv). Move Constructor.

i). Default Constructor :-

- A constructor that takes no arguments. If no constructor is provided by the programmer, the compiler generates a default constructor.
- It initializes data members with default values.

Syntax :

```
Class MyClass {
  public :
      MyClass () {
          std :: cout << "Default constructor called !" << std :: endl ;
    }
}
```

ii). Parameterized Constructor :-

- A constructor that takes arguments and allows the initialization of objects with specific values at the time of creation.

```
class Myclass {
private :
    int x ;
public :
    Myclass (int val) {
        x = val ;
        std:: cout << "Parameterized Constructor called with
                    value :" << x << std:: endl ;
    }
}
```

iii). Copy Constructor :-

- A constructor that creates a new object as a copy of an existing object. It is called when an object is initialized with another object of the same class, passed by value to a function, or returned from a function. If not explicitly defined, the compiler provides a default copy constructor.

```
class Myclass {
  private :
    int x ;
  public :
    Myclass (int val) : x(val) {}
    Myclass (const Myclass & obj ) {
        x = obj . x ;
        std:: cout <<" Copy Constructor called !" <<std :: endl;
    }
};
```

iv). Move Constructor :-

- A move constructor is used to transfer resources from a temporary object to a new object, leaving the temporary object in a valid but unspecified state. It is useful for optimizing the performance when dealing with objects that manage dynamic memory or other resources.

```
class MyClass {
  private:
      int * ptr;
  public:
      MyClass (int val) : ptr( new int(val)) {}

      MyClass (MyClass && obj) noexcept {
          ptr = obj. ptr;
          obj. ptr = nullptr;
          std:: cout << "Move Constructor called!" << std:: endl;
    }
    ~MyClass () {
          delete ptr;
      }
  };
```

- The special properties of Constructors are :

i). No return Type,
ii). Automatic Invocation,
iii). Overloading,
iv). Cannot Be virtual,
v). Can call Other Constructors
vi). Inheritance
vii). Destruction Order
viii). Cannot Be static

11. Difference between constructor and destructor.

Ans

Feature	Constructor	Destructor
Purpose	Initializes an object of a class.	Cleans up before an object is destroyed.
Name	Has the same name as the class.	Has the same name as the class, preceded by a tilde ('~').
Return Type	No return type, not even 'void'.	No return type, not even 'void'.
Parameters	Can take parameters	Cannot take any parameters.
Invocation	Automatically called when an object is created.	Automatically called when an object goes out of scope or is explicitly deleted.
Overloading	Can be overloaded	Can not be overloaded
Default Constructor/ Destructor	If not provided, the compiler generates a default constructor.	If not provided, the compiler generates a default destructor.
Virtual	Cannot be virtual.	Can be declared as 'virtual' to ensure proper cleanup in polymorphic classes.

Execution Order	Called in the order of inheritance.	Called in reverse order of construction.
Static Members	Cannot be static.	Cannot be static.
Multiple cells	Can be called explicitly.	Cannot be called explicitly.
Resource Management	Used to allocate resources	Used to release resources.

12. What is class ? Write syntax of class. Also write syntax of member function define outside the class with example.

Ans

- A class in C++ is a user-defined data type that serves as a blueprint for creating objects. It encapsulates data and functions that operate on the data into a single unit. The class defines the properties and behaviours that the objects created from the class will have.

- Syntax of a class :

```
class ClassName {
  private :
     // Private data members and member functions
  public :
     // Public protected data members and functions
  protected :
     // Protected data members and member functions
};
```

- The syntax for defining the member function of a class outside the class definition is :

```
ReturnType ClassName :: FunctionName (Parameter List) {
    // Function Body
}
```

Example :-

```
# include <iostream>

class Rectangle {
  private :
     int width;
     int height;
```

```
    public :
        void setDimensions (int w, int h);
        int calculateArea();
};

void Rectangle :: setDimensions (int w, int h) {
    width = w;
    height = h;
}

int Rectangle :: calculateArea() {
    return width * height;
}

int main() {
    Rectangle rect;
    rect.setDimensions (5, 10);
    std :: cout << "Area :" << rect.calculateArea() << std :: endl;
    return 0;
}
```

Output :-

```
Area : 50
```

13. Explain Private, Public and Protected access modifiers in OOP.

Ans

- In Object-oriented programming (OOP), access modifiers define the level of access control for class members. They determine how and where the members of a class can be accessed. C++ provides three primary access modifiers:

i). Private access modifier,
ii). Public access modifier,
iii). Protected access modifier

i). Private access Modifiers :-

- Keyword : 'private'
- Scope :
  - i). Members declared as 'private' are accessible only within the class in which they are declared.
  - ii). They cannot be accessed directly from outside the class, including by objects of the class or functions not defined inside the class.
  - iii). Private members are also not accessible by derived classes.
- Purpose : Used to encapsulate and hide the internal implementation details of a class, ensuring that the class's internal state can only be modified by the class itself.

Syntax :

```
class ClassName {
  private :
      // Private data members
      // Private member functions
};
```

ii). Public access Modifier :-

- Keyword : 'public'
- Scope :
  i). Members declared as 'public' are accessible from anywhere in the program.
  ii). They can be accessed by objects of the class, functions, and even from derived classes.
- Purpose : Used for members that need to be accessible from outside the class, such as interface functions that allow interaction with the class private data.

Syntax :

```
class ClassName {
  public :
    // Public data members
    // Public member functions
};
```

iii). Protected access Modifier :-

- Keyword : 'protected'
- Scope :
  i). Members declared as 'protected' are accessible within the class and by derived classes.
  ii). They are not accessible from outside the class except through inheritance.
- Purpose : Use when members need to be accessible by derived classes but should not be exposed to the outside world.

Syntax :

```
class ClassName {
  protected :
    // Protected data members
    // Protected member functions
};
```

14. Explain the following object-oriented terms with example : Object, class, Attributes, and Operations.

Ans

- In Object-Oriented Programming (OOP), certain fundamental concepts form the basis of how programming are structured and executed. These include Object, class, Attributes, and operations. Let's understand this with an example :

i). Object :-

- An object is an instance of a class. It represents a real-world entity or concept, combining data and behaviour that can operate on the data. Objects can be thought of as the "things" in your program that holds specific information and can perform tasks.

Example :-

```
class Car {
  public :
      std :: string brand ;
      std :: string model ;
      int year ;

      void startEngine() {
          std :: cout << "Engine Started !" << std :: endl ;
      }
};
int main() {
   Car myCar ;
   myCar.brand = "Toyota" ;
   myCar.model = "Corolla" ;
   myCar.year = 2020 ;
   myCar.startEngine () ;
    return 0 ;
}
```

ii). Class :-

- A class is a blueprint or template for creating objects. It defines the attributes and operations that objects created from the class can have. Essentially, a class encapsulates the properties and behaviours of the objects it represents.

Example :-

```
class Car {
    public :
        std :: string brand ;
        std :: string model ;
        int year ;

        void startEngine() {
            std :: cout << "Engine started!" << std :: endl ;
    }
};
```

iii). Attributes :-

- Attributes are the characteristics or data that belong to an object. Attributes holds the state of an object and are typically variables within a class.

Example :-

```
class Car {
    public :
        std :: string brand ;
        std :: string model ;
        int year ;
};
```

iv). Operations :-

- Operations are the behaviours or actions that an object can perform . Operations are defined in the class and can manipulate the attributes or perform specific tasks related to the object.

Example :-

```
Class Car {
  public :
      Std :: String brand;
      Std :: string model;
      int year;

      Void StartEngine() {
          Std :: cout << "Engine Started !" << Std :: endl;
    }
};
```

15. Explain operator overloading. Describe unary and binary operator overloading with the help of example.

Ans

- Operator overloading is a feature in C++ that allows developers to redefine or "overload" most of the built-in operators to work with user-defined data types. This enables operators to perform operations on objects just as they do on built-in data types.

- Operators can generally be classified into two types based on the number of operands they work with :

i). Unary Operator
ii). Binary Operator

i). Unary Operator :-

- Unary operators works on a single operand. Common unary operators include '++', '--', '!', '-' etc.

Example :-

```
#include <iostream>
class Complex {
    private :
        int real ;
        int imag ;
public :
    Complex (int r = 0 , int i = 0) : real (r), imag(i) {}

    Complex operator - () {
            return Complex ( - real , - imag) ;
    }
    void display() const {
        Std:: Cout << real << "+" << imag << "i" << Std:: endl ;
    }
};
```

```
int main() {

    Complex c1(5,10);
    Complex c2

    std:: cout << "Original Complex Number :";
    c1.display();
    c2 = -c1;

    std :: cout << "Negative Complex Number :";
    c2.display();

    return 0;
}
```

Output :-

```
Original Complex Number : 5 + 10i
Negative Complex Number : -5 - 10i
```

ii). Binary Operator :-

- Binary operators work on two operands. Common binary operators include '+', '-', '*', '/', '==', '>', '<' etc.

Example :-

```
# include <iostream>

class Complex {
  private :
      int real ;
      int imag ;
  public :
      Complex (int r=0 , int i=0) : real(r) , imag(i) {}
```

```
    Complex operator + (const Complex& other) {
        return Complex (real + other.real, imag + other.imag);
    }

    void display() const {
        std::cout << real << "+" << imag << "i" << std::endl;
    }
};

int main() {
    Complex c1 (3,4);
    Complex c2 (5,6);
    Complex c3;

    std::cout << "First Complex Number:";
    c1.display();
    std::cout << "Second Complex Number:";
    c2.display();
    c3 = c1 + c2;
    std::cout << "Sum of Complex Numbers:";
    c3.display();

    return 0;
}
```

Output :-

```
First Complex Number : 3+4i
Second Complex Number : 5+6i
Sum of Complex Number : 8+10i
```

16. Define a class complex, having data members as x and Y, define a friend function sum() to add two complex numbers and display all numbers using show() friend function.

Ans

Input :-

```
#include <iostream>

class Complex {
   private :
       int x ;
       int y ;
   public :
       Complex ( int a =0, int b =0) : x(a), y(b) {}

       friend Complex sum ( const Complex & C1 , const Complex& C2);
       friend void show (const Complex & c );
};
 Complex sum (const Complex& C1 , const Complex& C2) {
       return Complex (C1.x + C2.x , C1.y + C2.y);
 }
 void show ( const Complex & c ) {
      Std :: cout << c.x << " + " << c.y << "i" << std :: endl;
 }
int main() {
    Complex C1 (3,4);
    Complex C2 (5,6);
    Complex C3 = sum ( C1 + C2);
    Std :: cout << "First Complex Number : " ;
    show (C1);
    Std :: cout << "Second Complex Number :";
    show(C2);
    Std :: cout << "Sum of Complex Number :";
    show (C3);
    return 0;
}
```

Output :-

```
First complex Number : 3 + 4i
Second complex Number : 5 + 6i
Sum of Complex Numbers : 8 + 10i
```

17. Define friend function. Create two classes DM and DB which store the value of distances. DM stores distances in meters and centimeters and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB. Use a friend function to carry out the addition operation. The object stores the result may a DM object or DB object, depending on the units in which the results are required. The display should be in the format of feet and inches or meters and centimeters depending on the object on display. 1 Feet = 0.3048 Meter, 1 Meter = 3.28 Feet, 1 Inch = 2.54 Centimeter, 1 centimeter = 0.3937 Inch.

Ans

Input :-

```
# include <iostream>

class DB ;
class DM {
   private :
        float meters ;
        float centimeters ;
   public :
        DM ( float m = 0 , float cm = 0) : meters (m) ,
                                           centimeters (cm) {}

        void display () const {
           std :: cout << meters << "meters " << centimeters <<
                                  "centimeters " << std :: endl ;
         }
        friend DM add (const DM& dm , const DB& db) ;
} ;

class DB {
     private :
```

```
    float feet;
    float inches;
public:
    DB(float ft=0, float in=0) : feet(ft), inches(in) {}

    void display() const {
        std::cout << feet << "Feet" << inches << "Inches" << std::
                                                    endl;
    }
    friend DM add(const DM& dm, const DB& db);
};

DM add(const DM& dm, const DB& db) {

    float totalMetersfromDB = db.feet * 0.3048;
    float totalCentimeterfromDB = db.inches * 2.54;

    totalMetersfromDB += (totalCentimetersFromDB / 100);

    float totalMeters = dm.meters + totalMetersfromDB;
    float totalCentimeters = dm.centimeters;

    if(totalCentimeters >= 100) {
        totalMeters += static_cast<int>(totalCentimeters) / 100;
        totalCentimeters = static_cast<int>(totalCentimeters) %
                                                    100;
    }
    return DM(totalMeters, totalCentimeters);
}
int main() {
    DM dm(2,50);
    DB db(5,10);
    DM result = add(dm,db);
    std::cout << "Result in meters and centimeters:";
    result.display();
    return 0;
}
```

Output :-

Result in meters and centimeters : 4.224 meters 0 centimeters