C++ OOP

KOSS

VAIBHAV JALANI
20EE10078
Section 9

What is OOP ?

OOP stands for **Object-Oriented Programming.**

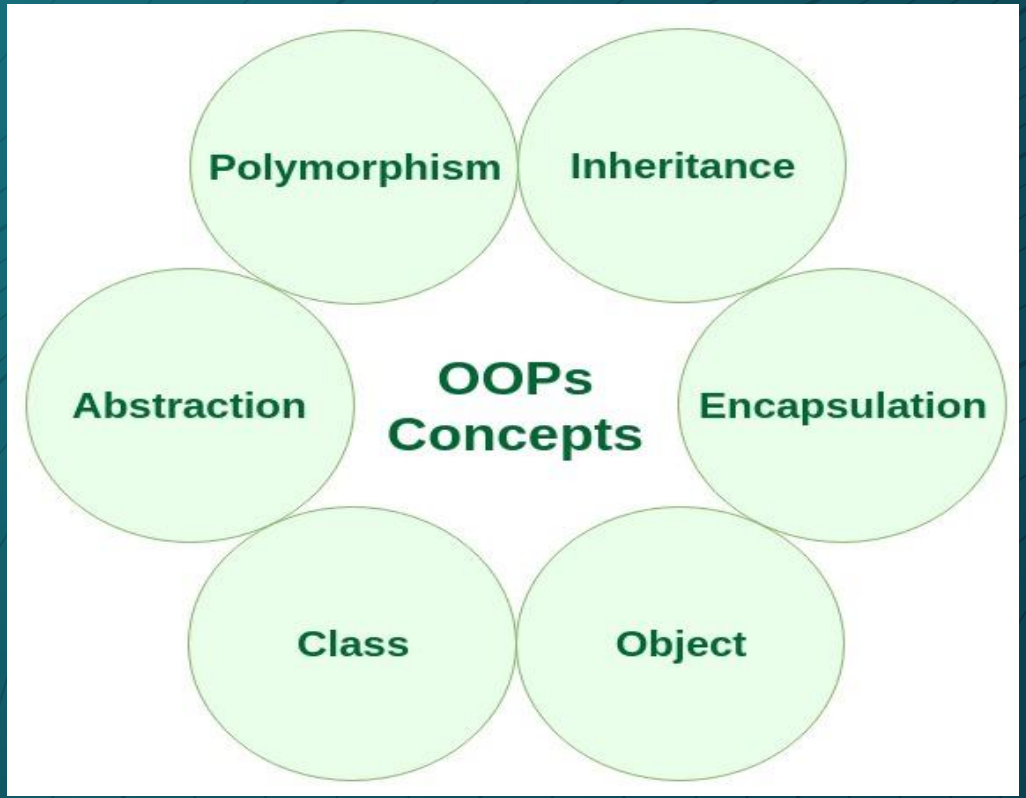
As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can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

It is faster and easier to execute and provides a clear structure for the programs. It makes the code easier to maintain, modify and debug.

Why OOP ?

- It is faster and easy to maintain. As a result it is more effective and the development time reduces.
- Program is divided into small parts called objects due to which adding new data and function is easy.
- In OOP, data is more important than function and thus it provides data hiding to make it more secure.
- New data objects can be created easily from existing objects making OOPs easy to modify.

6 characteristics of OOP



Classes & Objects

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 called object. A class is like a blueprint for an object. All the members of the class can be accessed through object. Object is a runtime entity, it is created at runtime.

```
class Student
{
  public:
    string name;
    string roll_no;
    int section;
};
```

```
int main()
{
    Student s1;
    s1.name =”Vaibhav Jalani";
    s1.roll_no = "20EE10078";
    s1.section = 9;
}
```

Classes vs Structures

Classes and structures are very similar in C++. Both can have data and function members. Both may declare any of their members private. Both of them can be inherited.

But there are differences also, which are as follows-

- By default, all the members of the structure are public. In contrast, all members of the class are private.
- Variables in a structure cannot be initialized during the declaration, but they can be done in a class.
- Class can have null values but the structure can not have null values.
- When a structure is implemented, memory allocates on a stack. In contrast, memory is allocated on the heap in class.

Encapsulation

Encapsulation is defined as binding together the data and the functions that manipulate them. Encapsulation also leads to data abstraction or hiding. It is used to make sure that sensitive data is hidden from users.

```
class Student {
    private:
        float cg ;

    public:
        void setcg(float c) {
                cg = c;
        }
        float getcg() {
                return cg;
        }
};
```

```
int main()
{
        Student myObj;
        myObj.setcg(9.7);
        cout << myObj.getcg();
        return 0;
}
```

Output:
9.7

Abstraction

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Hiding the internal details and showing functionality is known as abstraction.

- Abstraction using Classes
- Abstraction in Header files

Using header files:

```
int main()
{
  int n = 5;
  int power = 3;
  int result = pow(n,power);
  cout << "Answer is : " <<result<< endl;
  return 0;
}
```

Output:
Answer is : 125

Abstraction Using Classes:

```
class Sum    {
        private: int x, y, z;

        public:
                void add()
                {
                        cout<<"Enter two numbers: ";
                        x=3;
                        y=4;
                        z= x+y;
                        cout<<"Sum is: "<<z<<endl;
                }
};
```

```
int main()
{
        Sum new;
        new.add();
        return 0;
}
```

Output:
Sum is: 7

Polymorphism

Polymorphism as the ability of a message to be displayed in more than one form. When one task is performed by different ways it is called polymorphism. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

```
class Animal {
  public:
    void animalName() {
    cout << "The animal\n" ;
  }
};

class Pig : public Animal {
  public:
    void animalName() {
    cout << "The pig\n" ;
  }
};

class Dog : public Animal {
  public:
    void animalName() {
    cout << "The dog\n" ;
  }
};

int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;
  myAnimal.animalName();
  myPig.animalName();
  myDog.animalName();
}
```

Output:
The animal
The pig
The dog

Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance.

```
class Class1
{
    public:
        void myFunction()
        {
                cout << "Class 1" ;
        }
};

class Class2: public Class1 {
};
```

```
int main()
{
  Class2 myObj;
  myObj.myFunction();
  return 0;
}
```

Output:
Class1

There is another specifier named protected. This is like private, the only difference is that it can be used in inherited classes.

```
class Student{
    protected:
        float cg;
};

class Detail: public Student {
    public:
    int section;
    void setcg(float c) {
        cg = c;
    }
    floatt getcg() {
        return cg;
    }
};
```

```
int main()
{
        Detail myObj;
        myObj.setcg(9.7);
        myObj.section = 9;
        cout << "cg: " << myObj.getcg() << "\n";
        cout << "Section: " << myObj.section << "\n";
        return 0;
}
```

Output:
cg: 9.7
Section: 9

Overloading

Method overloading is the process of overloading the method that has the same name but different parameters. C++ allows to have more than one definition for a function or an operator, which is called function overloading and operator overloading respectively. When an overloaded function or operator is called, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions.

- Function Overloading:

```
class funcOver {
  public:
    void print(int i) {
      cout << "Int " << endl;
    }
    void print(float  f) {
      cout << "Float " << endl;
    }
};
```

```
int main(void) {
   funcOverpd;

   pd.print(8);
   pd.print(50.23);

   return 0;
}
```

Output:
Int
Float

OPERATOR OVERLOADING:

```
class rect {
   public:
      double area(void) {
        return length * breadth;
      }
      void setLength( double len ) {
        length = len;
      }
      void setBreadth( double bre ) {
        breadth = bre;
      }

      rect operator+(const rect& b) {
        rect r;
        r.length = this->length + b.length;
        r.breadth = this->breadth + b.breadth;
        return r;
      }

   private:
      double length;
      double breadth;
};
```

```
int main() {
   rect rect1;
   rect rect2;
   rect rect3;
   double area = 0.0;

   rect1.setLength(2.0);
   rect1.setBreadth(3.0);

   Box2.setLength(5.0);
   Box2.setBreadth(6.0);

  area = rect1.areae();
   cout << "Area of rect1 : " << area <<endl;

   area = rect2.area();
   cout << "Area of rect2 : " << area <<endl;

   rect3 = rect1 + rect2;

   area = rect3.areae();
   cout << "Area of rect3 : " << area <<endl;

   return 0;
}
```

Output

```
Area of rect1: 6.0
Area ot rect2: 30.0
Area of rect3: 63.0
```

Data Hiding & Access Specifiers

Data hiding is a technique specifically used in object-oriented programming (OOP) to hide internal object data. Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.

Access specifiers are used to implement data hiding.

- **Public:** All the class members declared under the public specifier will be available to everyone and can be accessed by other classes and functions too.
- **Private:** The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class.
- **Protected:** Protected is similar to private access specifier in the sense that it can't be accessed outside of it's class unless with the help of friend class. The difference is that the class members declared as Protected can be accessed by any derived class of that class as well.

Virtual Functions

A virtual function is a member function which is declared in the base class using the keyword virtual and is re-defined by the derived class.

```
class Shape{
public:
   void name(){
      cout << "Shape" << endl;
   }
};

class Square : public Shape{
public:
 int name(){
      cout << "Square " << endl
   }
};
```

```
int main(void)
{
   Shape* s;
   Square sq;
   s = &sq;
   s->name();

   return 0;
}
```

Output:
Shape

Abstract Classes

A pure virtual function is declared by assigning 0 in declaration. A class is abstract if it has at least one pure virtual function. A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, but we must override that function in the derived class, otherwise the derived class will also become abstract class

```
class Test{
   int x;
public:
    virtual void sample() = 0;
    int getX() { return x; }
};

int main(void){
    Test t;
    return 0;
}
```

Output:
Error

```
class Base{
public:
    virtual void sample() = 0;
};
class Derived: public Base{
public:
    void sample() { cout << "Derived \n"; }
};
int main(void){
    Base *b = new Derived();
    b->sample();
    return 0;
}
```

Output:
Derived

Constructors & Destructors

A constructor is a particular member function having the same name as the class name. It calls automatically whenever the object of the class is created.

Destructors have the same class name preceded by (~) tilde symbol. It removes and destroys the memory of the object, which the constructor allocated during the creation of an object.

The constructor which opens first, it closed at the last.

- Constructors may contain arguments while destructors can’t
- Constructors occupy memory while destructors don’t
- Constructors can be overloaded while destructor can’t
- Constructors have return types while destructors don’t
- Constructors can be used more than once, while destructors are used only once.

```
class c1 {
public:
  c1 () {
    cout<< "Constructor 1" <<endl;
  }
c1 (int a) {
    cout<< "Constructor 2" <<endl;
  }

  ~c1 () {
    cout << "Destructor" <<endl;
   }

  void display() {
    cout <<"Hello World!" <<endl;
   }
};
```

```
int main() {

   c1 obj1,obj2(4);

   obj1.display();
   obj2.display;

   return 0;
}
```

Output:
Constructor 1
Constructor 2
Hello World!
Hello World!
Destructor
Destructor

Garbage Collection

Garbage collection is the process in which programs try to free up memory space that is no longer used by objects. The garbage collector collects the memory which is occupied by variables or objects which are no more in use in the program.

Garbage collectors run implicitly as a program executes. Garbage collection is a form of automatic memory management. The opposite of automatic memory management is manual memory management, where memory is allocated and deallocated by the programmer.

Many modern OOP languages use garbage collection, including Java, Python, and C#. However some other languages like C++ and C (not OOP) do not have Garbage Collectors

THANK YOU