**C++ Object Oriented**

**Classes and Objects:**

The main purpose of C++ programming is to add object orientation to the C programming language. A class is used to describe a form of an object and contains data structs and functions to manipulate this data.

When you define a class, you define a blueprint for a data type.

access attributes of a class can be **public, protected, or public.**

A **public** member is accessible from anywhere outside the class but within a program.

A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

A **protected** member variable or function is very similar to a private member, but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

**Inheritance:**

Inheritance allows us to define a class in terms of another class. The existing class is called the **base** class, and the new class is referred to as the **derived** class.

A derived class can access all the non-private members of its base class.

* **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class but can be accessed through calls to the **public** and **protected** members of the base class.
* **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
* **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

**Overloading:**

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

When you call an overloaded **function** or **operator**, 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. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well (+, -, ()).

**Polymorphism:**

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

Meaning, I have the same function defined in the **base** class and in the **derived** class, so the compiler does not know which one to use. If nothing is said he will use the **base** class function.

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

**Data Abstraction:**

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction provides two important advantages −

* Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
* The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

**Data Encapsulation:**

Encapsulation is an Object-Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

Means that I can only manipulate the data using the classes function but I can’t directly access the class variables.

**Interfaces:**

An interface describes the behaviour or capabilities of a C++ class without committing to a implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error.

A class is made abstract by declaring at least one of its functions as **pure virtual** function.