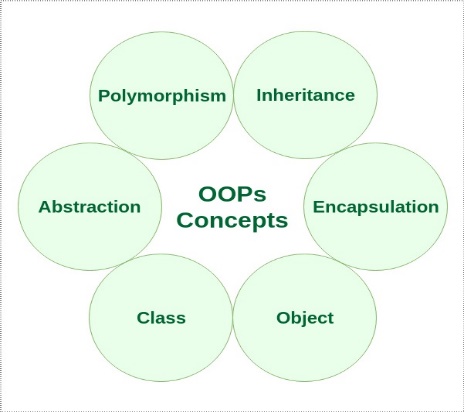
**OOPS**

* Object-oriented programming – As the name suggests uses objects 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.



**Class**

* A Class is a user-defined data type that has data members and member functions.
* 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.

**Object**

* An Object is an instance of a Class. When a class is defined, no memory is allocated but when an object is created memory is allocated.

**Encapsulation**

* Encapsulation is defined as wrapping up data and information under a single unit.
* Encapsulation is defined as binding together the data and the functions that manipulate them.
* Encapsulation also leads to data abstraction or data hiding.
* **Fully encapsulated class** – all data members are private.
* **ADV** – data hiding (security), code reusability, helps in unit testing

**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.

**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.

Two types of polymorphism:

* **Compile time Polymorphism or Static polymorphism**

Compile time polymorphism is a feature of object-oriented programming languages that allows you to define multiple methods with the same name but with different parameters. The compiler determines which method to call based on the method signature during compile time. Compile time polymorphism is also known as static polymorphism or early binding**.**

* + **Operator Overloading:** The process of making an operator exhibit different behaviours in different instances is known as operator overloading.

**\* , :**: , ?. , . operators cannot be overloaded.

* + **Function Overloading:** Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance. Functions that differ in return type cannot be overloaded.

**Class B{**

**Void operator+(Class\_B &obj)**

**{**

**//**

**}**

**}**

**// calling**

**B Ob1,Ob2;**

**Ob1 + Ob2;**

* **Runtime Polymorphism or dynamic polymorphism**

| **Virtual function** | **Pure virtual function** |
| --- | --- |
| **A virtual function is a member function of base class which can be redefined by derived class.** | **A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.** |
| **Classes having virtual functions are not abstract.** | **Base class containing pure virtual function becomes abstract.** |
| **Syntax:**   |  | | --- | | **virtual<func\_type><func\_name>()**  **{**  **// code**  **}** | | **Syntax:**   |  | | --- | | **virtual<func\_type><func\_name>()**  **= 0;** | |
| **Definition is given in base class.** | **No definition is given in base class.** |
| **Base class having virtual function can be instantiated i.e. its object can be made.** | **Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.** |
| **If derived class do not redefine virtual function of base class, then it does not affect compilation.** | **If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.** |
| **All derived class may or may not redefine virtual function of base class.** | **All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.** |

* **Virtual Function / Operator Overriding**
  + A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class. It tells the compiler to perform late binding where the compiler matches the object with the right called function and executes it during the runtime. This technique falls under Runtime Polymorphism.
  + Function overriding provides you with a way to override an existing functionality of a class inside a particular derived class. This can be useful when a child class requires its own version of a functionality

**Access Modifiers**

* **Public:** Public functions and variables can be accessed outside the class.
* **Protected:**  Can only be accessed within the class and can be inherited.
* **Private:** Default access modifier if not explicitly defined. Private variables and functions can only be accessed inside the class. Cannot be inherited.

**Inheritance**

* The capability of a class to derive properties and characteristics from another class is called Inheritance.
* **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
* **Super Class:** The class whose properties are inherited by a sub-class is called Base Class or Super class.

**Reusability:** Inheritance supports the concept of “reusability,” i.e., when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Private data member of any class cannot be inherited.

Types Of Inheritance: -

* Single inheritance
* Multilevel inheritance
* Multiple inheritance
* Hierarchical inheritance
* Hybrid inheritance

**Single Inheritance:** In single inheritance, a class is allowed to inherit from only one class. i.e., one subclass is inherited by one base class only.

**Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e., one subclass is inherited from more than one base class.

**Multilevel Inheritance:** In this type of inheritance, a derived class is created from another derived class.

**Hierarchical Inheritance:** In this type of inheritance, more than one subclass is inherited from a single base class. i.e., more than one derived class is created from a single base class. Combination of multiple and multilevel inheritance.

**Hybrid (Virtual) Inheritance:** Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

**Inheritance Ambiguity**

In multiple inheritances, when one class is derived from two or more base classes then there may be a possibility that the base classes have functions with the same name, and the derived class may not have functions with that name as those of its base classes. If the derived class object needs to access one of the similarly named member functions of the base classes, then it results in ambiguity because the compiler gets confused about which base’s class member function should be called.

**Message Passing**

Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent.

**Getters / Setters**

Public functions that return / set the value of private variables in a class.

**Structure Padding**

Structure padding is defined as the process of adding one or more empty bytes between the different data types to align data in memory. Structure padding increases memory consumption but reduces CPU cycles.

**Data Alignment**

Data alignment means putting the data in memory at an address equal to some multiple of the word size. This increases the performance of the system due to the way the CPU handles memory.

**Constructors**

* It is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally.
* Default constructors take no parameters whereas parameterized constructors take parameters they also are used to overload constructors.
* Constructor is a member function of a class, whose name is same as the class name.
* Constructor do not return value; hence they do not have a return type.

**Copy constructor**

* Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
* Copy constructor takes a reference to an object of the same class as an argument.
* The process of initializing members of an object through a copy constructor is known as copy initialization.
* Object should be passed by reference in copy constructor.

**Destructor**

* Destructor is a member function that is invoked automatically whenever an object is going to be destroyed.
* A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
* Destructor has the same name as their class name preceded by a tilde (~) symbol.
* It is not possible to define more than one destructor.
* The destructor is the only way to destroy the object created by the constructor. Hence destructor cannot be overloaded.
* Destructor neither requires any argument nor returns any value.
* It is automatically called when an object goes out of scope.
* Destructor release memory space occupied by the objects created by the constructor.
* In destructor, objects are destroyed in the reverse of object creation.

The destructor function is automatically invoked when the objects are destroyed.

* It cannot be declared static or const.
* The destructor does not have arguments.
* It has no return type not even void.
* A destructor should be declared in the public section of the class.
* The programmer cannot access the address of the destructor.
* Destructors can be made virtual.
* Can be called explicitly. object\_name. ~class\_name() or delete object\_name
* Destructor is automatically called for objects created statically.

**Const**

Whenever const keyword is attached with any method (), variable, pointer variable, and with the object of a class it prevents that specific object/method ()/variable to modify its data items value.

* **Constant variables**
  + Cannot be left uninitialized at the time of the assignment.
  + It cannot be assigned value anywhere in the program.
  + Explicit value needed to be provided to the constant variable at the time of declaration of the constant variable.
* **Constant Methods**
  + An object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object.

**Inside a class**

class

{

void fun () const

{ }

}

**Ordinary Function**

const void foo ()

{

}

**Static**

* Creates a data member that belongs to the whole class and not the object.
* Can be accessed without creating an object
* Initialize outside class Data\_type class\_name::member\_name = 100;
* Can be modified easily.
* Should be public
* Don’t have this keyword (pointer to object)
* Static functions can only access static member functions i.e cannot use other public or private data members.

**Process Memory Structure**

Int a = 10; (Stored in stack)

Int \*a = new int (10); (value is stored in heap and the pointer to the location in heap is stored in stack)

Memory in stack is automatically deleted when a variable goes out of scope but this is not the case for heap. Heap memory is not deleted until specifically asked. This causes a memory leak. That is why smart pointers are required.

RAII – Resource Acquisition in Initialization

Types of Smart Pointers:

* Shared PTR – shared ownership of resource
* Unique PTR – Exclusive ownership of resource
* Auto PTR (deprecated)
* Weak PTR (special case of shared PTR)