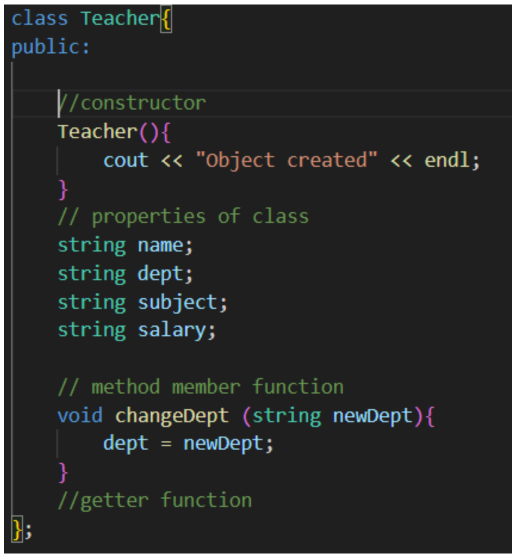
OBJECT ORIENTED PROGRAMMING

- **Object-Oriented Programming** is a methodology to design program that will easy to develop and maintenance using class and object.

- **Class** is a user-defined data type which defines its properties and its functions. Class is the only logical representation of the data. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.

  C++ Syntax (for class) :

  1. Class name should be in CamelCase (common convention).
  2. Constructor name should be same as class name.
  3. Object are entities in the real world and class are the blue print for this entities.



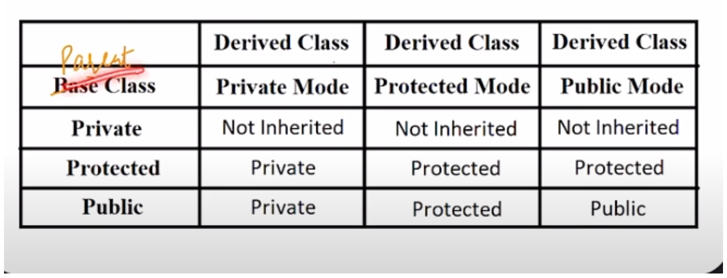
```
class Teacher{
public:

    //constructor
    Teacher(){
        cout << "Object created" << endl;
    }
    // properties of class
    string name;
    string dept;
    string subject;
    string salary;

    // method member function
    void changeDept (string newDept){
        dept = newDept;
    }
    //getter function
};
```

- **Object** is a run-time entity. It is an instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions.

  C++ Syntax (for object):

  ```
  student s = new student();
  ```

  Note : When an object is created **using** a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created **without** a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.



~~Base~~ Parent Class	Derived Class Private Mode	Derived Class Protected Mode	Derived Class Public Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

- **Inheritance**

  Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can **reuse, extend or modify** the attributes and behaviors which are defined in other classes.

  In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

  C++ Syntax :

  class derived_class :: visibility-mode base_class;

  visibility-modes = {private, protected, public}

  Types of Inheritance :

  1. Single inheritance : When one class inherits another class, it is known as single level inheritance
  2. Multiple inheritance : Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.
  3. Hierarchical inheritance : Hierarchical inheritance is defined as the process of deriving more than one class from a base class.
  4. Multilevel inheritance : Multilevel inheritance is a process of deriving a class from another derived class.
  5. Hybrid inheritance : Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

- **Encapsulation**

  Encapsulation is the process of combining data and functions into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. (**Data hiding**: a language feature to restrict access to members of an object, reducing the negative effect due to dependencies. e.g. "protected", "private" feature in C++).

- **Abstraction**

  We try to obtain an **abstract view**, model or structure of a real life problem, and reduce its unnecessary details. With definition of properties of problems, including the data which are affected and the operations which are identified, the model abstracted from problems can be a standard solution to this type of problems. It is an efficient way since there are nebulous real-life problems that have similar properties.

<u>Data binding</u> **:** Data binding is a process of binding the application UI and business logic. Any change made in the business logic will reflect directly to the application UI.

- **Polymorphism**

  Polymorphism is the ability to present the same interface for differing underlying forms (data types). With polymorphism, each of these classes will have different underlying data. A point shape needs only two coordinates (assuming it's in a two-dimensional space of course). A circle needs a center and radius. A square or rectangle needs two coordinates for the top left and bottom right corners and (possibly) a rotation. An irregular polygon needs a series of lines. Precisely, Poly means 'many' and morphism means 'forms'.

  <u>Types of Polymorphism</u> IMP

  1. Compile Time Polymorphism (Static)
  2. Runtime Polymorphism (Dynamic)

Let's understand them one by one :

- **<u>Compile Time Polymorphism</u>** : The polymorphism which is implemented at the compile time is known as compile-time polymorphism. Example - Method Overloading

Method Overloading : Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

1. The return type of the overloaded function.
2. The type of the parameters passed to the function.
3. The number of parameters passed to the function.

Example :

```
#include<bits/stdc++.h>
using namespace std;

class Add {
    public:
            int add(int a,int b){
                    return (a + b);
            }
            int add(int a,int b,int c){
                    return (a + b + c);
            }
};
int main(){
     Add obj;
     int res1,res2;
     res1 = obj.add(2,3);
     res2 = obj.add(2,3,4);
     cout << res1 << " " << res2 << endl;
     return 0;
}

/*
Output : 5 9
add() is an overloaded function with a different number of parameters. */
```

- **Runtime Polymorphism** : Runtime polymorphism is also known as **dynamic polymorphism**. Function overriding is an example of runtime polymorphism. Function overriding means when the child class contains the method which is already present in the parent class. Hence, **the child class overrides the method of the parent class**. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.

C++ Sample Code :

```
#include <bits/stdc++.h>
using namespace std;

class Base_class{
    public:
      virtual void show(){
            cout << "Apni Kaksha base" << endl;
      }
};

class Derived_class : public Base_class{
    public:
            void show(){
                  cout << "Apni Kaksha derived" << endl;
            }
};

int main(){
    Base_class* b;
    Derived_class d;
    b = &d;
    b->show(); // prints the content of show() declared in derived
    class return 0;
}

// Output : Apni Kaksha derived
```

- **Constructor** : Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as class or structure also constructor don't have any return type.

  There can be **two types** of constructors in C++.

  1. Default constructor : A constructor which has no argument is known as default constructor. It is invoked at the time of creating an object.
  2. Parameterized constructor : Constructor which has parameters is called a parameterized constructor. It is used to provide different values to distinct objects.
  3. Copy Constructor : A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object. It is of two types – default copy constructor and user defined copy constructor.

C++ Sample Code :

```
#include <bits/stdc++.h>
using namespace std;

class go {
        public:
                int x;
                go(int a){ // parameterized constructor.
                x=a;
            }
            go(go &i){ // copy constructor
                x = i.x;
            }
};
int main(){
        go a1(20); // Calling the parameterized constructor.
        go a2(a1); // Calling the copy constructor.
        cout << a2.x << endl;
        return 0;
}

// Output : 20
```

- **Destructor** : A destructor works opposite to constructor; it destructs the objects of classes. It can be defined **only once** in a class. Like constructors, it is invoked automatically. A destructor is defined like a constructor. It must have the same name as class, prefixed with a **tilde sign (~)**.

  Example :

```
#include<bits/stdc++.h>
using namespace std;

class A{
    public:
                // constructor and destructor are called automatically,
once the object is instantiated
           A(){
                cout << "Constructor in use" << endl;
           }
           ~A(){
                cout << "Destructor in use" << endl;
           }
};
int main(){
        A a;
        A b;
        return 0;
}
/*
Output: Constructor in use
                Constructor in use
                Destructor in use
                Destructor in use
*/
```

- **<u>'this' Pointer</u> : this** is a keyword that refers to the **current instance of the class.** There can be 3 main uses of 'this' keyword:

    1. It can be used **to pass the current object as a parameter to another method**

    2. It can be used **to refer to the current class instance variable.**

    3. It can be used **to declare indexers.**

<u>C++ Syntax</u> :

```
struct node{
        int data;
        node *next;

        node(int x){
                this->data = x;
                this->next = NULL;
        }
}
```

- **<u>Friend Function</u>** : Friend function acts as a friend of the class. **It can access the private and protected members of the class.** The friend function is not a member of the class, but it must be listed in the class definition. The non-member function cannot access the private data of the class. Sometimes, it is necessary for the non-member function to access the data. **The friend function is a non-member function and has the ability to access the private data of the class.**

Note :

1. A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.
2. Friend function uses objects as arguments.

Example IMP :

```
#include <bits/stdc++.h>
using namespace std;

class A{
        int a = 2;
        int b = 4;
        public:
                // friend function
                friend int mul(A k){
                return (k.a * k.b);
                }
};

int main(){
        A obj;
        int res = mul(obj);
        cout << res << endl;
        return 0;
}

// Output : 8
```

- **Aggregation :** It is a process in which one class defines another class as any entity reference. **It is another way to reuse the class**. It is a form of association that represents the HAS-A relationship.

- **Virtual Function IMP**: A virtual function is used to replace the implementation provided by the base class. The replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer.

  1. **A virtual function is a member function which is present in the base class and redefined by the derived class.**
  2. When we use the same function name in both base and derived class, **the function in base class is declared with a keyword virtual.**
  3. When the function is made virtual, then C++ determines at run-time which function is to be called based on the type of the object pointed by the base class
     pointer. **Thus, by making the base class pointer to point to different objects, we can execute different versions of the virtual functions.**

Key Points :

1. Virtual functions cannot be static.
2. A class may have a virtual destructor but it cannot have a virtual constructor.

C++ Example :

```
#include <bits/stdc++.h>
```

```
using namespace std;

class base {
    public:
        // virtual function (re-defined in the derived class)
        virtual void print(){
            cout << "print base class" << endl;
        }

        void show(){
            cout << "show base class" << endl;
        }
};

class derived : public base {
    public:
        void print(){
            cout << "print derived class" << endl;
        }

        void show(){
            cout << "show derived class" << endl;
        }
};

int main(){
    base* bptr;
    derived d;
    bptr = &d;

    // virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}

/*
output :
```

```
print derived class // (impact of virtual function)
show base class
*/
```

- <u>Pure Virtual Function</u> :
    1. A pure virtual function is not used for performing any task. It only serves as a placeholder.
    2. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
    3. A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as **abstract base classes.**
    4. The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

<u>C++ Syntax</u> :

```
virtual void display() = 0;
```

<u>C++ Example :</u>

```
#include<bits/stdc++.h>
using namespace std;

class Base{
    public:
        virtual void show() = 0;
};
class Derived : public Base {
    public:
            void show() {
                cout << "You can see me !" << endl;
            }
};
int main(){
    Base *bptr;
```

```
        Derived d;
        bptr = &d;
        bptr->show();
        return 0;
}

// output : You can see me !
```

- **Abstract Classes** : In C++ class is made abstract by declaring at least one of its functions as a **pure virtual function.** A pure virtual function is specified by placing "= 0" in its declaration. **Its implementation must be provided by derived classes.**

  Example :

```
#include<bits/stdc++.h>
using namespace std;

// abstract class
class Shape{
        public:
            virtual void draw()=0;
};
class Rectangle : Shape{
        public:
                void draw(){
                        cout << "Rectangle" << endl;
                }
};
class Square : Shape{
        public:
                void draw(){
                        cout << "Square" << endl;
                }
};

int main(){
        Rectangle rec;
        Square sq;
        rec.draw();
        sq.draw();
        return 0;
}

/*
Output :
Rectangle
Square
*/
```

- **Namespaces in C++** :
    1. The namespace is a logical division of the code which is designed to stop the naming conflict.
    2. The namespace defines the scope where the identifiers such as variables, class, functions are declared.
    3. **The main purpose of using namespace in C++ is to remove the ambiguity.** Ambiguity occurs when a different task occurs with the same name.
    4. For example: if there are two functions with the same name such as add(). In order to prevent this ambiguity, the namespace is used. Functions are declared in different namespaces.
    5. C++ consists of a standard namespace, i.e., std which contains inbuilt classes and functions. So, by using the statement "using namespace std;" includes the namespace "std" in our program.

C++ Example :

```
#include <bits/stdc++.h>
using namespace std;

// user-defined namespace
namespace Add {
        int a = 5, b = 5;
            int add() {
                return (a + b);
        }
}
```

```
int main(){
        int res = Add :: add(); // accessing the function inside namespace
        cout << res;
}

// output : 10
```

- **Access Specifiers** **IMP** **:** The access specifiers are used to define how functions and variables can be accessed outside the class. There are three types of access specifiers:

1. **Private**: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared by default both properties and member functions are **Private**.
2. **Public**: Functions and variables declared under public can be accessed from anywhere.
3. **Protected**: Functions and variables declared as protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance.

Key Notes

- **Delete** is used to release a unit of memory, **delete[]** is used to release an array.
- **Virtual inheritance** facilitates you to create only one copy of each object even if the object appears more than one in the hierarchy.
- **Function overloading:** Function overloading is defined as we can have more than one version of the same function. The versions of a function will have different signatures meaning that they have a different set of parameters.

Operator overloading: Operator overloading is defined as the standard operator can be redefined so that it has a different meaning when applied to the instances of a class.

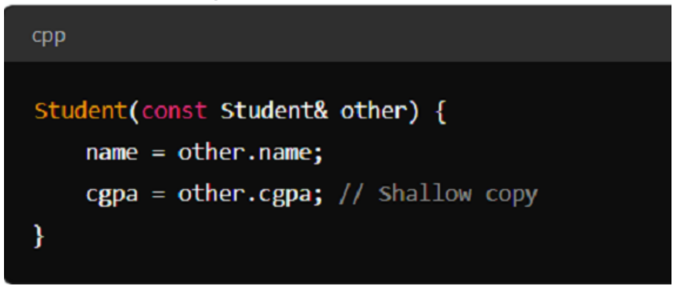
- **Overloading** is static Binding, whereas Overriding is dynamic Binding. Overloading is nothing but the same method with different arguments, and it may or may not return the same value in the same class itself. **Overriding** is the same method name with the same arguments and return types associated with the class and its child class.

Shallow Copy

A shallow copy creates a new object and then copies the non-static fields of the current object to the new object. If the field is a reference to an object, the reference is copied, meaning both the original and the copy will point to the same object in memory. This can lead to unexpected behavior if the shared object is modified from either reference.

Example:

In the Student class example provided, the shallow copy constructor simply copies the pointer cgpa from the original Student object to the new one. Thus, both objects share the same memory address for cgpa.



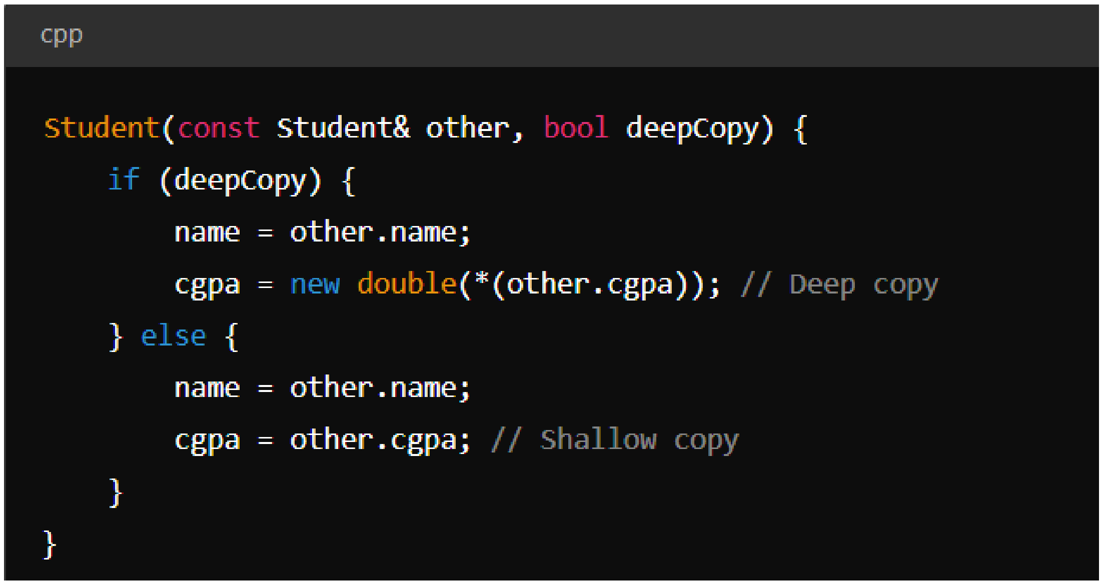
```cpp
Student(const Student& other) {
    name = other.name;
    cgpa = other.cgpa; // Shallow copy
}
```

Key Point:

1. Fast and memory efficient.
2. Risk of unintended side effects due to shared references.

Deep Copy
A deep copy creates a new object and then copies all fields, creating duplicates of dynamically allocated memory that the fields reference. This means the new object and the original object do not share the same memory addresses for their references, thus modifications to one do not affect the other.

Example:
In the Student class example, the deep copy constructor allocates new memory and copies the value pointed to by cgpa from the original object.



cpp

```cpp
Student(const Student& other, bool deepCopy) {
    if (deepCopy) {
        name = other.name;
        cgpa = new double(*(other.cgpa)); // Deep copy
    } else {
        name = other.name;
        cgpa = other.cgpa; // Shallow copy
    }
}
```

Key Point:

1. Ensures complete independence between the original and copied objects.
2. More resource-intensive compared to a shallow copy due to additional memory allocation.

https://www.geeksforgeeks.org/oops-interview-questions/

### Abstraction:

****Using Abstract Classes****

Abstract classes are used to provide a base class from which other classes can be derived. They cannot be instantiated and are meant to be inherited. Abstract classes are typically used to define an interface for derived classes.

Read about it what is use of it .....