OOPS-Object Oriented Programming Structure(C++)

Why do we need it?

To solve complex problems and to be able to represent real life entity using class.

Class are the building blocks of oops

Class is a more complex data structure.

Class contains attributes and methods and behviour. Class represents a blue print from which objefs can be created.

Simple example:

```
#include<iostream>

Using std::string;

Class Employee{

Public:

   String Name;

 String Company;

  Int Age;

Void intro(){

Std::cout<<"Name is"<<Name<<"working in"<<Company<<std::endl;

     }

}

Int main(){

Employee employee1;

employee1.Name="yutika";

employee.Company="Morgan Stanley";

employee.Age=21;

employee1.intro();

}
```

Acess Modifiers:

1) Private: Can’t be accessed outside the class.
2) Public: Can be accessed outside the class.
3) Protected: it is inherited by its child classes.

Constructors: It is a special type of method.

Three rules of a constructor:

1) It doesn’t have a return type.
2) It has the same name as that of a class.
3) It is public but there are private constructors too.

Example of a constructor: Referring to above problem.

```
Employee(string name, string company, int Age){

Name=name;

Company = company;

Age=age;

}
```

Four Pillars of OOPs:

1) **Encapsulation :** Bundling data and methods together.
   Bundling is done to prevent others to access and modify the attributes and behaviour.
   We can access encapsulated classes through it’s methods(public methods).
   Example:

   ```
    #include <iostream>
   #include <string>

   class Student {
   private:
       std::string name;
       int age;
   ```

```
public:
    Student(const std::string& n, int a) : name(n), age(a) {}


    std::string getName() {
        return name;
    }

    int getAge() {
        return age;
    }

    void setName(const std::string& n) {
        name = n;
    }

    void setAge(int a) {
        if (a >= 0) {
            age = a;
        }
    }
};

int main() {
    Student student("John", 20);


    std::cout << "Name: " << student.getName() << std::endl;
    std::cout << "Age: " << student.getAge() << std::endl;

    // Modify encapsulated attributes using setter methods
    student.setName("Alice");
    student.setAge(22);

    std::cout << "Updated Name: " << student.getName() << std::endl;
    std::cout << "Updated Age: " << student.getAge() << std::endl;
    return 0;
}
```

2) **Abstraction :** Hiding a complex thing behind a procedure to make things simpler.

Interface concepts are implemented by using abstract class.

Example:

```
#include <iostream>

class Shape {
public:
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    void draw() {
        std::cout << "Drawing a Circle" << std::endl;
    }
};

class Square : public Shape {
public:
    void draw() {
        std::cout << "Drawing a Square" << std::endl;
    }
};

int main() {
    Circle circle;
    Square square;

    Shape* shapes[] = {&circle, &square};

    for (Shape* shape : shapes) {
        shape->draw();
    }
    return 0;
}
```

3) **Inheritance:** Base class(super/parent) is inherited by child(derived/sub) class and child class can have extra attributes and methods as well.
   Ex: class Developer:Employee{
   }

   Child Parent

Protected members of base class can be inherited by all it’s child classes.

Example:

```
#include <iostream>

#include <string>

class Person {

public:

    Person(const std::string& name, int age)

        : name(name), age(age) {}


    void introduce() {

        std::cout << "Name: " << name << ", Age: " << age << std::endl;

    }


private:

    std::string name;

    int age;

};
```

```
class Student : public Person {
public:
    Student(const std::string& name, int age, const std::string& school)
        : Person(name, age), school(school) {}

    void study() {
        std::cout << name << " is studying at " << school << std::endl;
    }

private:
    std::string school;
};

int main() {
    Student student("Alice", 20, "XYZ University");

    student.introduce();
    student.study();

    return 0;
}
```

4) **Polymorphism:** Greek origin, meaning multiple forms. It means ability of an object/method to have many forms. Parent class reference is used to refer to an object of child class.

```
Employee* e1 = &d;
    ↓          ↓
```

Base class derived class object

Pointer

Example:

```
#include <iostream>
class Animal {
public:
   Animal(const std::string& name) : name(name) {}
   virtual void speak() {
      std::cout << name << " makes some generic animal sound." << std::endl;
   }

private:
   std::string name;
};
class Dog : public Animal {
public:
   Dog(const std::string& name) : Animal(name) {}

   // Override the speak method
   void speak() override {
      std::cout << name << " says Woof!" << std::endl;
   }
};

class Cat : public Animal {
```

```
public:
    Cat(const std::string& name) : Animal(name) {}
    void speak() override {
        std::cout << name << " says Meow!" << std::endl;
    }
};

int main() {
    Animal* animals[] = {new Dog("Buddy"), new Cat("Whiskers")};

    for (Animal* animal : animals) {
        animal->speak();
    }

    return 0;
}
```