class & objects

class

Fruit

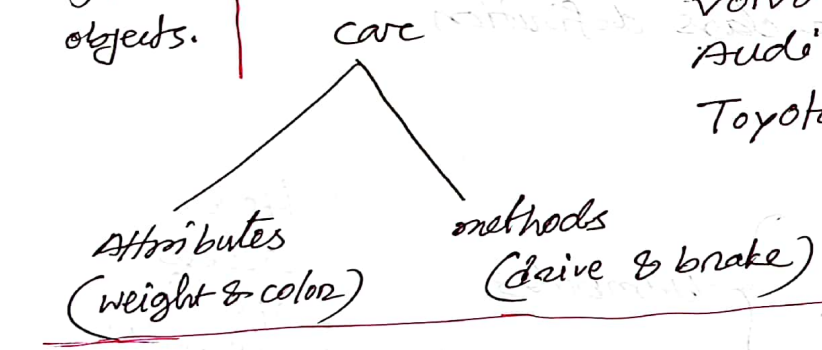
A class is a template for objects.

objects

Apple
Banana
Mango

Volvo
Audi
Toyota

An object is an instance of a class.




```
class Car{
  public:
    string brand;
    string model;
    int year;
    void display(){
      cout<< "Nice Car";
    }
};
```

```
int main(){
  Car myObj;
  myObj.brand = "BMW";
  myObj.model = "X5";
  myObj.year = 1996;

cout<< myObj.brand <<endl;
cout<< myObj.model <<endl;
cout<< myObj.year <<endl;
cout<< myObj.display() <<endl;
return 0 }
```

class is a user-defined datatype. It has its own data members and member functions which are used by creating an instance of the class.

```
class Car{
    string brand;
    string model;
    int year;
public:
    int speed(int maxspeed); }
};
```

Attributes

methods

By default the access specifier of class is private

// Inside class definition

```
class myClass{
    public:
        int id;
        string name;
        void display(){
            cout << "SAU" << endl; }
        }
};
```

Attributes

method

Here, access specifier is public

// outside class definition

```
class myClass{
  public:
    void myMethod();
};
void myClass::myMethod(){
    cout<<"gfg";
}
```

Access Specifiers

Access specifiers define how the members (attributes and methods) of a class can be accessed.

public - members are accessible from outside the class.

private - " cannot be " " " " " "

protected - " " " " " " " but they can be accessed in inherited classes.

An object is an instance of a class that have state and behaviour.

syntax: class_name object_name;

ex : Car myObj;

```
#include<iostream>
using namespace std;
class Print{                    // create class
   public:
      void display(){
         cout<<" South Asian University ";
         }
};
int main(){
   Print myObj;                 // create object
   myObj.display();
   return 0;
}
```

//output

South Asian University

```
#include <iostream>
using namespace std;
class Person {          // create class
    private:
        int run;
        string msg;
    public:
        void play (){
            run = 30;
            cout << "Today I scored " << run << " runs" << endl;
        }
        void walk (){
            msg = "Today I walked 3 kilometers";
            cout << msg;
        }
};
int main(){
    Person myObj;       // create object
    myObj.play();
    myObj.walk();
    return 0;
}
```

Constructor

A constructor is a special method that is automatically called when an object of a class is created.

The main purpose of constructor is used to initialize the object.

Types
- default
- parameterized
- copy
- move

syntax

```
class A{
  public:
    A(){
    }
};
```

{ class name and constructor name should be same

```
class myClass {
    public:
        myClass () {          // constructor
            cout << "gk";
        }
};
int main() {
    myClass myObj;
    return 0;
}
```

Here, ~~use~~ No need to call constructor

Default constructor

A constructor with no parameters is known as a default constructor.

```
class Wall {
    private:
        double length;
    public:
        Wall () {    // default constructor
            length = 5.5;
            cout << "The length of wall is: " << length << endl;
        }
};
```

```
int main() {
    Wall myObj;
    return 0;
}
```

parameterized constructor

A constructor with parameters is known as a parameterized constructor.

```
class Wall{
  private:
    double length;
    double height;
  public:
    Wall(double len, double hgt){   // parameterized constructor
        length = len;
        height = hgt;
    }
    double Area(){
        return length * height;
    }
};
int main(){
    Wall myObj-1(10.5, 8.6);
    Wall myObj-2(8.5, 6.3);
    cout<<" Area of wall 1: "<<myObj-1.Area()<<endl;
    cout<<" Area of wall 2: "<<myObj-2.Area();
    return 0;
}
```

copy constructor

The copy constructor is used to copy data from one object to another.

```
class Wall{
  private:
    double length;
    double height;
  public:
    Wall(double len, double hgt){
        length = len;
        height = hgt;
    }

    Wall (Wall &obj){          // copy constructor
        length = obj.length;
        height = obj.height;
    }
    double Area(){
        return length * height;
    }
};
```

```
int main(){
  Wall obj-1 (10.5, 8.6);
  Wall obj-2 = obj-1;
  cout<< obj-1.Area()<<endl;
  cout<<obj-2.Area();
  return 0;
}
```

Destructor

Destructor is also a special type of member function that is used to de-allocate the memory, allocated by the constructor.

Syntax

```
class A{
  public:
    A{
    }
    ~A{
    }
};
```

```
int count=0;
class A {
  public:
    A() {
      cout<< "object " <<++count<< "created";
    }
    ~A(){
      cout<< "object"<<count--<< "deleted";
    }
};
int main(){
  A obj1, obj2, obj3;
  return 0;
}
```

Abstraction

Data abstraction is a technique by which only necessary data is shown to the user and unnecessary data is hidden.

```
class myBank{
  private:
     int atmPIN, Balance;
  public:
     string bName, IFSC;
     int accNumber;
   void input(){
      atmPIN=1234;
      Balance = 67500;
      bName = "EBL";
      IFSC = "ebl012345";
      accNumber = 23416793;
   }
   void output(){
      cout<<atmPIN<<endl;
      cout<< Balance<<endl;
      cout<< bName<<endl;
      cout<< IFSC <<endl;
      cout<< accNumber <<endl;
   }
};
```

```
int main(){
  myBank obj;
  obj.input();
  obj.output();
  return 0;
}
```

Output

```
1234
67500
EBL
ebl012345
23416793
```

```
class gkAbstraction {
  private :
     int a, b;
  public:
     void set (int x, int y){    // private members
        a = x;
        b = y;
     }
     void display (){
        cout << a << endl;
        cout << b << endl;
     }
};
int main() {
     gkAbstraction obj;
     obj. set (10, 20);
     obj. display ();
     return 0;
}
```

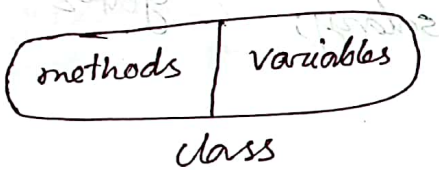
Here, we can not access the variables a & b directly.

output

```
a = 10
b = 20
```

Encapsulation:

Encapsulation is a technique that is used to hide the sensitive data from users. It is a good practice that increases security of data.




```
class Student{
  private:
    int age;

  public:
    void setData(int studentAge)
    {
      age = studentAge;
    }
    int getData(){
      return age;
    }
};
```

Data hiding

Setter

Getter

get & set methods are used to read or modify private data.

```
class Employee{
  private:
    int salary;
  public:
    void setSalary(int s){
      salary = s;
    }
    int getSalary(){
      return salary;
    }
};
int main(){
  Employee obj;
  obj.setSalary(50000);
  cout << obj.getSalary();
  return 0;
}
```

private/data hiding

setter

getter

member access operator

```
class Rectangle{
  private:
    int length;
    int height;
  public:
    void setArea(int len, int hgt){
        length = len;
        height = hgt;
    }
    int getArea(){
        return length * height;
    }
};
int main(){
    Rectangle obj;
    obj.setArea(5, 6);
    cout << "The area is: " << obj.getArea() << endl;
    return 0;
}
```

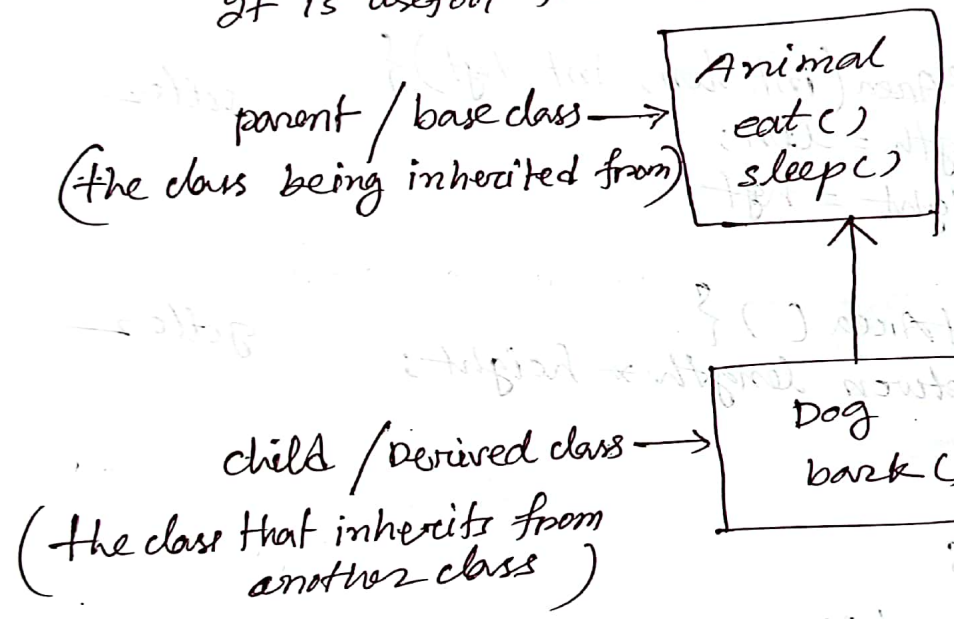
data hiding

setter

getter

Inheritance

Inheritance is a technique to inherit attributes and methods from one class to another.
It is useful for code reusability.



To inherit from a class, use the : symbol.

Syntax

```
class Parent{

};
class child : public Parent{

};
```

: is used to indicate that a class is a subclass (derived class) of another class, which is its superclass (base class).

Types
- public
- private
- protected
- single / simple
- Multi-level
- multiple
- Hierarchical
- Hybrid

Single Inheritance

```
// base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut" << endl;
    }
};
// Derived class
class Car : public Vehicle {
  public:
    string model = "Mustang";
};
int main() {
  Car myObj;
  myObj.honk();
  cout << myObj.brand << myObj.model << endl;
  return 0;
}
```

Output

```
Tuut, tuut
Ford Mustang
```

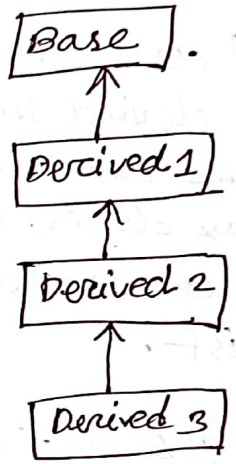
A class which contain only one base class and only one derive class is called single inheritance

```
// Base class
class Animal{
    public:
        void eat(){
            cout<<"I can eat"<<endl;
        }void sleep(){
            cout<<"I can sleep"<<endl;
        }
};
// derived class
class Dog: public Animal{
    public:
        void bark(){
            cout<<"I can bark! woof woof"<<endl;
        }
};
int main(){
    Dog myObj;
    myObj.eat();
    myObj.sleep();
    myObj.bark();
    return 0;
}
```

Multi-level inheritance

A class can also be derived from one class, which is already derived from another class.

```
class A {
};
class B : public A{
};
class C : public B {
};
```



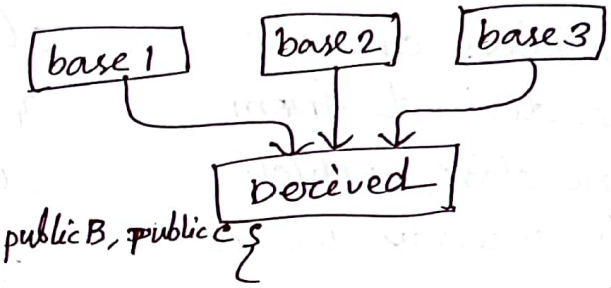

```
// base class (parent)
class myClass {
  public:
    void fun() {
      cout << "gfg" << endl;
    }
};
// derived class (child)
class Child : public myClass {
};
// derived class (GrandChild)
class GrandChild : public Child {
};
```

```
int main() {
  GrandChild myObj;
  myObj.fun();
  return 0;
}
```

Multiple Inheritance

A class can also be derived from more than one base classes, using a comma-separated list.

```
class A{ };
class B{ };
class C{ };
class D: public A, public B, public C{
};
```

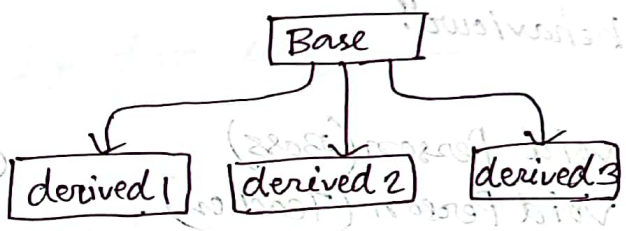



```
// base class 1
class myclass{
  public:
    void fun1(){
      cout<<"gk"<<endl;
    }
};
// base class 2
class anotherclass{
  public:
    void fun2(){
      cout<<"mk"<<endl;
    }
};
// derived class
class child: public myclass, public anotherclass{
};
```

```
int main(){
  child myobj;
  myobj.fun1();
  myobj.fun2();
  return 0;
}
```

Hierarchical Inheritance

If more than one class is inherited from the base class.




```
class A{};
class B: public A{
};
class C: public A{
};
class D: public A{
};
```

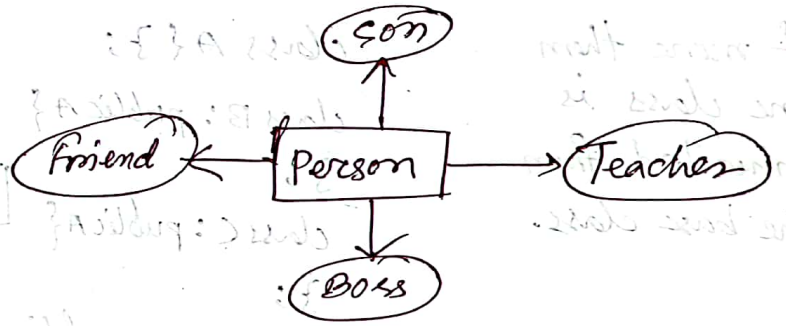
```
// base class
class Animal{
  public:
    void info(){
      cout<<"This Animal"<<endl;
    }
};
// derived class 1
class Dog: public Animal{
    public:
      void bark(){
        cout<<"woof woof"<<endl;
      }
};
```

```
// derived class 2
class Cat: public Animal{
  public:
    void meow(){
      cout<<"meow meow"<<endl;
    }
};
int main(){
    Dog obj1;
    obj1.info();
    obj1.bark();
    Cat obj2;
    obj2.info();
    obj2.meow();
    return 0; }
```

Polymorphism

Polymorphism is a greek word whose meaning is "same object having different behaviour".

Polymorphism = Poly + Morphism
Poly → Many
Morphism → Form
Many + Form → manyform




```
void Person(Boss)
void Person(Teacher)
void Person(son)
void Person(Friend)
```

Types

- Compile time polymorphism (function overloading)
- Run time polymorphism (function overriding)

Template

Template is the frame which defines its actual meaning in a C++ programming. The main purpose of template is to it accept any type of value at the time of program execution.

We can use template in C++ by two ways —

(1) Function template
(2) Class template

Function template : Function template is also known as generic function. A normal function works only one type of value at a time but function template works with different different type at a time.

Syntax

```
template <class type>
return_type function_name (parameter_list)
{
    // code
}
```

```
#include<iostream>
using namespace std;
template <class A>
void print(A x, A y){
    cout<< x <<endl<< y <<endl;
}
int main(){
    print(5,10);
    print('g','k');
    print(4.5, 7.9);
    print("gk");
return 0;
}
```

output

5
10
g
k
4.5
7.9
gk

class template : class template is also know as generic class. We use class template when user doesn't know what kind of value to pass from the parameters.

Syntax

```
template < class type >
class class_name {
    // code
}
```

```
#include<iostream>
using namespace std;
template <class A>
class print{
  public:
    print(A x, A y){
      cout<<x<<endl<<y<<endl;
    }
};
```

```
int main(){
  print<int> obj(5,10);
  return 0;
}
```

Program

```
#include<iostream>
namespace A{
    int a;
    void print(){
        a=15;
        std:: cout<<a<<endl;
    }
}
namespace B{
    int a;
    void print(){
        a=25;
        std:: cout<<a;
    }
}
int main(){
    A:: print();
    B:: print();
    return 0;
}
```

:: → Scope resolution operator

```
#include<iostream>
int main(){
    std:: cout<<"gk";
    return 0;
}
```