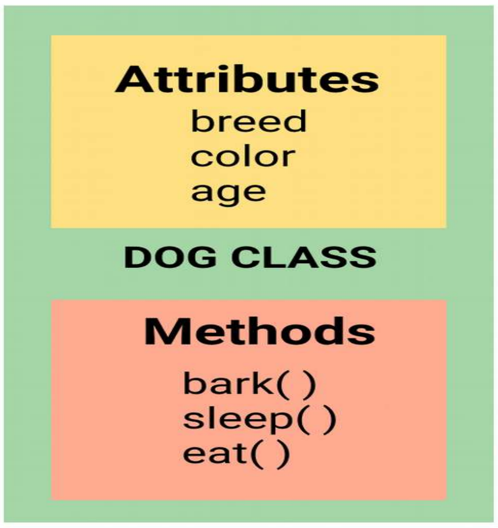
OOP in C++

Introduction to Object Oriented Programming: -

Classes and Objects are basic concepts of Object-Oriented Programming which revolve around the real-life entities.

Class

A class is a user-defined blueprint or prototype from which real-world objects are created. It represents the set of properties or methods that are common to all objects of one type.
We can call class as a collection of objects, which is logical entity and does not take space in the memory.
Example: Dog Class can be represented as shown in the diagram below.



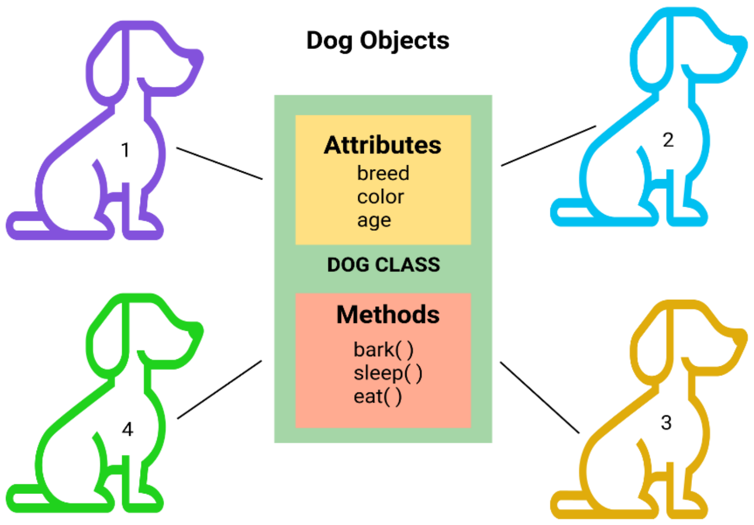
Object

It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods.
Objects having memory addresses and take up some space. They can interact with each other without knowing data and code.

An object consists of:

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

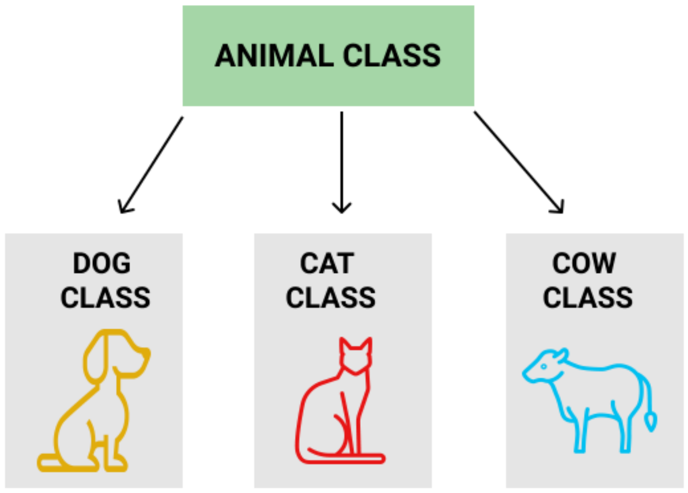
Example: Dog objects created from Dog Class



Inheritance

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

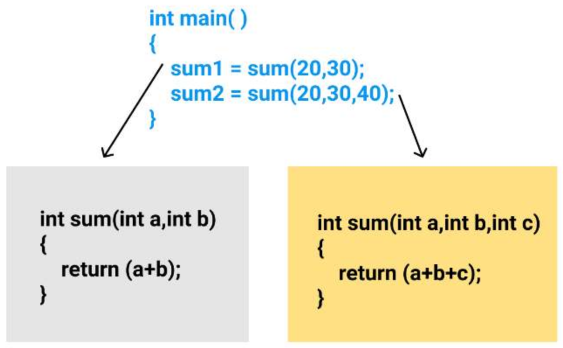
- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we 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.
- **Example:** Dog, Cat, Cow can be Derived Class of Animal Base Class.



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.
Real life examples of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So, the same person possesses have different behavior in different situations. This is called polymorphism.

Example: Suppose we have to write a function to add some integers, sometimes there are 2 integers, sometimes there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



```
int main( )
{
   sum1 = sum(20,30);
   sum2 = sum(20,30,40);
}
```

```
int sum(int a,int b)
{
   return (a+b);
}
```

```
int sum(int a,int b,int c)
{
   return (a+b+c);
}
```

Abstraction

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. **Ex: A car is viewed as a car rather than its individual components.**
Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and help in classifying/grouping the objects.

Consider a **real-life example** of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes, etc. in the car. This is what abstraction is.

Advantages of Abstraction

- It reduces the complexity of viewing the things.
- Avoids code duplication and increases reusability.
- Helps to increase security of an application or program as only important details are provided to the user.

Encapsulation

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
- **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Encapsulation vs Data Abstraction

- Encapsulation is data hiding (information hiding) while Abstraction is detail hiding (implementation hiding).
- While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

Class and Class members:-

Classes are the building blocks of Object-Oriented programming. It is a user-defined data-type which contains data members carrying values that describe the characteristics of the entity the class represents. It contains member methods/functions which describe the behavior of the class-entity. Note that the **class is just a blueprint** that has no physical memory associated with it. **When a class is instantiated, an object is created** which is provided a memory/storage location.

Class Definition and Object Instantiation

The syntax for class declaration is as:

```
class < class-name > {
```

```
< access-modifier >:
    //data-members
    //constructors
    //member-functions
    //destructors
};   //don't forget the semi-colon here !!
```

Each of the above terms (constructors, destructors, access-modifiers etc.) will be explained in detail afterwards.

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

using namespace std;

class Employee {  //Class Declaration

    public:

        string id, name;

        int years;  //experience (in years)

        Employee(string id, string name, int years) {

            this->id = id;

            this->name = name;

            this->years = years;

        }

        void work() {

            cout << "Employee: " << this->id << " is working\n";

        }

};

int main()

{

    //Class Instantiation (Direct)

    Employee emp("GFG123", "John", 3);

    //Class Instantiation (Indirect)

    Employee *emp_ptr = new Employee("GFG456", "James", 4);

    cout << "Employee ID: " << emp.id << endl;

    cout << "Name: " << emp.name << endl;

    cout << "Experience (in years): " << emp.years << endl;

    emp.work();
    cout << endl;

    cout << "Employee ID: " << emp_ptr->id << endl;

    cout << "Name: " << emp_ptr->name << endl;

    cout << "Experience (in years): " << emp_ptr->years << endl;

    emp_ptr->work();

    return 0;

}
```

Output:

```
Employee ID: GFG123
Name: John
Experience (in years): 3
Employee: GFG123 is working

Employee ID: GFG456
Name: James
Experience (in years): 4
Employee: GFG456 is working
```

NOTE: Don't bother about **this** and other keywords which might as of now seem to be unexplained, as they require an additional whole article to explain. We shall cover each of them in detail afterwards.

The statement *Employee emp("GFG123", "John", 3)* inside the *main()* is the object instantiation statement. It is a direct-instantiation of the class. i.e. emp itself is the object stored in RAM, (much like declaring an int, float, struct, etc.). Upon execution of the statement, storage is allocated according to the size of the class and thereafter the constructor - Employee(...) inside the class is run to initialize the data members. We shall study in depth more object initialization formats (there are multiple ways to do so), and more about constructors in later sections.

The statement *Employee *emp_ptr = new Employee("GFG456", "James", 4)* is the indirect way of instantiating class-objects. Here, we use a pointer to point to the object created. We use the **new** keyword and refer to the member entities using **->** operator.

Data Members

Data Members are the identifiers that describe the characteristics and hold important data related to the class. In our Employee class, we have *id, name, years* as the data members. Data members can be anything from primitive data-types (int, char, double, etc.) to collections (arrays, strings, etc.) to user-defined data-types (structs, unions, even other class objects, etc.). Anything that can hold data value can be a data-member.

Member Functions

Member Functions/Methods are class-functions/methods which describe the behavior of the class. i.e. what kind of operation we can perform with objects of this class. e.g. In our Employee example, *work()* is a member function. Member functions in C++ can be defined inside/outside the class, however, it is mandatory to have prototype declaration inside the class if go for an outside-the-class definition. e.g.

```
#include <bits/stdc++.h>
using namespace std;
class Employee {  //Class Declaration
    public:
        string id, name;
        int years;  //experience (in years)
        Employee(string id, string name, int years) {
            this->id = id;
            this->name = name;
            this->years = years;
        }
        //Prototype Declaration
        void work();
};
//Outside-class definition
void Employee::work() {
    cout << "Employee: " << this->id << " is working\n";
}
int main()
{
    //Class Instantiation (Direct)
    Employee emp("GFG123","John",3);
    emp.work();
    return 0;
}
```

Output:

```
Employee: GFG123 is working
```

As we observe, while providing the definition for a class member-function we use the **scope-resolution operator ::** with the **class-name::function-name** format to differentiate it from normal global-scoped functions.

Access modifiers and Abstraction :-

Data abstraction is one of the most essential and important features of object-oriented programming in C++. 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.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes, etc in the car. This is what abstraction is.
Abstraction using Classes: We can implement Abstraction in C++ using classes. Classes help us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Modifiers

There are 3 types of access modifiers in C++, which we discuss in detail below:

- **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

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

using namespace std;

class Circle

{

    public:

        double radius;

        double compute_area() {

            return 3.14 * radius * radius;

        }

};

int main()

{

    Circle obj;

    // accessing public data member outside class

    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << endl;

    cout << "Area is: " << obj.compute_area();

    return 0;

}
```

🎬 **Output:**

```
Radius is: 5.5
Area is: 94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

🎬 **Private**: The class members declared as **private** can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions (discussed later) are allowed to access the private data members of a class.

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

using namespace std;

class Circle
```

```
{
    //private data member
    private:
        double radius;
    //public member function
    public:
        double  compute_area() {
            //member function can access private
            //data member radius
            return 3.14 * radius * radius;
        }
};
int main()
{
    Circle obj;
    //trying to access private data member
    //directly outside the class
    obj.radius = 1.5;
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Error generated by the above code:

```
prog.cpp: In function 'int main()':
prog.cpp:9:16: error: 'double Circle::radius' is private
         double radius;
                ^
prog.cpp:27:9: error: within this context
     obj.radius = 1.5;
         ^
```

The output of the above program will be a compile-time error because we are not allowed to access the private data members of a class directly outside the class. If we comment out the *obj.radius = 1.5;* statement, then the program compiles fine. We can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

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

using namespace std;

class Circle

{

    //private data member

    private:

        double radius;

    //public member functions

    public:

        int getRadius() { return radius; }

        void setRadius(double r) { radius = r; }

        double compute_area() { return 3.14 * radius * radius; }

};

int main()

{

    Circle obj;

    obj.setRadius(5);

    cout << "Radius: " << obj.getRadius() << endl;

    cout << "Area: " << obj.compute_area() << endl;

    return 0;

}
```

🎬 **Output:**

```
Radius: 5
Area: 78.5
```

We can use **getters** and **setter** public functions to indirectly access and manipulate the values of private data-members.

🎬 **Protected**: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

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

using namespace std;
```

```
//Base Class
class Parent
{
    //protected data members
    protected:
        int id_protected;
};
//Derived Class
class Child : public Parent
{
    public:
        void setId(int id) {
            id_protected = id;
        }
        void displayId() {
            cout << "id_protected is: " << id_protected << endl;
        }
};
int main() {
    Child obj;
    //member function of the derived class can
    //access the protected data members of the base class
    obj.setId(81);
    obj.displayId();
    return 0;
}
```

- **Output:**

```
id_protected is: 81
```

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

Constructers:-

A **Constructor** is a member function of a class that initializes objects of a class. In C++, Constructor is automatically called when an object (instance of the class) is created. It is a special member function of the class.

How constructors are different from a normal member function?

A constructor differs from member-functions in the following ways:

- Constructor has the same name as the Class itself.
- Constructors don't have a return type.
- A Constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Default Constructor

It is not mandatory for the programmer to write a constructor for each class. C++ by default provides a default constructors with no parameters, and no statements for the body. Much like being:

```
Employee() {}   //as per our example
#include <bits/stdc++.h>

using namespace std;

class Employee {

public: // public access-modifier

    string id, name;

    int years;

};

int main()
```

```
{
    Employee emp;
    return 0;
}
```

In the above code, once the object is created, the default constructor is called. We can overload the default constructor (covered below).

Constructor Overloading

We need to first get a grasp of overloading first, which is explained as:
Overloading: Having the same name for a member-function/constructor as long as the list of arguments is different is called overloading. In such a case, depending upon the arguments passed, the appropriate overloaded function is deduced and called. An example of constructor overloading:

```
#include <bits/stdc++.h>
using namespace std;
class Employee {
public: // public access-modifier
    string id, name;
    int years;
    Employee()
    {
        id = "";
        name = "";
        years = 0;
    }
    // Overloaded constructor
    Employee(string id, string name, int years)
    {
        this->id = id;
        this->name = name;
        this->years = years;
    }
    // Overloaded constructor
    Employee(string id, string name)
    {
        this->id = id;
        this->name = name;
        years = 0;
    }
    void getDetails()
    {
        cout << "ID: " << id << ", Name: " << name
             << ", Experience: " << years << endl;
    }
};
int main()
{
    // 1st constructor is called
    Employee emp1;
    // 2nd constructor is called
    Employee emp2("GFG123", "John", 4);
    // 3rd constructor is called
    // where years is 0 (no experience for a fresher)
    Employee fresher("GFG456", "James");
    emp1.getDetails();
    emp2.getDetails();
    fresher.getDetails();
    return 0;
}
```

Output:

```
ID:, Name:, Experience: 0
ID: GFG123, Name: John, Experience: 4
```

```
ID: GFG456, Name: James, Experience: 0
```

Member Initializer List

Member Initialization List is a new syntactic construct introduced in modern C++, which allows us to write concise initialization code in constructors. The basic syntax is as:

```
Constructor(< arguments >) : < mem1(arg1), mem2(arg2), ...., > {
    //additional code to execute after initialization
}
```

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

using namespace std;

class Employee {

public: // public access-modifier

    string id, name;

    int years;

    Employee(string id, string name, int years)

        : id(id), name(name), years(years)

    {

        // extra code to run after initialization

    }

    // does the same as:

    // Employee(string id, string name, int years) {

    // this->id = id;

    // this->name = name;

    // this->years = years;

    //     // extra-code to run after initialization

    // }

    void getDetails()

    {

        cout << "ID: " << id << ", Name: " << name

             << ", Experience: " << years << endl;

    }
};

int main()

{

    Employee emp("GFG123", "John", 4);

    emp.getDetails();

    return 0;

}
```

Output:

```
ID: GFG123, Name: John, Experience: 4
```

Destructors

Destructors like constructors are special members of a class that is executed once the lifetime of the object expires. It is like the final clean-up code required before deleting the class instance. Unlike JAVA, C++ doesn't perform automatic garbage collection. Hence, it often becomes the responsibility of the developer to de-allocate memory (not required further). A classic example:

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

using namespace std;

char* p_chr;

class String {

public:

    char* s;

    int size;

    String(char* c)

    {

        size = strlen(c);

        s = new char[size + 1];

        p_chr = s; // assign to global variable

        strcpy(s, c);

    }

};
```

```
void func()

{

    String str("Hello World");

}

int main()

{

    func();

    cout << p_chr << endl;

    return 0;

}
```

Output:

```
Hello World
```

In the above code, we dynamically create a character array in our constructor, where we copy the string passed as an argument. Since it is a dynamically-allocated memory, once, the lifetime of *str* object expires (inside the *func()* call), still the memory is not de-allocated, as in main(), when we print *p_chr*, we get the string. Thus, we can see that proper clean-up of pointer references doesn't occur. Thus, we need destructors where we would explicitly de-allocate the memory and perform other clean-up code. Syntax of destructor:

```
~ Classname { //clean-up code }
```

The above code with destructors:

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

using namespace std;

char* p_chr;

class String {

public:

    char* s;

    int size;

    String(char* c)

    {

        size = strlen(c);

        s = new char[size + 1];

        p_chr = s;
        strcpy(s, c);
    }
    // Destructor
    ~String()
    {
        delete[] s;
    }
};
void func()
{
    String str("Hello World");
}
int main()
{
    func();
    cout << p_chr << endl;
    return 0;
}
```

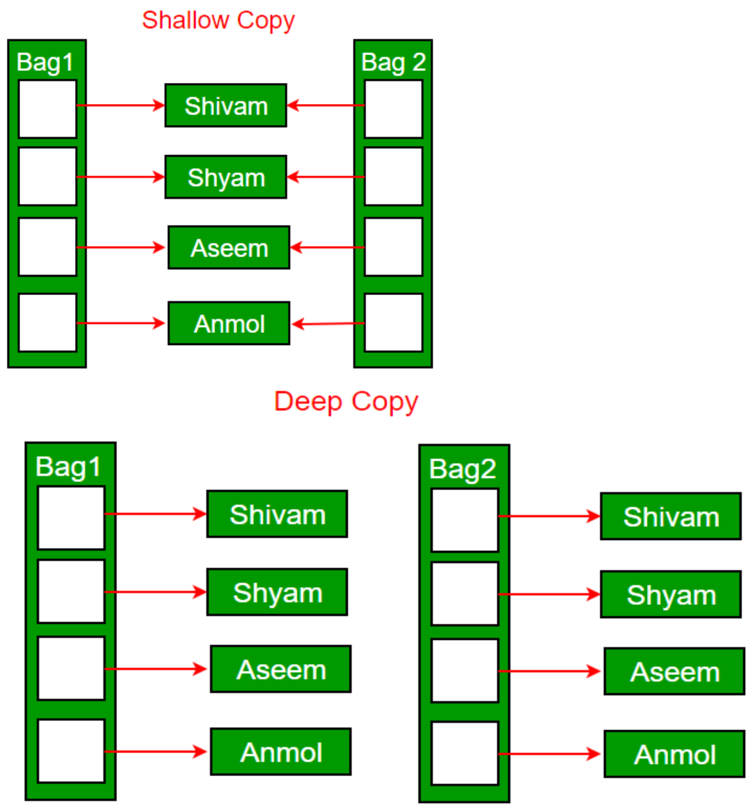
The above code prints nothing because *p_chr* reference is gone due to destructor call.

Copy Constructor

A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
Classname (const Classname &object);
```

Copy constructor, in general, is not required to be defined by the user, as the compiler automatically provides a default copy constructor. However, this default copy constructor performs a shallow copy only (i.e. copy values only). This results in pointer variables pointing the same instances upon copy. We need to define our own copy constructor only if an object has pointers or any runtime allocation of theresource like file handle, a network connection.etc.



As an example:

```
#include <bits/stdc++.h>
using namespace std;
class Array {
public:
    int n;
    int* ref;
    Array(int n)
        : n(n)
    {
        ref = new int[n];
        for (int i = 0; i < n; i++)
            *(ref + i) = i;
    }
};
int main()
{
    Array arr1(10);
    // copy constructor called
    // at this point
    Array arr2 = arr1;
    // changing n-value in 2nd instance
    arr2.n = 5;
    // changing array-values in 2nd instance
    for (int i = 0; i < 10; i++)
        *(arr2.ref + i) *= 2;
    cout << "n-value of 1st instance: " << arr1.n << endl;
    cout << "Array values of 1st instance:\n";
    for (int i = 0; i < 10; i++)
        cout << *(arr1.ref + i) << " ";
    cout << endl;
    return 0;
}
```

Output:

```
n-value of 1st instance: 10
Array values of 1st instance:
0 2 4 6 8 10 12 14 16 18
```

In the above code, we find the value of member *n* for *t1* not modified as it is not a pointer value. Thus, upon copying new instance of *n* got created for *t2*. Any change to n in *t2* didn't change *t1.n*. However, *t1.ref* is a pointer. So, upon copy-constructor call, the address value got copied to *t2.ref*, and thus, any change at t2.ref (as we here are multiplying by 2), gets reflected at t1.ref also because both of them are pointing to the same array. This is an example of a shallow-copy. To fix this, we write our custom copy-constructor:

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

```
class Array {
public:
    int n;
    int* ref;
    Array(int n)
        : n(n)
    {
        ref = new int[n];
        for (int i = 0; i < n; i++)
            *(ref + i) = i;
    }
    // copy-contructor definition
    Array(const Array& obj)
        : n(obj.n)
    {
        ref = new int[n];
        for (int i = 0; i < n; i++)
            *(ref + i) = *(obj.ref + i);
    }
};
int main()
{
    Array arr1(10);
    // copy constructor called
    // at this point
    Array arr2 = arr1;
    // changing n-value in 2nd instance
    arr2.n = 5;
    // changing array-values in 2nd instance
    for (int i = 0; i < 10; i++)
```

```
        *(arr2.ref + i) *= 2;
    cout << "n-value of 1st instance: " << arr1.n << endl;
    cout << "Array values of 1st instance:\n";
    for (int i = 0; i < 10; i++)
        cout << *(arr1.ref + i) << " ";
    cout << endl;
    return 0;
}
```

Output:

```
n-value of 1st instance: 10
Array values of 1st instance:
0 1 2 3 4 5 6 7 8 9
```

We can see that in our copy-constructor, we re-create a dynamic memory for the array and then copy the values. This results in a deep-copy. Meaning changes in *arr2* doesn't affect *arr1*.

‘this’ pointer:-

To understand *'this'* pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
The compiler supplies an implicit pointer along with the names of the functions as *'this'*.
The *'this'* pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. *'this'* pointer is a constant pointer that holds the memory address of the current object. *'this'* pointer is not available in static member functions as static member functions can be called without any object (with class name).
For class X, the type of this pointer is 'X* const'. Also, if a member function of X is declared as const, then the type of *'this'* pointer is 'const X *const'.

Following are the situations where *'this'* pointer is used:

When local variable's name is same as member's name

```
#include <bits/stdc++.h>
using namespace std;
class Employee {
    public:
        string id, name;
        int years;
        //'this' keyword here retrieves the object's
        //instance variables: id, name, years hidden
        //by their same-name local counterparts
        Employee(string id, string name, int years) {
            this->id = id;
            this->name = name;
            this->years = years;
        }
        //here we don't need to use 'this' keyword
        //explicitly as their are no local variables
        //with the same name. So, compiler automatically
        //deduces it as instance variables
        void printDetails() {
            cout << "ID: " << id
                 << ", Name: " << name
                 << ", Experience: " << years;
        }
};
int main()
{
    Employee emp("GFG123", "John", 4);
    emp.printDetails();
    return 0;
}
```

Output:

```
ID: GFG123, Name: John, Experience: 4
```

For constructors, initializer list can also be used when parameter name is same as member's name.

To return reference to the calling object

/* Reference to the calling object can be returned */

Test& Test::func ()

```
{
   // Some processing
   return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

```
#include <bits/stdc++.h>
using namespace std;
class Employee {
    private:
        string id, name;
        int years;
    public:
        Employee setId(string id) {
            this->id = id;
            return *this;
        }
        Employee setName(string name) {
            this->name = name;
            return *this;
        }
        Employee setYears(int years) {
            this->years = years;
            return *this;
        }
        void printDetails() {
            cout << "ID: " << id
                 << ", Name: " << name
                 << ", Experience: " << years;
        }
};
int main()
{
    Employee emp;
    emp.setId("GFG123").setName("John").setYears(4).printDetails();
    return 0;
}
```

Output:

```
ID: GFG123, Name: John, Experience: 4
```

In the above code, each time we call the setter methods, the employee instance we are referring to is returned using the *'this'* pointer. We can thus re-use this instance to chain more method calls.

Static Data members and method:-

We looked at the static keyword earlier in the respect of functions. How declaring an identifier as static gives its lifetime scope of the program, such that it retains its value even after successive function calls. But, static has a different meaning when it comes to classes.

Static data-members

Declaring a data member in a class as static gives it class-scope. i.e. The variable no longer remains specific and bound to one particular object instance. Thereafter, changing the value from one instance reflects over all the instances. As an example:

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

using namespace std;

class Test {

    public:

        static int x;

};

/*static members need to be

defined outside the class*/

int Test::x = 1;

int main()

{

    Test t1, t2;

    cout << "Access from instance: " << t1.x << endl

         << "Access from Class: " << Test::x << endl;

    Test::x = 5;

    cout << "t1.x: " << t1.x << endl

         << "t2.x: " << t2.x << endl

         << "Test::x: " << Test::x << endl;

    return 0;

}
```

Output:

```
Access from instance: 1
Access from Class: 1
t1.x: 5
t2.x: 5
Test::x: 5
```

As we can see from the above program, any change made to the member *x*, it gets reflected over both instances *t1* and *t2*. We can also access static members using Class-name and scope-resolution ~ *Test::x*.

NOTE: When we declare a static variable inside a class, we are just telling the compiler the existence of such a variable. It is treated as a variable with a global scope and is initialized only when the program starts. No memory is allocated at that point. Thus, we can't directly initialize a static data-member along with the declaration. i.e. *static int x = 1;* is reported as a compilation error. We must explicitly initialize it outside the class. Another thing to note is that this initializer statement works regardless of whether we declare the static variable as public, private or protected.

Static member-functions

Static Member Functions are similar to the static data-members implying that they too have class-scope. In addition to that, they are allowed to access only other static fields (data & member). However, they can be called using object instances. (i.e. *obj.static_method()* is valid). Some important points regarding static member functions are given below:

- Static member-functions can't access non-static data-members.
- Static member-functions can't access other non-static member-functions.
- Static member-functions have no *this pointer, as it is not associated with any particular instance.
- There is no concept of a static constructor.
- Static member-functions are useful for accessing static data-members which are not declared public.

As an example:

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

using namespace std;

class Test {

    private:

        static int x;

    public:

        //set private static integer x

        static void setX(int x) { Test::x = x; }

        //get private static integer x

        static int getX() { return x; }

};

//Initializer statement is valid

//even though variable is declared

//private
```

```
int Test::x = 1;

int main()

{

    Test::setX(5);

    cout << Test::getX();

    return 0;

}
```

Output:

```
5
```

Friend Function in C++:-

For the sake of data-hiding, we emphasize the use of a private modifier for critical data-members. However, there might be some situations where multiple classes need to work together closely, so much so that they require access to each other's private members too. Well, we can solve the problem by using getters and setters, but it would be tedious to do so for all the private members. Instead, we use the ***friend*** keyword to provide access to the private fields to outside entities (global functions, other class-member functions etc.).

Friend Class

A *Friend Class* can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private members of other classes.e.g.A LinkedList class may be allowed to access private members of Node.

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

using namespace std;

class Node

{

    private:

        int key;

        Node *next;

        Node(int key) : key(key), next(nullptr) {}

    public:

        friend class LinkedList;
```

```
};
class LinkedList
{
    public:
        Node *root;
        LinkedList(int key) {
            root = new Node(key);
        }
        void insert(int key) {
            Node *trav = root;
            while (trav->next != nullptr)
                trav = trav->next;
            trav->next = new Node(key);
        }
        void print() {
            Node *trav = root;
            while (trav != nullptr) {
                cout << trav->key << " ";
                trav = trav->next;
            }
            cout << endl;
        }
};
int main()
{
     LinkedList list(0);
     list.insert(1);
     list.insert(2);
     list.insert(3);
     list.insert(4);
     list.print();
     return 0;
}
```

Output:

```
0 1 2 3 4
```

As we can see in the above code, *LinkedList* has access to all the private fields of *Node* class, because it has been declared as a friend inside the Node class.

Friend Function

Like a friend class, a friend function can be given special access to private and protected members. A friend function can be:

- A Member Function of another class

```
#include <bits/stdc++.h>
using namespace std;
//forward-declaration is
//necessary for usage in A
//as B is not defined yet
class B;
class A
{
    public:
        void showB(B &x);
};
class B
{
    private:
        int b;
    public:
        B() : b(0) {}
        //Friend function Declaration
        friend void A::showB(B &x);
};
//Friend Member Function Definition
void A::showB(B &x)
{
    cout << "B::b = " << x.b;
}
int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

Output:

```
B::b = 0
```

🎬 A global Function

```
#include <bits/stdc++.h>
using namespace std;
class A
{
    private:
        int a;
    public:
        A() : a(0) {}
        //global friend function
        friend void showA(A&);
};
void showA(A &x) {
    std::cout << "A::a=" << x.a;
}
int main()
```

```
{
    A a;
    showA(a);
    return 0;
}
```

Output:

```
A::a=0
```

Anonymous objects in C++:-

Anonymous objects are created without assigning a reference to them. Thus, they can be used only once (i.e. in the same statement only). e.g.

```
#include <bits/stdc++.h>
using namespace std;
class Math {
    public:
        int add(int x, int y) { return x + y; }
        int mul(int x, int y) { return x * y; }
};
int main()
{
    cout << Math().add(5,6) << endl;
    cout << Math().mul(5,6) << endl;
    return 0;
}
```

Output:

```
11
30
```

In the above code, in each of the *cout* statements, a new instance of the *Math* class is instantiated and then the said operation is performed. Thus, there is no way to reference those objects afterword since there is no reference attached to them.

Operator overloading in C++:-

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this

ability is known as operator overloading.
For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.
Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc. An example for the case of complex numbers addition:

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

using namespace std;

class Complex {

    private:

        int real, imag;

    public:

        Complex(int r = 0, int i =0)  {real = r;   imag = i;}

        // This is automatically called when '+' is used with

        // between two Complex objects

        Complex operator + (Complex const &obj) {

             Complex res;

             res.real = real + obj.real;

             res.imag = imag + obj.imag;

             return res;

        }

        void print() { cout << real << " + i" << imag << endl; }

};

int main()

{

    Complex c1(10, 5), c2(2, 4);

    Complex c3 = c1 + c2; // An example call to "operator+"

    c3.print();

}
```

Output:

```
12 + i9
```

What is the difference between operator functions and normal functions?
Operator functions are the same as normal functions. The only difference is that the name of an operator function is always the operator keyword followed by the symbol of operator and operator functions are called when the corresponding operator is used.
Following is an example of global operator function.

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

using namespace std;

class Complex

{

    private:

        int real, imag;

    public:

        Complex(int r = 0, int i =0)  {real = r;   imag = i;}

        void print() { cout << real << " + i" << imag << endl; }

        // The global operator function is made friend of this class 
so

        // that it can access private members

        friend Complex operator + (Complex const &, Complex const &);

};

Complex operator + (Complex const &c1, Complex const &c2)

{

    return Complex(c1.real + c2.real, c1.imag + c2.imag);

}

int main()

{

    Complex c1(10, 5), c2(2, 4);

    Complex c3 = c1 + c2; // An example call to "operator+"

    c3.print();

    return 0;

}
```

Output:
```
12 + i9
```

Can we overload all operators? Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

- **.** (dot)
- **::** (scope-resolution)
- **?:** (ternary-operator)
- **sizeof**

Important points about operator overloading

1. For operator overloading to work, at least one of the operands must be an user-defined class object.
2. *Assignment Operator* - Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor).
3. *Conversion Operator* - We can also write conversion operators that can be used to convert one type to another type. As an example:

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

using namespace std;

class Fraction

{

    int num, den;

    public:

        Fraction(int n,  int d) : num(n), den(d) {}

        // conversion operator: return float value of fraction

        operator float() const {

            return float(num) / float(den);

        }

};

int main()
```

```
{
    Fraction f(2,5);
    float val = f;
    cout << val;
    return 0;
}
```

🎬 **Output:**

```
0.4
```

4 Overloaded conversion operators must be a member method. Other operators can either be a member method or a global method.

5 Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

```
#include <bits/stdc++.h>
using namespace std;
class Point
{
    private:
        int x, y;
    public:
        Point(int i=0, int j=0) : x(i), y(j) {}
        void print() {
            cout << endl << " x = " << x << ", y = " << y;
        }
};
int main() {
    Point t(20, 20);
    t.print();
    t = 30;   // Member x of t becomes 30
    t.print();
    return 0;
}
```

Output:

```
 x = 20, y = 20
 x = 30, y = 0
```

Overloading vs Overriding :-

Overloading

Overloading is a feature that allows a class to have multiple methods with the same name, the only difference lies in their list of arguments. i.e. The argument list for each of the methods differ, and it helps the compiler or the run-time environment to identify which method to call depending upon the parameters passed. Constructors can be overloaded too.

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

using namespace std;

class Math

{

    public:

        //Overloaded add() methods

        static int add(int x, int y) { return x + y; }

        static void add(int a[], int b[], int sum[], int n) {

            for (int i=0;i<n;i++)

                //use the 1st form to get addition of

                //two numbers

                sum[i] = add(a[i], b[i]);

        }

        //Overloaded mul() methods

        static int mul(int x, int y) { return x * y; }

        static void mul(int a[], int b[], int prod[], int n) {

            for (int i=0;i<n;i++)

                //use the 2nd form to get product

                //of two numbers

                prod[i] = mul(a[i], b[i]);

        }
};

int main()

{

    int a[] = {1, 2, 3, 4, 5};

    int b[] = {9, 8, 7, 6, 5};

    int sum[5], prod[5];

    Math::add(a, b, sum, 5);

    Math::mul(a, b, prod, 5);

    for (int i=0; i<5; i++)

        cout << sum[i] << " ";

    cout << endl;

    for (int i=0; i<5; i++)

        cout << prod[i] << " ";

    cout << endl;

    return 0;

}
```

Output:

```
10 10 10 10 10
9 16 21 24 25
```

As shown in the above program, we have 2 sets of overloaded functions, namely integer addition-array addition and integer product-array-product (sort of dot product). Depending on the type of arguments passed, the compiler decides which one of the set of overloaded functions it needs to call.

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

using namespace std;

class Math

{

    public:

        //Overloaded add() methods

        static int add(int x, int y) {

            cout << "1st form called: ";

            return x + y;
        }
        static double add(int x, double y) {
            cout << "2nd form called: ";
            return x + y;
        }
        static double add(double x, double y) {
            cout << "3rd form called: ";
            return x + y;
        }
};
int main()
{
    cout << Math::add(2, 3) << endl;
    cout << Math::add(2, 3.0) << endl;
    cout << Math::add(2.0, 3.0) << endl;
    return 0;
}
```

Output:

```
1st form called: 5
2nd form called: 5
3rd form called: 5
```

NOTE: Return-type is not a factor of uniqueness: Having the list of arguments identical with different return-types doesn't remove ambiguity. Hence, the following declarations are invalid:

int add(int x, int y) { ... }

double add(int x, int y) { ... }

Overriding

Inheritance allows Derived Classes to inherit Base class data-members as well as member functions. Thus, if we call base class function with derived class instance, it would run perfectly:

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

```
using namespace std;

class Base {

    public:

        void whoami() {

            cout << "I'm Base Class!!\n";

        }

};

class Derived : public Base {

};

int main()

{

    Base b;

    Derived d;

    b.whoami();

    d.whoami();

    return 0;

}
```

Output:

```
I'm Base Class!!
I'm Base Class!!
```

In the above program, we see no definition of *whoami()* method inside the Derived Class. So when we call *d.whoami()*, the compiler first looks into the Derived Class for its definition. Then, it looks inside its Parent class, where it finds the definition and that version is called.

However, suppose we want to change the behavior of the inherited method inside our Derived Class. This feature provided by OOP is called Overriding. To do that, we provide the full method definition as usual:

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

using namespace std;

class Base {

    public:

        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};
class Derived : public Base {
    public:
        void whoami() {
            cout << "I'm Derived Class!!\n";
        }
};
int main()
{
    Base b;
    Derived d;
    b.whoami();
    d.whoami();
    return 0;
}
```

Output:

```
I'm Base Class!!
I'm Derived Class!!
```

Sometimes, we don't want to replace the complete functionality of the inherited methods, but add some extra functionality to it. We can do so by calling the Base Class's method from the Derived class and then continue with our overriding added statements.

```
#include <bits/stdc++.h>
using namespace std;
class Base {
    public:
        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};
```

```
class Derived : public Base {

    public:

        void whoami() {

            // call Base class's version

            Base::whoami();

            cout << "I'm Derived Class!!\n";

        }

};

int main()

{

    Derived d;

    d.whoami();

    return 0;

}
```

Output:

```
I'm Base Class!!
I'm Derived Class!!
```

We call the Base class's *whoami()* using the scope-resolution operator with the classname preceding it.

Virtual Functions and Polymorphism :-

Run-time Polymorphism

Consider a situation where we have Derived Class which has overridden some method of the Base Class. Polymorphism allows us to have a Base Class reference a Derived Class Object. Then, in such a case, which function to call (Base or Derived) is decided at run-time, as the compiler is unable to resolve which one to call during compilation. Below is a classic example of Run-time Polymorphism:

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

using namespace std;

class Base

{

    public:
        void whoami() {
            cout << "I'm Base\n";
        }
};
class Derived: public Base
{
    public:
        void whoami() {
            cout << "I'm Derived\n";
        }
};
int main(void)
{
    Base *b_ptr = new Derived;
    //run-time polymorphism
    b_ptr->whoami();
    return 0;
}
```

Output:

```
I'm Base
```

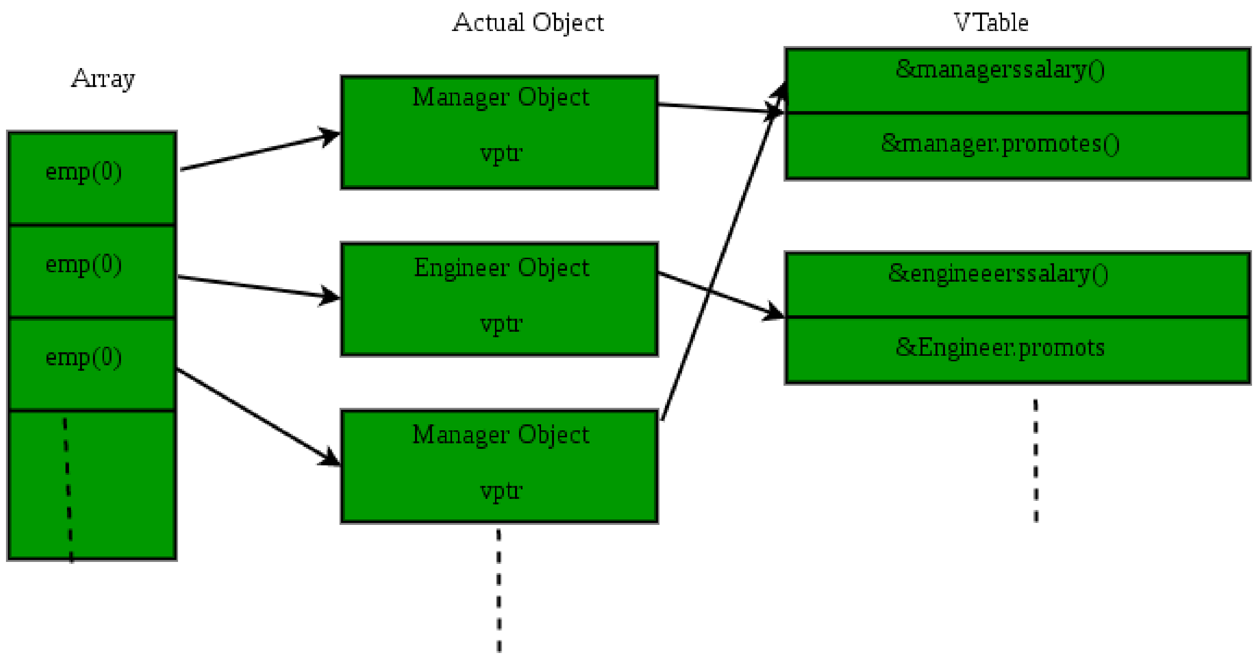
In the above code, since, the pointer reference is of Base-type, so at runtime, it is resolved to call the *whoami()* version of the Base class. Therefore, *"I'm Base"* is printed. But, suppose we want to call the Derived Class's version of the function. Here, comes the play of **virtual** keyword.

Virtual Functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*,.. etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. e.g. we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```
class Employee
{
    public:
        virtual void raiseSalary()
        {  /* common raise salary code */  }
        virtual void promote()
        { /* common promote code */ }
};
class Manager: public Employee
{
    public:
        virtual void raiseSalary()
        {  /* Manager specific raise salary code, may contain
              increment of manager-specific incentives*/  }
        virtual void promote()
        { /* Manager specific promote */ }
};
// Similarly, there may be other types of employees
// We need a very simple function to increment the salary of all employees
// Note that emp[] is an array of pointers and actual pointed objects can
// be any type of employees. This function should ideally be in a class
// like Organization, we have made it global to keep things simple
void globalRaiseSalary(Employee *emp[], int n)
{
    for (int i = 0; i < n; i++)
        emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()
                               // according to the actual object, not
                               // according to the type of pointer
}
```

like *globalRaiseSalary()*, there can be many other operations that can be appropriately done on a list of employees without even knowing the type of actual object.
Virtual functions are so useful that later languages like Java kept all the methods as virtual by default.

How does compiler do this magic of late resolution?
Compiler maintains two things to this magic :



vtable - A table of function pointers. It is maintained per class.
vptr - A pointer to vtable. It is maintained per object.

Compiler adds additional code at two places to maintain and use *vptr*.
1) Code in every constructor. This code sets vptr of the object being created. This code sets *vptr* to point to *vtable* of the class.
2) Code with polymorphic function call (e.g. *bp->show()* in above code). Wherever a polymorphic call is made, compiler inserts code to first look for *vptr* using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, address of derived derived class function *show()* is accessed and called.

Multiple Inheritance and Diamond Problem :-

Multiple Inheritance is a feature of C++ where a class can inherit from more than one class.
The constructors of inherited classes are called in the same order in which they are

inherited. For example, in the following program, B's constructor is called before A's constructor.

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

using namespace std;

class A

{

    public:

      A() { cout << "A's constructor called" << endl; }

};

class B

{

    public:

      B() { cout << "B's constructor called" << endl; }

};

class C: public B, public A  // Note the order

{

    public:

      C() { cout << "C's constructor called" << endl; }

};

int main()

{

    C c;

    return 0;

}
```

Output:

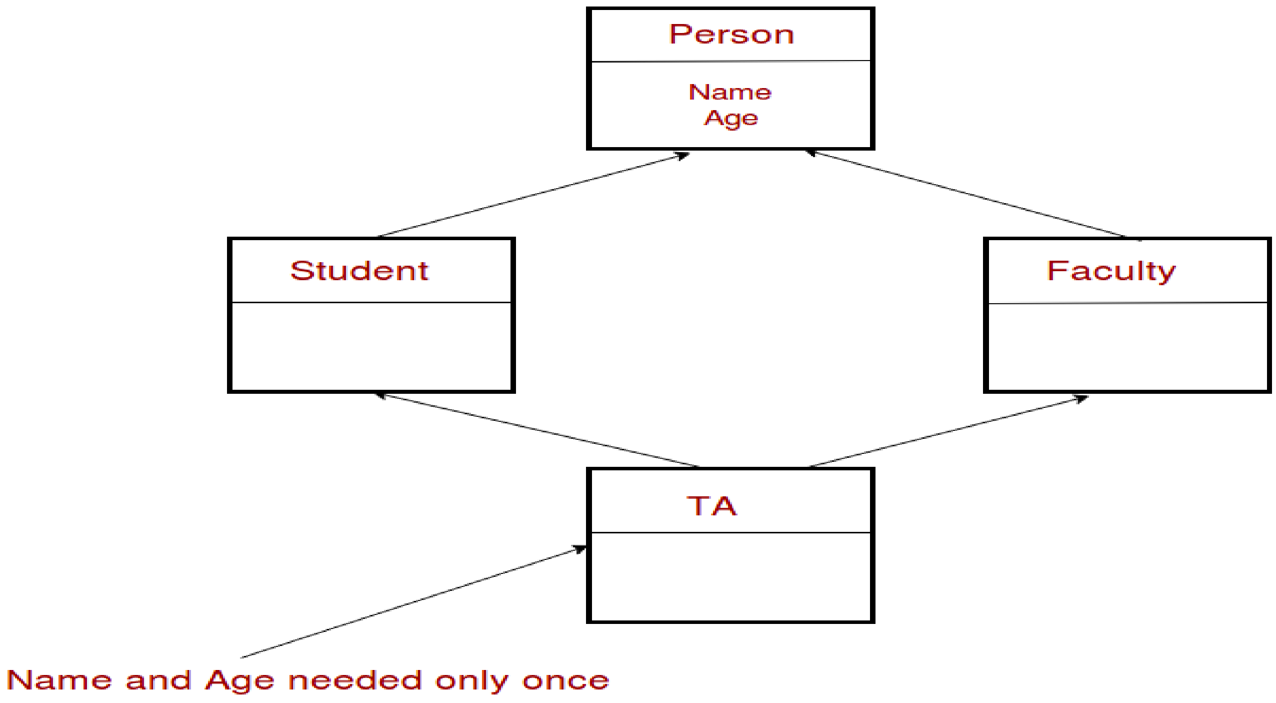
```
B's constructor called
A's constructor called
C's constructor called
```

The Diamond Problem The diamond problem occurs when two superclasses of a class have a common base class. As an example, in the following diagram, the TA class gets two copies of all attributes of the Person class that leads to an ambiguity.

```
#include <bits/stdc++.h>
using namespace std;
class Person
{
    // Data members of person
    public:
        Person(int x) {
            cout << "Person::Person(int ) called" << endl;
        }
};
class Faculty : public Person
{
    // data members of Faculty
    public:
        Faculty(int x) : Person(x) {
            cout << "Faculty::Faculty(int) called" << endl;
        }
};
class Student : public Person
{
    // data members of Student
    public:
        Student(int x) : Person(x) {
            cout << "Student::Student(int) called" << endl;
        }
};
class TA : public Faculty, public Student
{
    public:
        TA(int x): Student(x), Faculty(x) {
            cout << "TA::TA(int) called" << endl;
        }
};
int main()  {
    TA ta(30);
    return 0;
}
```

Output:

```
Person::Person(int ) called
Faculty::Faculty(int) called
Person::Person(int ) called
Student::Student(int) called
TA::TA(int) called
```



In the above program, the constructor of the *Person* is called two times. The destructor of *Person* will also be called two times when the object *ta* is destructed. So object *ta* has two copies of all members of *Person*, this causes ambiguity. The solution to this problem is using **virtual classes**. We make the classes *Faculty* and *Student* as virtual base classes to avoid two copies of *Person* in the *TA* class. As an example, consider the following program:

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

```
using namespace std;

class Person {

    public:

        Person(int x) { cout << "Person::Person(int) called" <<
endl; }

        Person() { cout << "Person::Person() called" << endl; }

};

class Faculty : virtual public Person {

    public:

        Faculty(int x) : Person(x)   {

            cout << "Faculty::Faculty(int) called" << endl;

        }

};

class Student : virtual public Person {

    public:

        Student(int x) : Person(x) {

            cout << "Student::Student(int) called" << endl;

        }

};

class TA : public Faculty, public Student  {

    public:

        TA(int x) : Student(x), Faculty(x) {

            cout << "TA::TA(int) called" << endl;

        }

};

int main()  {

    TA ta(30);

}
```

Output:

```
Person::Person() called
Faculty::Faculty(int) called
Student::Student(int) called
```

```
TA::TA(int) called
```

In the above program, the constructor of the *Person* is called once. One important thing to note in the above output is the default constructor of *Person* is called. When we use the virtual keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call the parameterized constructor

How to call the parameterized constructor of the *Person* class? The constructor has to be called in *TA* class:

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

using namespace std;

class Person

{

    public:

        Person(int x) { cout << "Person::Person(int) called" <<
endl; }

        Person() { cout << "Person::Person() called" << endl; }

};

class Faculty : virtual public Person

{

    public:

        Faculty(int x) : Person(x) {

            cout << "Faculty::Faculty(int) called" << endl;

        }

};

class Student : virtual public Person

{

    public:

        Student(int x) : Person(x) {

            cout << "Student::Student(int) called" << endl;

        }

};
```

```
class TA : public Faculty, public Student  {

    public:

        TA(int x) : Student(x), Faculty(x), Person(x) {

            cout << "TA::TA(int) called" << endl;

        }

};

int main()  {

    TA ta(30);

    return 0;

}
```

Output:

```
Person::Person(int) called
Faculty::Faculty(int) called
Student::Student(int) called
TA::TA(int) called
```

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through a parent class. It is allowed only when the virtual keyword is used.

Structs and Classes in C++ :-

In C++, a structure is the same as a class except for a few differences. The most important of them is security. A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details. Following are the points that expound on this difference:

1. Members of a class are private by default and members of a struct are public by default. e.g.

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

using namespace std;

struct A {

    int x; // x is public

};

class B {
    int x; // x is private

};

int main()

{

    A a;

    B b;

    cout << a.x << endl;

    cout << b.x << endl;

    return 0;

}
```

🎬 Error generated by above code:

```
prog.cpp: In function 'int main()':
prog.cpp:10:9: error: 'int B::x' is private
     int x; // x is private
         ^
prog.cpp:19:15: error: within this context
     cout << b.x << endl;
               ^
```

Above code generates compilation-error because we tried to access data-member *x* for instance *b* (which is private).

2. When deriving a **struct from a class/struct**, default access-specifier for a base class/struct is **public**. And when **deriving a class**, the default access specifier is **private**.

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

using namespace std;

class Base

{

    public:

        int x; // x is public

};

class Derived1 : Base

//equivalent to private Base

{

};
```

```
struct Derived2 : Base

//equivalent to public Base

{

};

int main()

{

    Derived1 d1;  //class

    Derived2 d2;  //struct

    cout << d1.x << endl;

    cout << d2.x << endl;

    return 0;

}
```

Error generated by above code:

```
prog.cpp: In function 'int main()':
prog.cpp:8:13: error: 'int Base::x' is inaccessible
         int x; // x is public
             ^
prog.cpp:26:16: error: within this context
     cout << d1.x << endl;
                ^
```

The above code generates compilation error because of the access statement *d1.x*. Since we didn't specify the access-modifier for the Base class, *x* became private in Derived Class.